

SCENARIST™ AUTOMATED SCENARIO GENERATION SYSTEM

FINAL REPORT

Final Report for the Project

Research in Artificial Intelligence
for
Noncommunications Electronic Warfare Systems

Contract No. DAAB07-89-C-P017

August 31, 1991

Report No. TR-C207-8

Prepared by:

Dr. J. George Caldwell, Principal Investigator
Vista Research Corporation
3826 Snead Drive
Sierra Vista, Arizona 85635
(602)378-2130 (602)790-0500

Submitted to:

Center for Electronic Warfare
/Reconnaissance, Surveillance
and Target Acquisition
US Army Communications-Electronics Command
ATTN: AMSEL-RD-EW-C
Fort Monmouth, NJ 07703-5000

Copyright (C) 1991 Vista Research Corporation

Report Documentation Page

TR-C207-8

Scenarist™ Automated Scenario
31 Aug 91
Generation System: Final Report

Dr. J. George Caldwell
TR-C207-8

Vista Research Corporation
C-207
3826 Snead Drive
Sierra Vista, AZ 85635
DAAB07-89-C-P017
(602) 378-2130, (602) 790-0500

US Army Communications-Electronics Command
Final, 9/1/89-
ATTN: AMSEL-RD-EW-C (Dr. Frank Elmer)
8/31/91
Fort Monmouth, NJ 07703-5000

This report is the final report for the project, "Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems." This project was conducted by Vista Research Corporation under contract no. DAAB07-89-C-P017 from the US Army Communications-Electronics Command. The goal of the project was to develop an automated scenario generation system. The system was named the "Scenarist."

The report describes the work accomplished on the project over its course, the project products, and the Scenarist system. The

Scenarist system is implemented in a microcomputer-based program, which has been documented in a user's manual, a programmer's manual, and a variable glossary. This report includes information on system concepts that complements the material in those other manuals, but was not called for in any other required contract deliverable.

Unrestricted
287

Unclassified

Unclassified



Contents

Foreword	6
Summary	7
I. Introduction	10
II. Project Task Descriptions	20
III. Scenarist System Concept	31
IV. System Requirements Analysis	53
V. Scenarist System Top-Level Design.....	69
VI. Scenarist Development Software Top-Level Design.....	92
VII. Scenarist Development Software Detailed Design.....	199
VIII. Scenarist Software Documentation	294
IX. Validation and Verification of the Scenarist System	296
X. Summary of Scenarist Capabilities and Limitations	300

XI. Areas for Future Scenarist System Development	304
References	314
Glossary	318
Distribution List.....	320
Appendix A.....	328
Appendix B	328

Foreword

This report was prepared by the staff of Vista Research Corporation, under Contract No. DAAB07-89-C-P017 to the US Army Communications-Electronics Command. Project staff included Dr. J. George Caldwell, Principal Investigator, Mr. William N. Goodhue, Ms. Sharon K. Hoting, Dr. William O. Rasmussen, Mr. Fletcher A. K. Aleong, Mr. Eric Weiss, Dr. Marty A. Diamond, and Mr. Christopher S. Caldwell. Government monitoring of the project was provided by Dr. Frank Elmer, Acting Chief, SIGINT/ESM Branch, Advanced Concepts Division, Center for Electronic Warfare / Reconnaissance, Surveillance and Target Acquisition (EW/RSTA). The CECOM Project Manager was Dr. Elmer.

We extend our thanks to the many individuals who helped us during the course of the project, particularly Major Louis Poli (SC USAR, CECOM Center for EW/RSTA), who reviewed the Scenarist and provided many comments aimed at making the system more useful; Captain John Aloisio of the US Army Intelligence Center and School, who provided information about the TRAILBLAZER system; Messrs. Jim Cole, Dan Searls, Grady Taylor and Steve Matsuura of the US Army Electronic Proving Ground's Electromagnetic Environmental Test Facility, who provided input to the system requirements analysis and sample unit data; and Mr. Oliver Cathey of SPARTA, Inc., who also provided information on user requirements.

Summary

This report is the final report for the project, "Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems." This project was conducted by Vista Research Corporation under contract no. DAAB07-89-C-P017 from the US Army Communications-Electronics Command.

The goal of the project was to develop an automated scenario generation system using the technology of artificial intelligence, specifically, knowledge-based (expert) systems. The system was named the "Scenarist." The Scenarist is a rule-based system that determines the location of subordinate units and equipment in a unit in accordance with rules that specify locations in accordance with tactical doctrine as a function of geographic (terrain) features, the mission of friendly forces, and the enemy threat. The system allows the user to specify the locations and objectives of military units on the battlefield, and then determines the locations of subordinate unit and equipment locations on the battlefield in accordance with the rules. After the user places a unit, the system displays an initial ("canonical") laydown of the unit's subordinate units and equipment, and then attempts to relocate the subordinate units and equipment in accordance with the rules. It informs the user whenever a subordinate unit or equipment is unsuitably placed, and specifies which rule of the knowledge base is violated. If

the user decides that certain rules are not reasonable, he may change them and relocate the units and equipment in accordance with the new rules. Or, the user may manually position certain items on the battlefield and request that their positions not be altered by the rules.

The system development was accomplished using the established methodology of systems engineering (requirements specification and analysis, top-level design, detailed design, implementation and test). The Scenarist incorporates the NASA CLIPS knowledge-based (expert) system, and is intended for use with digital mapping data. Mapping data used in the system development and test included digital mapping data extracted from samples provided with the US Army's GRASS geographic information system. A system test was conducted using rules derived from tactical doctrine for the TRAILBLAZER electronic warfare system. The system is programmed in the C programming language, and the current system prototype is implemented on an 80386 microcomputer.

This final report describes project activities, accomplishments, and deliverables. It also provides a description of the Scenarist system concept and design, and a discussion of the rationale underlying the system design features. The Scenarist has been implemented as a microcomputer program, which has been documented in a separate Programmer's Manual, Variable Glossary, and User's Manual. This final report includes background information about the Scenarist concepts and design that would be

helpful to someone using or modifying the system,
but are not part of the Programmer's Manual,
Variable Glossary, or User's Manual.

I. Introduction

A. Purpose of Report

This report is the final report for the project, "Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems." The purpose of this project was to develop an automated means of generating tactical military scenarios, using the technology of artificial intelligence, specifically, knowledge-based (expert) systems. This work was funded as a Phase II project under the Small Business Innovation Research (SBIR) program. The work was performed by Vista Research Corporation under contract number DAAB07-89-C-P017 from the US Army Communications-Electronics Command (CECOM). The automated scenario generation system developed under this project was named the "Scenarist."

The purpose of this final report is to describe the Scenarist development effort in terms of the project activities and products, and to provide a general description of the Scenarist. The Scenarist was implemented as a microcomputer program, and a User's Manual (Reference 16), Programmer's Manual (Reference 17), and Variable Glossary (Reference 18) were developed to document the scenarist software system, i.e., to describe its structure and tell how to use it. In addition to providing a general description of the Scenarist, this final report also presents design concepts and details that were not called for in the software documentation or any other required contract deliverable.

B. Summary Description of the Scenarist

The Scenarist Scenario Generation System is a workstation-based aid for use by military scientists to develop tactical military scenarios. As used in this report, a "tactical military scenario" is a specification of the initial positions, activities, and planned movements of tactical military units. The specification may be at various levels of detail, ranging from a specification of the positions of high-echelon tactical units (divisions, brigades, battalions) to the positions and activities of low-level units (e.g., companies) or individual platforms or items of equipment (e.g., a vehicle or radars).

The immediate application of the Scenarist is to develop laydowns of noncommunications equipment for use by the US Army Communications-Electronics Command (CECOM) in its analysis of electronic warfare systems and concepts. Later applications may include development of scenarios for use in a wide range of military applications, including force planning; research, development, test and evaluation; operational planning and evaluation; and training and education.

The Scenarist is a rule-based system that determines the location of subordinate units and equipment in a unit in accordance with rules that specify locations in accordance with tactical doctrine as a function of geographic (terrain) features, the mission of friendly forces, and the

enemy threat. The system allows the user to specify the locations and objectives of military units on the battlefield, and then determines the locations of subordinate unit and equipment locations on the battlefield in accordance with the rules. After the user places a unit, the system displays an initial ("canonical") laydown of the unit's subordinate units and equipment, and then attempts to relocate the subordinate units and equipment in accordance with the rules. It informs the user whenever a subordinate unit or equipment is unsuitably placed, and specifies which rule of the knowledge base is violated. If the user decides that certain rules are not reasonable, he may change them and relocate the units and equipment in accordance with the new rules.

The system development was accomplished using the established methodology of systems engineering (requirements specification and analysis, top-level design, detailed design, implementation and test). The Scenarist incorporates the NASA CLIPS knowledge-based (expert) system, and is intended for use with digital mapping data. Mapping data used in the system development and test included digital mapping data extracted from samples provided with the US Army's GRASS geographic information system. A system test was conducted using rules derived from tactical doctrine for the TRAILBLAZER electronic warfare system.

The system is programmed in the C programming language, and the current system prototype is implemented on an 80386 microcomputer having a

VGA or EGA color monitor. The system displays units, subunits, and equipment on maps that depict various geographic features, such as elevation, terrain type, roads, rivers, bodies of water, and urban areas. The user inputs data to the system either by means of preprepared data files or by means of the keyboard and a mouse. The user interface makes heavy use of windows and menus. The screen display can be output to the printer. The locations of the units and equipment can be output to the printer or to a text file.

The system accepts maps of varying resolutions. The system accepts a terrain-type map (represented digitally in a "map file"), an elevation map, and a road map. The terrain-type and elevation maps are required, but the road map is optional. The various maps need not be of the same resolution. In a typical application, the system is first used to position items on a low-resolution map, and is then used to reposition the items on a succession of higher-resolution maps.

The Scenarist system design is heavily object oriented. Map features and military units and equipment are "objects," whose attributes and interrelationships are factors in terms of which the placement rules may be formulated. The Scenarist utilizes a parametric representation of military units which allows the user to specify a considerable amount of data about a unit, while at the same time allows for rapid repositioning of the unit's subordinate units and equipment whenever the unit is moved.

C. Summary Description of the Scenarist Development Project

1. Project Motivation and Goals

CECOM funded the development of the Scenarist in the hope of producing an automated scenario generation system that would reduce the time and cost of generating scenarios, and make the task of validating the scenarios less difficult. Prior to undertaking the development of the Scenarist, CECOM funded a study to assess the feasibility of using the technology of knowledge-based systems to generate scenarios. That feasibility study was funded as a Phase I Small Business Innovation Research (SBIR) study (Reference 1). The Phase I study demonstrated the feasibility of this approach, and, based on this assessment, a Phase II proposal was solicited (References 2, 3) and a Phase II effort -- the current project -- was funded.

CECOM uses military scenarios as input to computerized models that evaluate the performance of electronic warfare system concepts. Computerized evaluation models are used to evaluate concepts before making costly decisions to build new systems. In order to evaluate electronic warfare concepts and systems under realistic conditions, it is necessary to specify realistic military scenarios in which the systems will operate. For example, the positions of a number of radars may be specified on a battlefield, and an airborne vehicle flown over the battlefield area (in the model). The

performance of a hypothetical radar warning receiver or search methodology may be assessed by observing how many radars are detected.

Previously, manual methods have been used to generate scenarios; That is, experienced military personnel would synthesize the scenarios. There are several problems associated with the manual approach. First, it is costly and time consuming. The generation of a realistic scenario can consume many weeks or months of skilled personnel. Second, it is difficult to demonstrate the validity of the generated scenarios. The validity of the scenario rests primarily on the reputations of the scenario builders, and careful review by other experts. Third, it is not easy to make changes or variations in a scenario; if modified, the scenario must be revalidated by additional expert review. Fourth, the scope of inference for analyses based on manually generated scenarios is not clearly defined.

2. Project Objectives

The primary objective of the Scenarist development project was to develop an automated scenario generation system that could place division-level units on a battlefield in accordance with tactical doctrine, taking into account terrain features, friendly mission, and enemy threat. It was a project requirement that the system use the methodology of knowledge-based (expert) systems, and be able to accept digital terrain data. It was also a project objective that CECOM would be able to run

the developed system on computer equipment available in the SIGINT/ESM Branch, and that the model would be capable of being used to assist the generation of scenarios for system evaluation models used by CECOM.

The equipment and models used by CECOM changed during the course of the project. In order to make efficient use of project resources, it was not feasible to change the Scenarist design dynamically in response to these changes. For this reason, it was decided early in the project that the Scenarist would be designed for use on an 80386 microcomputer, and that the system would be capable of producing an ASCII output file containing the identifications of units and equipment and their locations. CECOM would accept responsibility for reformatting the output file for use in specific applications.

In addition to the goal of developing an automated scenario generation system that would be of use to CECOM, it was desired that the developed system could be adapted for use by other military organizations. This goal derives from the mission of the SBIR program, which is oriented toward the funding of research that will result in research products that are of value not only to the funding organization, but to other organizations as well.

To achieve the goal of developing a system that could be of value to organizations in addition to CECOM, Vista identified system characteristics that would promote achievement of this goal. During the course of the system

development effort, system requirements were developed that corresponded both to CECOM's objectives for the system and the SBIR program objective of developing a system of general utility.

In this project, the term "scenario" is defined to mean a description of the positions of military units and equipment at a point in time, together with optional additional information about the statuses, activities, intended movements, missions and objectives of those units. Other terms for the specification of the positions of the units and equipment are "laydown," "deployment," or "simulated tactical deployment."

3. Project Activities

The activities of the Scenarist development project centered on eight project tasks, which implemented a classical systems engineering approach to the development of the automated scenario generation system. These tasks were the following:

1. Project Planning
2. Requirements Analysis
3. Top-Level Design
4. Detailed Design
5. Implementation and Test
6. Develop Interface with CECOM Systems
7. Demonstration Cases
8. Final Report

The content of each of these tasks will be described in Chapter II of this report.

4. Project Accomplishments, Products and Deliverables

The project efforts resulted in the successful development and demonstration of an automated scenario generation system that satisfied all of the project objectives. The Scenarist system is a knowledge-based system that places subordinate units and equipment on a battlefield in accordance with rules that take into account terrain features, friendly mission, and enemy threat. It runs on an 80386-based microcomputer, and produces an output file containing the descriptions and locations of units and equipment.

The main product of the research effort is the Scenarist software, which is distributed on a number of 3-1/2" high-density diskettes (and also on 5-1/4" double-density diskettes), a User's Manual, a Programmer's Manual, and a Variable Glossary. The User's Manual describes procedures for installing and running the system. The Programmer's Manual describes the software in a way that will facilitate future modifications and extensions of the system. The Programmer's Manual incorporates modern procedures for software documentation, including structure charts, data flow diagrams, and a complete program listing. The Variable Glossary defines all of the functions and variables of the Scenarist programs.

In addition to the software, the project produced a number of other products, including over a dozen reports, briefing summaries, and demonstration packages. These products will be described in Chapter II.

D. Report Organization

This final report contains eleven chapters and two appendices. Chapter II describes the project activities, accomplishments, products and deliverables. Chapter III presents the Scenarist system concept. Chapter IV presents the system requirements, both those that were developed at the beginning of the project and those that evolved during its course. Chapter V presents the Scenarist system top-level design. Chapter VI presents the top-level design for the "development" software, i.e., the software that was developed in the project. Chapter VII presents the detailed design for the development software. Chapter VIII describes the Scenarist system documentation. Chapter IX discusses efforts taken to verify and validate the system. Chapter X summarizes the Scenarist system capabilities and limitations. Chapter XI describes areas for future development of the Scenarist system. Appendix A contains the test report that describes the results of the Scenarist test. This test report is included as an appendix as an illustration of the application of the Scenarist to the placement of real units on real terrain. Appendix B contains the output from a demonstration of the Scenarist.

II. Project Task Descriptions

This chapter describes the project activities, accomplishments, products, and deliverables. During the project, the project officer was provided with monthly progress reports that described project activity and progress during the preceding month, and with quarterly progress reports that described project activities and progress during the preceding quarter. This chapter summarizes the highlights of those project reports. The description of the project activities, accomplishments, products, and deliverables is presented in terms of the tasks of the project work plan.

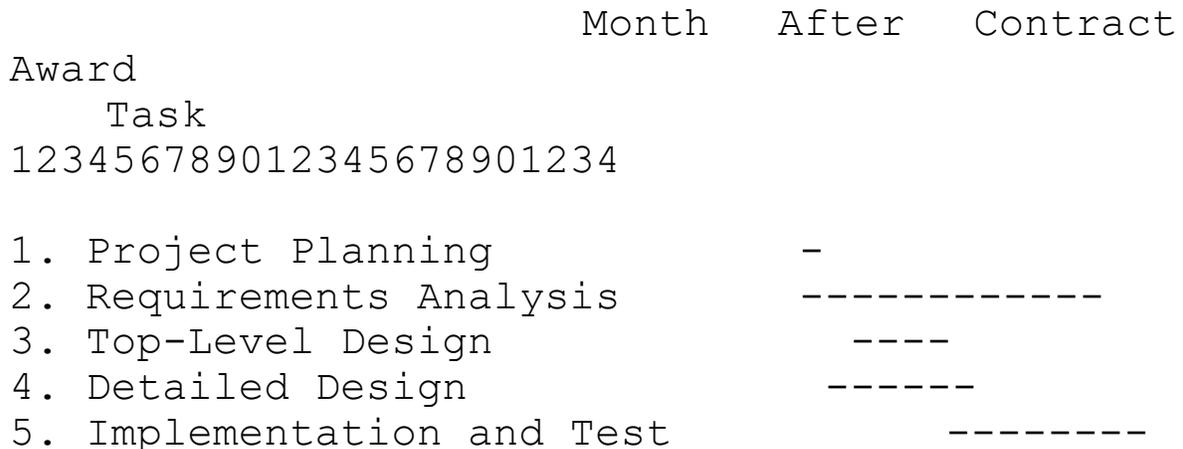
A project work plan, comprised of a list of tasks and schedule for their accomplishment, was developed at the beginning of the project. During the first year of the project, minor revisions were made to the project work plan, and the schedule was revised. Figure 1 presents the revised project schedule, which was adhered to for the duration of the project. Figure 1 also includes a list of the deliverables that were planned and produced during the course of the project.

The tasks listed in Figure 1 encompassed all of the project's activities. The project tasks were identified and defined to implement a systems engineering approach to the Scenarist development. The following paragraphs describe briefly the content of each of the eight project tasks.

1. Project Planning

Task 1, Project Planning, was concerned with the development of a work plan for the project, and with the revision of the work plan in response to project developments. The major activities in Task 1 were the development of the initial project work plan and the development of a revised work plan after eight months had passed. The revision to the work plan eliminated three tasks in the original work plan that were concerned with the selection of a tactical combat model for which the Scenarist would be used to produce Scenarios, and with the development of procedures for producing for that tactical combat model. These three tasks had been proposed in the original proposal, but were of no interest to CECOM. CECOM's interest was in assuring that the Scenarist design would be able to produce scenario input to certain CECOM models. For this reason, the three proposed tasks were replaced by a single task, "Develop Interface with CECOM Systems."

Figure 1. Revised Project Schedule (May 1, 1990)



6. Develop Interface with
CECOM Systems

7. Demonstration Cases --
8. Final Report
--

Deliverables

123456789012345678901234

- | | | | | | | | |
|--|---|---|---|---|---|---|---|
| 1. Quarterly Status Report | x | x | x | x | x | x | |
| x | x | | | | | | |
| 2. Prelim. Soft. Req. Spec. | | | x | | | | |
| 3. Software Requirements Spec. | | | | | | x | |
| 4. Software Des Doc (Top Level) | | | | | x | | |
| 5. Software Des Doc (Detailed, Draft) | | | | | x | | |
| 6. Software Design Document (Final) | | | | | x | | |
| 7. Software Test Description | | | | | | | x |
| 8. Outline of Soft. User's & Prog. Manuals | | | | | | | x |
| 9. Software User's Manual | | | | | | | x |
| 10. Software Programmer's Manual | | | | | | | |
| x | | | | | | | |
| 11. Presentation Material | x | | | | x | | x |
| x | | | | | | | |
| 12. Final Report | | | | | | | |
| x | | | | | | | |

Trips:

123456789012345678901234

- | | | | | | | | |
|---------------------------|---|--|--|--|--|---|---|
| 1. Presentations to CECOM | x | | | | | x | x |
| x | | | | | | | |

During the course of the project some changes were made in the project funding profile, and some changes were made in the rates at which labor effort was expended, from what was originally proposed. Those changes had no effect, however,

on the overall project schedule at the level of detail presented in Figure 1.

Figure 1 presents a list of the deliverables proposed to be delivered to CECOM. Under the terms of the contract, only two kinds of deliverables were required: eight quarterly progress reports and a final report. The additional deliverables listed in Figure 1 were included for two reasons: (1) to provide CECOM with information on project progress on a regular basis; and (2) to provide software development documentation in compliance with the DOD-STD-2167A standard, "Defense System Software Development." CECOM had not required compliance with the 2167A software development standard, but use of it had been proposed by Vista.

The contract called for the submission of quarterly project status reports, which were produced and submitted as scheduled (Reference 4). In addition, it was agreed between the Vista project director and the Government project officer that it would be desirable to summarize project status on a more frequent basis than quarterly, and Vista submitted letter-style monthly progress reports throughout the project term (Reference 4). The Government project officer was also informed of progress through a number of telephone discussions and three visits to CECOM.

During the detailed design phase of the project, it became clear that production of a software detailed design document would consume an

inordinate amount of resources. Moreover, a "rapid prototyping" approach to the system development had been proposed. Under this approach, the system development was implemented by developing a series of progressively more advanced system versions (prototypes). After a prototype was developed, its performance was reviewed. Based on this review, modifications could be made to the original system requirements or design, prior to the development of the succeeding prototype. Since resources were not sufficient to develop a series of models and to produce a useful software detailed design document, it was agreed between Vista and the CECOM project officer to drop the production of the software detailed design document, as long as the software would be adequately described in the Programmer's Manual.

With respect to project trips to CECOM, three trips were originally proposed. In the May schedule revision, four trips were planned. In all, only three trips were made, per the original proposal.

The "presentation material" referred to in Figure 1 includes materials provided at briefings.

In addition to the deliverables listed in Figure 1, a number of other products were produced. These included: a Software Development Plan (Reference 5); a System Design (System/Segment Design Document) (SSDD, Reference 6); three Scenarist demonstration packages, which were distributed to interested parties during the

course of the project; and a Test Report that described the results of the system software test.

A complete list of all of the project deliverables is included in the list of references at the end of this report. (The list of references includes only the final versions of the DOD-STD-2167A reports, and does not include the Preliminary Software Requirements Specifications document, which was replaced by the Software Requirements Specifications document.)

2. Requirements Analysis

A classical systems engineering approach was adopted for the Scenarist system development. This approach consists of the following major steps: (1) requirements analysis (including requirements specification); (2) top-level design; (3) detailed design; (4) implementation and test. The first step in this approach is requirements analysis. In this step, we reviewed CECOM's and our own objectives for the system, and derived a number of system requirements from those objectives. In addition, we also identified and reviewed the needs of other potential users of the system, and identified additional requirements for those potential users.

As mentioned, a rapid prototyping approach was adopted for the system development. As part of this approach, it is permissible to reexamine the reasonableness of the initial system

requirements, and to modify them if, in the light of information gained from experience with a prototype, such modification appears to be in the best interest of the project's objectives.

The original system requirements were presented in the System/Segment Design Document, the Software Requirements Specification (Reference 7), and the Test Plan (Reference 12). The final system requirements are presented in the Test Report (Reference 15), which is included as Appendix A of this final report. (Note: the Software Test Description of Figure 1 was renamed as the Test Plan.)

3. Top-Level Design

The identified requirements were used as a basis for synthesizing a number of alternative top-level designs for the Scenarist. These alternatives included consideration of alternative methodological approaches, programming languages, user interfaces, expert systems, geographic information systems, data sources, and computers for use in the Scenarist system. The relative advantages and disadvantages of the various alternatives was considered in terms of the extent to which requirements were satisfied and in terms of cost and the resources available to the project.

The various design alternatives considered, and the rationale for selecting a preferred alternative, are presented in the System/Segment Design Document, and the Software Design Document (Reference 8).

4. Detailed Design

The Scenarist detailed design evolved during the project's rapid prototyping process. The final software design is described in the Programmer's Manual. In addition, Chapter V of this final report includes some additional information on the detailed system design, that is not included in either the System/Segment Design Document (which document was produced prior to the start of the rapid prototyping process) or the Programmer's Manual (which document specifically addresses software documentation, not overall system design).

5. Implementation and Test

The term "implementation" refers to programming (coding) the system. As discussed previously, a rapid prototyping approach to system development was used, in which a succession of system versions was developed (i.e., designed and programmed). The three Scenarist demonstration models (References 9, 10, 13, 14) that were produced during the course of the project (and delivered to CECOM) illustrate model versions that were available at three points in time. These demos served as a basis for discussion for model changes to be made in the next prototype.

With respect to system testing, a Test Plan describing the planned system test was produced and submitted to CECOM. The test was then conducted in accordance with the Test Plan, and

the results were presented in a Test Report (included as Appendix A to this report).

The products of the Implementation and Test task were the Test Plan, the Test Report, the User's Manual, the Programmer's Manual, and the system software (on microcomputer diskettes). In addition, program listings were prepared and submitted to CECOM on a monthly basis (Reference 11). The final program listing is Volume 2 of the Programmer's Manual.

Near the end of the project, a Variable Glossary was prepared that contained definitions of all functions and variables used in the Scenarist programs.

6. Develop Interface with CECOM Systems

The purpose of this task was to determine what features the Scenarist should possess so that it would be most useful to CECOM in generating scenario input to its electronic warfare systems evaluation models. During the course of the project, discussions were held with the CECOM project officer and other CECOM personnel to determine what system features were required in order for the Scenarist to be of maximum value to CECOM. As mentioned earlier, the models with which CECOM desired to use the Scenarist output changed over the course of the project, and for this reason it was decided simply to output a laydown in the form of an ASCII file containing unit/equipment identifications and locations.

Another issue that arose in this task was the issue of what source should be used for digital terrain data. In the early phases of the project, it was hoped that the Scenarist would be able to access Defense Mapping Agency digital terrain data (Digital Terrain Elevation Data, or DTED, and Digital Feature Analysis Data, or DFAD). The DTED were used by CECOM. A problem that arose was that Vista did not possess the computer equipment and tape drives required to process the tapes on which CECOM's DMA DTED data were stored. Contact was made with DMA to obtain DTED on diskettes or compact laser disks usable by an 80386 microcomputer, but it was not possible (because of limited production capacity for these media and because of the Gulf War) to obtain sample data in time for use by this project. Because of these difficulties, the project officer agreed that the objectives of the project would be satisfied if sample digital terrain data available as part of the US Army's GRASS geographic information system (which Vista possessed) were used for the system test.

7. Demonstration Cases

Two types of demonstrations were produced during the course of the project. First, during the course of the rapid prototyping development process, three demonstration packages were produced and sent to the CECOM project officer and various other interested parties. Second, the system test constituted a demonstration of the application of the Scenarist to generate a laydown of a real system (the TRAILBLAZER system) on real terrain (the GRASS sample terrain, near

Spearfish, SD). As mentioned, a description of the system test is included in Appendix A.

8. Final Report

The content of this final report was described earlier. As mentioned above, in addition to describing the project activities and accomplishments, this report also presents some detailed design information not available in other project documents.

In particular this report presents a discussion of the rationale for the top-level and detailed system and software system designs. The Programmer's Manual presents data flow diagrams for all major system modules and a complete listing of all program code. Furthermore, the Variable Glossary presents definitions of all software system functions and variables. These documents do not, however, present a discussion of why the model variables, data structures, and functions are defined the way they are, from the viewpoint of satisfying the system requirements and achieving the project objectives.

The User's Manual concentrates on getting a user started with the system, but it does not provide discussion about the theoretical concepts that help guide choices such as how to decide on what "geographic types" to use for subordinate units and what map resolutions are appropriate. It addresses mainly the identification of the functions provided by the program. A very important aspect of using the system is, however, the making of "modeling" decisions in the course

of the problem setup. The Scenarist model is complex, complicated, and powerful. Although the user interface is relatively user-friendly, it offers the user a lot of choices. How to use the system capabilities is assisted greatly by an understanding of how the system is designed and how it works. This final report includes this information.

III. Scenarist System Concept

A. Background on Scenario Generation

1. The Need for Military Scenarios

In order to evaluate the performance or effectiveness of a military system, it is desirable to conduct the evaluation in a specific military context. A description of the initial positions, activities and intended movements of military forces is referred to as a "military scenario," or, simply, a "scenario." In order to evaluate military systems in a battlefield context, it is necessary to have a capability to develop scenarios.

The term "scenario" is often used in a general context to refer to the political, economic and social events leading up to conflict. As the term is used in this report, the term does not include a description of these events leading up to the conflict -- it refers only to the configuration (positions, activities, and intended movements) of units and equipment at a specific point in time, with no reference to the

events leading to the configuration. Furthermore, while the term may be used to refer to a description of the initial positions, activities and intended movements of high-echelon units, we will also use it to refer to a very detailed description of the positions and activities of low-level units and individual equipments, i.e., to a detailed tactical deployment. The specific meaning that is intended for the term will be clear from context.

To be precise, in this report we shall use the term "scenario" to refer to a specification of the initial conditions of force elements (military items, including units, platforms or equipment) prior to a battle, plus a specification of the mission and objectives (or targets) of the units. The term "initial conditions" refers to the number and characteristics (locations, activities) of the force elements. An "objective" is any entity on the battlefield that is related to a force element by a "mission." An objective may be another force element, a geographic feature (e.g., a location, line, boundary, road, river, body of water, area), or an item of intrinsic value (e.g., population, agricultural area, industrial target, political territory). A "mission" is a process or action, such as "proceed to," "occupy," "observe," "jam," "neutralize," or "destroy."

The definition of the terms used in the preceding paragraph is deliberately vague. In the Scenarist, the specific definitions of force elements, objectives, and missions is left to the

user. These entities and processes, once defined, may be referred to in the rules used to determine the placement of the force elements.

To summarize, the term "scenario" as used in Scenarist applications is a specification of the initial positions, activities and intended movements of military units, and a specification of their missions relating to objectives. The positions of the units may also be referred to as a "tactical deployment," or a "laydown." In general, the term "scenario" is often used to refer to a description of the positions, activities, and planned movements of high-echelon units, and the terms "deployment" and "laydown" to a detailed description of all of the force elements or other equipment associated with the high-level scenario description.

There are many different types of systems that must be evaluated, and many different contexts in which they are to be evaluated. For these reasons, there is a corresponding wide variety of scenarios that are developed and used by military analysis organizations. This wide variety of applications of scenarios is the primary reason why the term has a wide range of meaning.

The level of detail of a scenario depends on the application. In general, the positions and activities of the major units are always specified. What details are specified depends on the application. For example, in a CECOM application, it may be desired to assess the

ability of an aircraft to detect radars. In this application, the scenario must specify the locations, duty cycles, powers, frequencies and modulation characteristics of the radars in a division. In an application at the US Army Electronic Proving Ground's (USAEPG's) Electromagnetic Environmental Test Facility (EMETF), it may be desired to determine the probability of successful communication over communications links in a corps-level deployment. In this situation, it may be necessary to specify the locations, duty cycles, and frequencies of hundreds of thousands of communications emitters. In an application at the US Army Intelligence Center and School (USAICS), it may be desired to assess the ability of intelligence analysts to collect intelligence data from a variety of data collection sensors. In this application, it is desired to specify the locations and activities of these sensors and the items that stimulate the sensors. In a tactical training exercise utilizing a tactical combat model, it may be desired to specify the locations of direct-fire equipment and targets.

2. Applications that Require Scenarios

The preceding paragraphs have considered the role of scenarios in test and evaluation applications. The test and evaluation application area has been emphasized because it is of particular interest to CECOM and because it is one of the most important application areas for scenarios, from the viewpoint of visibility. While scenarios play an important role in test and evaluation applications, this area is not the

only one in which scenarios are important. In general, there are a number of areas of military science in which scenarios are used. These use areas include force planning; research, development, test and evaluation; operational planning and evaluation; and training and education. These areas are defined as follows:

- o Force planning includes force structure analysis (determination of the organizational form and types of equipment associated with land, sea, and air combat units of a specified size, such as a battalion, carrier task group, or tactical air wing); force level analysis (determination of the number of combat units of the same type to include in a larger force assigned to a military purpose); and force mix analysis (determination of the mix of land, sea, and air units that constitute a force assigned to a military purpose).

- o Research, development, test and evaluation is concerned with concept development, system design and development, system test and evaluation (development test and evaluation and operational test and evaluation).

- o Operational planning and evaluation includes analysis of operational concepts, tactics and doctrine.

- o Training and education includes training of military commanders under realistic conditions and education of military science

students in warfare concepts and analysis and gaming.

For large-scale evaluations of systems in a combined-arms context, the principal means of developing scenarios is manual. The US Army Training and Doctrine Command (TRADOC) Analysis Center (TRAC) located at Fort Leavenworth works with 21 Army proponent agencies to develop what is called a Standard Scenario (previously a "SCORES" scenario). The process is costly and time consuming: up to two years may be required to develop a Standard Scenario.

3. Shortcomings of Existing Scenario-Generation Methodology

From a test and evaluation viewpoint, the existing methodology for generating scenarios has several shortcomings. These include the following:

- o Expensive and Time-Consuming. The manual process for generating scenarios is costly and time consuming.

- o Limited Scope. Because of the time and effort involved, both to generate the scenario and to develop a detailed tactical deployment corresponding to it, few scenarios are implemented as tactical deployments for input to tactical combat or other military system evaluation models. The result is that the evaluative scope of a system evaluation based on a single scenario is restricted. It becomes difficult to

estimate what the system performance will be in a wide range of circumstances.

o Ill-Defined Basis for Inference. The typical process for developing a scenario is to use human experts (military analysts, tacticians). A problem that arises with this approach is that, because no well-defined mathematical or probabilistic framework underlies the scenario development, the evaluation results obtained using such scenarios are conditional on the scenario actually used, and the methods of mathematics or statistics cannot be used to extrapolate the results to a broader population of scenarios. In general, there is no general-purpose automated methodology currently available for generating probability samples of scenarios (or tactical deployments).

o Lack of Reproducibility. With manual methods, two different teams of experts will likely generate different scenarios, even given the same requirements for the scenario. Such variation may be similar to that occurring in real battlefield deployments, and may even be advantageous in some applications, especially training. The presence of this type of uncontrolled variation may be problematic in test and evaluation applications, however, where systematic procedures, controlled variation, and reproducibility may be important.

o Difficult to Validate. Manually generated scenarios are difficult to validate. Perhaps the best validation of a manually generated scenario is the reputations of the experts who constructed the scenario. The difficulty in validation stems from the fact that it is difficult to validate the scenario ex post, that is, by looking at the scenario after the fact. Ex post validation depends largely on verification that all of the many interrelationships among the entities of the scenario are reasonable. (Validation is not easy even for model-generated scenarios, in which case the model must be validated. One of the advantages of a rule-based system for generating scenarios is that the model validation is somewhat easier for this type of model than some others (such as simulation); the reason for this is that the model possesses a high degree of face validity, since a rule-based system is simply a mechanism for building a scenario directly from all of the known rules about deployments.)

o Nonparametric and Non-Model-Based Approaches. A scenario (or tactical deployment) is often specified in detail (by specifying the location and activity of every force element (unit or equipment)) in the absence of a mathematical model of tactics or force deployment. Borrowing from statistical terminology, this type of description may be referred to as a "nonparametric" representation. A problem with nonparametric representations is that

they are often awkward to work with. The nonparametric specification is not an efficient, model-based framework for describing the scenario or the scenario generation process (in terms of a manageable number of entities, variables, relationships, or procedures). Without a model-based framework, efficient, convenient, easy-to-use means for rapidly generating, modifying, or even describing scenarios are generally not available.

It was in recognition of the drawbacks of the existing methodology for scenario generation -- costly, time consuming, limited scope, ill-defined basis for inference, difficulty in validation, nonparametric representation -- that we proposed to attempt the development of automated means for scenario generation.

4. Alternative Scenario Representation Schemes

We shall now define more specifically what we mean by "parametric" and "nonparametric" scenario representations. These concepts are very important with respect to the conceptual framework of the Scenarist, and the discussion that follows is hence somewhat detailed. A nonparametric representation (description, specification) differs from a parametric one in that in a nonparametric description the number of variables (parameters) describing the scenario increases without bound as the resolution of the objects defining the scenario (a fixed number of maps and a fixed number of units) increases; with a parametric description,

the scenario is described in terms of a relatively small number of variables (parameters) whose number does not increase without bound as the resolution of the objects defining the scenario increases. The "resolution" of the objects is the amount of detail available to define them. In other words, no matter how much detailed data is available about maps or units of the scenario, a parametric scenario description has a fixed number of variables describing it. Of course, the values of these descriptors may fluctuate somewhat as the level of detail increases.

An example borrowed from the field of statistics will be used to clarify the distinction between parametric and nonparametric representations. A (one-dimensional stationary Gaussian) stochastic process may be represented (completely specified, or "characterized") in three different ways -- in terms of a complete definition of a joint probability distribution of the random variables x_t, x_{t+1}, \dots representing the process, or in terms of a parametric model defining the probabilistic relationship of the value of the random variable x_t at one point in time (t) to the values of the variable at other points in time, or in terms of the mean, variance, and autocorrelation function of the variable. For example, a two-parameter autoregressive/moving average process with one moving average parameter and one autoregressive parameter is completely specified by the values of these two parameters and the mean and variance of the model "error" terms: $x_t - cx_{t-1} = a_t - da_{t-1}$, where x_t denotes the value of the process variable

at time t and a_t denotes the value of the model error term at time t (the a_t 's are a "white noise" process -- a sequence of uncorrelated normally-distributed random variables having zero mean and constant variance). The constants c and d (and the variance of the a_t 's) are the parameters of the model. Equivalently, this stochastic process may be completely specified by the mean, variance, and autocorrelation function of the process. The autocorrelation function is a function that specifies the correlation between the random variable x_t and x_{t-k} as a function of the "lag" k (the difference in the time index of the two variables).

The major distinction between the parametric-model representation of the process and the autocorrelation representation is that the parametric-model is specified by three numbers (the values of c , d , and the variance of a_t), whereas the autocorrelation representation is specified by the value of the variance and the entire autocorrelation function -- an infinite sequence of numbers! While both representations have their uses, the parametric-model representation is much more convenient for identification or summary-descriptive purposes.

The concepts of parametric and nonparametric representations arise both with respect to the theoretical-model description of a process (as described above) and sample data. Continuing the preceding example, suppose that a sample of n time series observations is available for the process. The sample data may be summarized in

terms of three parameter estimates (for c , d , and the variance of a_t), or in terms of the variance and a sample estimate of the autocorrelation function. While the sample autocorrelation estimate is a useful guide in suggesting the parametric structure of a stochastic process for which the parametric model is unknown, it is a very inefficient means of summarizing the sample data -- the number of terms ("parameters") of the autocorrelation function increases without bound as the sample size increases.

With respect to scenario description, the concept of a parametric representation is operationalized as follows. Every unit is defined in terms of a fixed number of subordinate units. The characteristics of the unit are defined in terms of a fixed number of variables, such as the unit location and the types and locations of the subordinate units. As many units may be included in the scenario as desired, but the number of variables (parameters) defining each one is fixed. No matter how much data are provided about the units or the terrain on which the units are located, the number of parameters defining each unit in the Scenarist system is the same. With respect to the maps, the situation is as follows. No matter what the resolution of the map, the same number of factors affecting the placement of units are derived from the map data. (These factors are the quantities used in rules, such as whether a line-of-sight condition exists between two units.)

Note that a digital map may be represented either parametrically or nonparametrically, and that

the nature of a map specification does not affect whether a scenario representation is parametric or nonparametric. A cellular-map representation is a nonparametric map representation (since the number of values defining the map, i.e., the cell values) increases without bound as the map resolution increases without bound. A vector-map representation may be a parametric representation, if it is defined in such a way that the number of objects (or other variables) defining or characterizing the map does not increase without bound as the map resolution increases without bound. In classifying a scenario as parametric or nonparametric, we are not concerned about whether the maps used in conjunction with the scenario are parametric or nonparametric representations. In the classification of a scenario as parametric or nonparametric, we are concerned only with the number of factors whose values are derived from maps, not with the parametric or nonparametric nature of the maps themselves.

A practical advantage of using nonparametric scenario representations is that the scenario may be developed and described in the absence of an analytical (mathematical, statistical, model-based) theory of combat. This feature may be considered an advantage because tactical combat is a very complex temporal/spatial/stochastic process, and the problem of developing mathematical models of tactical combat is formidable. Moreover, in spite of the lack of a theoretical conceptual framework (model) for the scenario or

scenario-generation process, the end result of the process -- a detailed scenario description -- has the same general appearance as for a scenario-generation process based on a well-founded mathematical model of combat. The high level of detail of the final product (the detailed scenario description) is often represented as affording the scenario-generation process a high degree of "face validity." ("Face validity" refers to the degree of correspondence between entities and processes of a model and the corresponding entities and processes of the real world.)

A major problem with nonparametric representations is that they are difficult to work with. Changes to a scenario or tactical deployment are made directly, by making specific changes to the locations or activities of individual units or equipments. Unlike a "parametric," or model-based approach, it is not possible simply to change the value of a parameter, and have the scenario appropriately modified. There is not currently available a well-defined, accepted taxonomic framework for describing the characteristics (features) of scenarios, or analytical (parametric) methodology for developing scenarios having prescribed values of taxonomic descriptors.

In summary, a nonparametric representation of a scenario may be highly detailed and lend a high level of face validity to the scenario-generation process. It is, however, an awkward representation that is difficult to work with (i.e., it requires a great deal of time

and effort to construct a scenario and to make changes to an existing scenario).

It is noted that, while a parametric (model-based) framework is a convenient basis for describing and generating scenarios, it is nevertheless the full-blown, high-detail, complete specification of the scenario that is needed as the basis for accomplishing a specific evaluation or training application. It is sometimes necessary or desirable to distinguish between the complete, detailed description of a scenario (i.e., of every unit or item of equipment of interest), and a summary description of the scenario, i.e., a description of the scenario in terms of a small number of model parameters. We shall refer to the summary description of a scenario as such, or as a "parametric description." We may refer to the complete scenario description as a "complete scenario," "complete scenario description," "detailed scenario description," "generated scenario," "fleshed-out scenario," "sample scenario," "scenario realization," or "scenario instantiation."

Regardless of what approach is used to generate a scenario (manual, automated, model-based), the final product of the scenario-generation process is the same. The amount of effort required to generate a scenario, however, may vary dramatically. Using a manual process, the scenario-generation process may require months or even years of effort, by a large team of experts. For a scenario-generation process based on a mathematical model, however, the

scenario-generation process is much easier. For example, suppose that the model is a probability model. In this case, the scenario is obtained simply by selecting a particular sample from the probabilistic scenario specification. This sample contains as much detail as is needed by the application; i.e., the sampling units may be major units, minor units or individual items of equipment. This sample is used for the application.

A major problem with using a non-model-based process for scenario generation is that if no analytical framework (such as statistical or logical inference or a parameterized algorithm) is available in which to conceive abstractions of scenarios, then there is in general no convenient or efficient means of producing scenarios possessing specified properties, or for making even minor changes to a scenario (such as moving a unit by a small amount). Furthermore, as noted above, the task of validating the scenario generation process may be difficult. It is in general not possible to distinguish how a scenario was generated, given the complete scenario description. Two scenarios -- one generated manually and the other by a parametric probability model -- have the same general appearance. In general, the terms "parametric" and "nonparametric" refer either to a description of the scenario or to the nature of the process that generated it (just as these terms could be used to refer to representations of either a theoretical stochastic process or a sample realization of a process). Regardless of whether the scenario-generation process is based

on a parametric-model representation or a nonparametric representation of scenarios, the complete scenario has the same general appearance, i.e., the specification of unit or equipment locations and activities.

The situation is analogous to the problem of deciding whether a set of numbers could have been generated by a random process, such as a uniform distribution, normal distribution, or some other specified probability distribution. The property of randomness is an attribute of the process that generates the numbers, and not of the numbers themselves. If a process used to generate the numbers is known, there is little difficulty in assessing its randomness, and in generating samples of random numbers using the process. If, however, only a list of numbers is available, it is not possible to determine whether they were generated by a specific random process; at best, the numbers may be used as a basis for conducting various goodness-of-fit tests to assess the likelihood of their having been generated by a specified random process.

The Scenarist is based on parametric representations of military units and geographic features. The structure of each military unit is defined in terms of a fixed number of unit parameters, and the rules for positioning items are specified in terms of a fixed number of factors derived from maps and unit locations. No matter how much detail is available about units or maps, the Scenarist rule processing is based only on a fixed set of unit parameter values, a fixed set of terrain features that are

estimated from maps, and a fixed set of spatial relationships among units or items within units. (Note: Although the model parameters are fixed in a particular "run" of the system, they may, of course, be changed. For example, a user may wish to consider an additional factor in a rule. The system was designed with the capability to accept changes in the model specification.)

B. Selection of Rule-Based System

It was the goal of the Scenarist development project to develop an automated scenario generation capability that, on the one hand, could provide scenarios of use to the project sponsor (CECOM), and, on the other hand, could also provide scenarios to a variety of other applications. This goal would be achieved if a system could be developed that could determine the initial positions, activities, and intended movements of major units, and could also determine the location of subordinate units and equipments associated with these major units.

What we conceived for the present contract effort was a system that would position units on the battlefield in a hierarchical fashion, and also position particular items of equipment associated with those units. At the end of this two-year development effort, we planned to have available a system that could position military units and the noncommunications equipment associated with those units. In addition, we intended that the system would be adaptable (in a future effort) to position any other type of equipment, such as communications emitters or

intelligence sensors or direct-fire weapons, given the positions of the units.

By the term "hierarchical," we mean that, given the location of a high-echelon unit such as a corps, the system would position the next-lower-ranking echelon units, in this case the divisions. Then, having placed the divisions, the system would place the next-lower echelon units, e.g., brigades. This process of positioning units would continue to as low a level (e.g., company, platoon) as desired, or as required to form a sound basis for positioning equipment associated with the units. At each stage of the positioning process, the system would take into account the environment (i.e., terrain elevation and other geographic features), friendly mission, and enemy threat, in accordance with tactical doctrine.

The contract Statement of Work called for the development of an automated scenario-generation system using artificial intelligence technology, specifically, knowledge-based (expert) systems. The system that was to be available at the end of the two-year project was to contain rules for placement of units and noncommunications emitters associated with those units. Future development efforts would produce systems that contain rules for placement of other types of equipment (e.g., communications emitters, intelligence sensors).

It is noted that there are a number of useful conceptual frameworks in which scenario generation may be cast. Procedures could be

developed to generate scenarios using the methodologies of statistical sampling, optimization, game theory, simulation, knowledge-based systems, or combinations of these and other approaches. These alternative approaches have different advantages and disadvantages. While the current contract required the use of the technology of knowledge-based systems as a basis for generating scenarios, it was originally planned that the completed system would include statistical aspects as well. Because the development of the system turned out to be much more ambitious than originally realized, however, it was not possible to include statistical aspects in the completed system (e.g., through rules that require or allow location sampling, or through the generation of a population of scenarios from which probability samples of scenarios may be selected). The completed system does not explicitly address optimization (i.e., determination of a laydown that maximizes or minimizes a prescribed objective function), but rules could be developed to address this aspect.

The principal advantages of using a rule-based system for scenario generation appear to be two. First, the method possesses a high degree of face validity. Rules of tactical doctrine are used to place units and equipment in the field, and the developed system attempts to formalize those rules and use them to place units and equipment in the model. Second, even though the finished system is rather complex, the conceptual framework is quite simple: in every instance in

which a rule is observed for placing a unit or equipment in the real world, the Scenarist model is expanded to include that rule (and the factors in terms of which the rule is articulated).

The disadvantages of the method are not yet clear. The current project was unable (because of resource constraints) to implement probability sampling in the system. For this reason, the system cannot (easily or directly) be used to generate probability samples of scenarios. This shortcoming is not an intrinsic shortcoming of rule-based systems, however, and could be overcome with additional development. Based on the TRAILBLAZER test of the Scenarist, it is clear that a considerable effort is required to assemble the information needed to define units and to devise rules, to modify the system so that it can determine all of the factors needed by the rules, and to implement actions as the result of the rules' "firings." Similar problems would occur, however, with any approach. Future use of the Scenarist system will reveal the its nature and the usefulness of using knowledge-based systems for scenario generation.

D. System Concept Flow Diagram

Figure 2 presents a macro-level flow chart that describes the system concept from a user's perspective. The flow diagram of Figure 2 presents the system concept from the point of view of what basic scenario-generation functions would be available to the user (or, in the case of system set-up, required of the user); it does

not present a flow diagram of the logic involved to implement those functions. This system concept is the concept that evolved during the course of the project, starting with the initial system concept presented in the System/Segment Design Document and continuing through the rapid-prototyping system development process.

Figure 2. System Concept Flow Diagram

System Set-Up:

- o Obtain digital mapping data
- o Create map files (of varying resolution) in Scenarist format
- o If desired, define background map files
- o Obtain data on unit compositions and organization
- o Describe units in Scenarist structure
- o Enter unit/platform/equipment symbols and labels
- o Code functions to draw (new) symbols
- o Obtain data on placement rules
- o Identify factors to be included in rules;
define rules; identify actions to be executed
when rules fire
- o Code functions to compute rule factors, actions
- o Code functions to execute rules, or enter rules
in CLIPS knowledge base
- o Set up Scenarist Project File (which specifies

rule the initial map files, unit files, and files to use)

System Processing:

Present User with a Menu of Function Choices:

- o Define Unit (specify unit boundaries, parent unit, subordinate items (subordinate units, platforms, equipments), objective, mission)
- o Display Map
- o Change Map Location, Resolution
- o Zoom Map
- o Place Unit on Map
- o Place Forward Edge of Battle Area (FEBA) on Map
- o Move Unit
- o Manually Reposition Subordinate Items of a Unit
- o Execute Rules to Reposition Subordinate Items of a Unit
- o Store Modified Unit
- o Print Screen
- o Output Scenario Description (Unit/Equipment IDs and Locations) to Printer or File

IV. System Requirements Analysis

A. The Original System Requirements

During the requirements analysis phase of the project design effort (i.e., the first six months of the project), several lists of system requirements were developed. These lists are presented in Tables 1-3, entitled "Original CECOM Requirements," "Original Vista Research Corporation Requirements," and "Original Other Potential Users, Requirements." They are described in detail in the System/Segment Design Document, and were originally presented as Tables C1-C3 of Appendix C of the Test Plan prepared prior to the test of the Scenarist software system. This section briefly describes these initial system requirements.

1. CECOM Requirements

CECOM has a requirement for an automated scenario-generation system because the current manual methods for generating and validating emitter laydowns are costly and time-consuming. The availability of valid automated means for producing emitter laydowns would facilitate CECOM's ability to perform its mission.

Table 1. Original CECOM Requirements

Develop a Scenario Generation System that:

1. Shall have the ability to accept and process small local geographic areas of Government-provided DMA terrain data. The Government will provide this data in a format defined by Vista Research on CD-ROM.

2. Shall have a user interface that accepts general scenario generation instructions from a military scientist.

3. Shall have the ability to access and process organizational data in the scenario generation process.

4. Shall output specific emitter laydowns to:

a. an ASCII file of laydown information that will be used by CECOM to create input to CECOM's ADEM Model. The specific definition and format of this laydown information will be provided to Vista Research Corporation.

b. an ASCII file of laydown information that will be used by CECOM to create input to CECOM's Dynamic Ground Target Simulator (DGTS) Model. The specific definition and format of this laydown information will be provided to Vista Research Corporation.

5. Shall provide the user with the ability to change the placement of specific emitters after the scenario has been generated.

6. Shall utilize knowledge-based (expert) system technology (a requirement of the contract Statement of Work).

7. Shall maneuver forces on the battlefield by force structure (a requirement of the contract Statement of Work).

8. Shall utilize digital terrain and threat parametric data (a requirement of the contract Statement of Work).

9. Shall allow a scenario generation capability with rapid turnaround (a requirement of the contract Statement of Work).

10. Shall generate complex threat scenarios by force structure at the brigade/division level (a requirement of the contract Statement of Work).

11. Shall be developable within the time and resources available under the Phase II SBIR contract awarded by CECOM to Vista for development of the system.

¶Table 2. Original Vista Research Corporation Requirements

Develop a Scenario Generation System that:

1. Shall be developed using hardware and software that is relatively inexpensive.

2. Shall be developed on a platform that will be marketable in Phase III to many potential users.

3. Shall generate a map of a specified geographic area of interest on a graphics display device.

4. Shall use terrain data as the source for map display and for establishing where units and equipment may be placed.

5. Shall use an expert system to position major units.

6. Shall provide a capability to specify planned movement of major units from original positions established by the laydown.

7. Shall have a user interface that accepts general scenario generation instructions from a military scientist.

8. Shall have the ability to access and process organizational data in the scenario generation process.

9. Shall output the laydown of major units to:

a. the system's graphics monitor

b. a file on tape or disk for subsequent input to other models, such as a tactical combat model.

10. Shall have a capability to generate probability samples of scenarios.

11. Shall offer the user the ability to restrict sampling to scenarios having specified characteristics.

12. A version of the system will be available for use on an 80386 computer.

¶Table 3. Original Other Potential Users' Requirements

A. USAEPG Requirements

Develop a Scenario Generation System that:

1. Can generate scenarios that can be used in dynamic (over-time) simulations of communications-electronics system performance
2. Is highly automated, i.e., can generate scenarios without requiring substantial manual involvement
3. Includes a capability for frequency assignment
4. Follows OPFAC rules (the OPFAC, or Operational Facility, process is a process for providing battlefield system configuration descriptions, grouping battlefield systems into operational facilities, identifying radio nets, and locating operational facilities on the battlefield)
5. Automatically processes VTAADS files (TAADS, The Army Authorization Documents System, is an automated system that supports the development and documentation of organizational structures such as Modification Tables of Organization and Equipment. VTAADS, or Vertical TAADS, is a multicommand standard automated system for those users having complete in-house functional and automated data processing (ADP) capabilities).
6. Has an interface to video graphics

7. Has an automated interface to the Communications Data Base

8. Automatically processes Scenario-Oriented Recurring Evaluation System (SCORES) Simulated Tactical Deployments (STDs)

B. USAICS Requirements

Develop a Scenario Generation System that:

1. Reflects the full and complete functions of MI battalions in a division-level operation

2. Generates scenarios that separately and collectively exercise the personnel and equipment during both the CPX and FTX elements of the combined exercises

¶Table 3 (cont.). Other Potential Users' Requirements

3. Generates scenarios that separately and/or collectively exercise the assets of the MI battalions in the areas of human intelligence (HUMINT), communications intelligence (COMINT), signals intelligence (SIGINT), counterintelligence, and intelligence products

4. Accommodates AirLand Battle doctrine

5. Is supported by a digital database comprised of digitized images, graphics, charts, overlays and photography that can be manipulated by the computer to depict enemy situations in any deployment on any terrain or map. The database

can project individual and/or collective equipments or forces. In addition, it can project any terrain to any scale that has been digitized and superimpose equipment and forces on that imagery to scale to match the projected imagery.

¶

The essential CECOM requirement is that the system determine noncommunications emitter locations by taking into account digital terrain data and tactical doctrine on the placement of emitters. The position locations will be input to a CECOM simulation model used to evaluate electronic warfare systems.

The CECOM requirements were derived in part from the original contract Statement of Work (SOW) and in part from discussions with CECOM personnel early in the project. They reflected what the CECOM personnel believed at the time were the most desirable system features. While many of the CECOM requirements are still desired, some of them became undesirable either because they did not lead to a good system design or because CECOM's intended applications for the system changed.

For example, the SOW included the requirement that the scenario generation system make use of the RAND Corporation Strategy Assessment System (RSAS); during the design phase, it was concluded that NASA's CLIPS knowledge-based system was much more appropriate for this application.

In addition, there was a requirement for the Scenarist to create input for CECOM's Air Defense Effectiveness Model (ADEM). As time passed, CECOM's interest in creating scenario input to ADEM diminished, and interest increased in having a capability to generate scenario input for the Command Systems Group Mission Planner.

Another problem that arose is that CECOM was unable to provide Vista with Defense Mapping Agency mapping data on CD-ROM disks. This difficulty stemmed mainly from the fact that the data have just recently become available on the CD-ROM medium, and they are not readily available. Vista did not possess the data processing equipment (9-track tape drives) necessary to read the DMA data from the medium on which the data were readily available.

As a compromise, it was decided that the development goals of the project would be realized if Vista simply used digital mapping data that the firm already had access to, in sample data files provided with the US Army's GRASS geographic information system. The issue of obtaining mapping data on CD-ROM would be left for a later development effort.

¶The basic requirement of CECOM is for an automated system that can position noncommunications equipment on a battlefield, taking into account digital terrain data and tactical doctrine. This scenario-generation capability will be used to support simulation studies of electronic warfare systems. After the scenario-generation system has specified the

locations of noncommunications emitters, the simulation models will fly aircraft over the battlefield, and determine what emitters are detected. This information will be used to assess the performance of the electronic warfare detection systems.

At the time of the requirements specification phase of the project, there were two systems for which CECOM desired scenarios: the Dynamic Ground Target Simulator (DGTS) and the Command Systems Group Mission Planner. For either application, what was desired is an ASCII file containing emitter locations, which could be used by CECOM to create input files, properly formatted (by CECOM personnel) for input to the appropriate system.

Table 1 ("Original CECOM Requirements") presents a list of the CECOM requirements that were identified during the requirements specification phase of the project.

2. Vista Research Corporation Requirements

When Vista prepared its SBIR Phase I proposal to develop an automated scenario generation system we were not aware of CECOM's particular applications for such a system. We had determined that a general need for such a system exists in many US military organizations. As part of the needs assessment we conducted prior to preparing our proposal, we had developed our own list of system requirements. These requirements were general, intended to result in a general-purpose scenario-generation system

that, with appropriate "tailoring," could be of value in a variety of applications.

The project to develop an automated scenario-generation system was funded under the Small Business Innovation Research (SBIR) program. It is a stated goal of the SBIR program that SBIR contracts should be directed toward the development of systems that are of general benefit to the country, and not of value solely to the funding organization. More specifically, it is a goal of the program to fund research that has the potential for "commercialization," in Phase III of the development program. For defense applications, "commercialization" corresponds to the use of the development by organizations other than the funding organization.

With these SBIR program objectives in mind, Vista developed a number of system requirements that, in our view, would have promoted the value of the automated scenario generation to a variety of other potential system users, in Phase III of the development effort. Table 2 ("Original Vista Research Corporation Requirements") presents a list of the requirements identified by Vista Research Corporation.

3. Other Potential Users' Requirements

¶In addition to identifying requirements that we believe would make the system of value to a wide user community, we contacted two specific potential Phase III users, in an attempt to identify their requirements for an automated scenario generation system. These two

organizations were the US Army Electronic Proving Ground (USAEPG) and the US Army Intelligence Center and School (USAICS), located at Fort Huachuca, Arizona. Table 3 ("Original Other Potential Users' Requirements") presents requirements that were derived from these contacts, or from available literature on their missions and current activities.

Note that there was no contractual requirement to develop, in the current project, a system that would satisfy all of the requirements of potential Phase III users, such as USAICS or USAEPG. What we hoped to accomplish in considering the requirements of these potential Phase III users was an awareness of what capabilities would be desirable in enhanced versions of the Scenarist, after the Phase II development for CECOM was completed. By considering the additional future requirements for an automated scenario generation system, it was hoped to develop a system design that would facilitate future system expansions.

B. The Revised System Requirements

The Scenarist development project was a research project. While the course of the development was guided by the various system requirements that were identified at the beginning of the project, the research nature of the project and the "rapid-prototyping" approach to the software development allowed for the possibility that the requirements could be modified during the course of the system development process.

Tables 4 and 5 ("Final CECOM Requirements" and "Final Vista Research Corporation Requirements," originally presented as Tables D1 and D2 of Appendix D of the Test Plan) present lists of the system requirements as they evolved over the course of the project. These final requirements reflect the general intent of the original Statement of Work, but they differ in specific detail from the specific detail included in the original SOW. They are an evolutionary product of the development efforts of the past two years, and they differ a little from the preliminary requirements identified during the formal design phase (first six months) of the project.

The Tables 4 and 5 requirements lists are very close to the CECOM Requirements and the Vista Research Corporation Requirements developed in the design phase and presented in Tables 1-3, modified to account for circumstances that arose

¶ Table 4. Final CECOM Requirements

Develop a Scenario Generation System that:

1. Shall have the ability to accept and process small local geographic areas of digital terrain data. The contractor may use sample data available from the US Army's GRASS geographic information system to demonstrate this capability.
2. Shall have a user interface that accepts general scenario generation instructions from a military scientist.

3. Shall have the ability to access and process organizational data in the scenario generation process.

4. Shall output specific emitter laydowns in the form of an ASCII file that specifies the identification and location of equipment.

5. Shall provide the user with the ability to change the placement of specific equipment after the scenario has been generated.

6. Shall utilize knowledge-based (expert) system technology (a requirement of the contract Statement of Work).

7. Shall maneuver forces on the battlefield by force structure (a requirement of the contract Statement of Work).

8. Shall utilize digital terrain and threat parametric data (a requirement of the contract Statement of Work).

9. Shall allow a scenario generation capability with rapid turnaround (a requirement of the contract Statement of Work).

10. Shall generate complex threat scenarios by force structure at the brigade/division level (a requirement of the contract Statement of Work).

11. Shall be developable within the time and resources available under the Phase II SBIR

contract awarded by CECOM to Vista for development of the system.

¶Table 5. Final Vista Research Corporation Requirements

Develop a Scenario Generation System that:

1. Shall be developed using hardware and software that is relatively inexpensive.
2. Shall be developed on a platform that will be marketable in Phase III to many potential users.
3. Shall generate a map of a specified geographic area of interest on a graphics display device.
4. Shall use terrain data as the source for map display and for establishing where units and equipment may be placed.
5. Shall use an expert system to position major units.
6. Shall provide a capability to specify planned movement of major units from original positions established by the laydown.
7. Shall have a user interface that accepts general scenario generation instructions from a military scientist.
8. Shall have the ability to access and process organizational data in the scenario generation process.

9. Shall output the laydown of major units to:

a. the system's graphics monitor

b. a file on tape or disk for subsequent input to other models, such as a tactical combat model.

10. A version of the system will be available for use on an 80386 computer.

¶during the implementation phase. The changes to the original list were as follows:

1. In the CECOM Requirements, the requirement for the Government to provide digital mapping data on CD-ROM was dropped, and reference to inputting scenario data to specific models (e.g., ADEM, DGTS) was dropped. (The Government was unable to provide the mapping data, and agreed to reformat the equipment-location data produced by the system.)

2. In the Vista Research Corporation Requirements, the two requirements relating to the capability to generate probability samples of scenarios and allow for statistical sampling of scenarios were dropped. (These requirements are technically feasible, but project resources did not allow for their development.)

3. The Other Requirements were dropped. These requirements represent reasonable

extensions of the basic Scenarist, but they are not directly related to the project Statement of Work.

As verified in the system software test and described in the Test Report, the Scenarist system satisfies all of the requirements listed in Tables 4 and 5.

IV. Scenarist System Top-Level Design

A. System Architecture

The Scenarist consists of two major subsystems: the hardware subsystem and the software subsystem. The hardware subsystem includes a microcomputer system that accomplishes all of the processing and display functions of the system. The software subsystem includes all of the computer programs that control the operation of the computer and its processing functions, and the computer data files accessed by the programs.

1. Hardware Subsystem

The hardware subsystem is the tangible equipment that accomplishes the data input, storage, retrieval, presentation, and processing necessary to generate a scenario. The hardware system selected for the Scenarist platform was an 80386 microcomputer using an MS-DOS operating system and possessing the following equipment: 1 megabyte of direct-access memory, a 40-megabyte hard disk, a high-density 3-1/2" diskette drive, a VGA color monitor, and a mouse.

(Note: The distribution diskettes are also available on 5-1/4" double-density diskettes for users who have 5-1/4" diskette drives.)

During the consideration of hardware alternatives, consideration was given to other graphics workstations as the system platform. Strong consideration was given to using a Sun SPARCstation for the platform. This alternative was dropped in favor of the 386-based microcomputer for several reasons. First, although the SPARCstation was faster than the 386 machine, it was believed that the system could in fact be implemented on the 386. CECOM did not impose any firm requirements relative to running speed, so no strong advantage was perceived to accrue from expending more money to obtain faster speed. Second, no project funds were to be expended on hardware. The firm provided 386SX "personal" computers to each of the technical staff. Since these computers could do the job, a management decision was made not to expend the firm's resources for the purchase of the more expensive SPARCstation equipment, of which only a single station would be available. Third, it was considered that having the system available on a low-cost microcomputer could be an advantage to marketing the completed system. Fourth, the CECOM SIGINT/ESM Branch did not possess the SPARCstation, and had no immediate plans to acquire one. The Branch did possess a 386 machine.

For all of the preceding reasons, the decision was made to implement the Scenarist on a 386-based microcomputer.

¶At the time that this decision was made, the only significant perceived disadvantage of the 386-based system relative to the SPARCstation was speed. During the course of the project, however, another significant disadvantage was realized. That disadvantage was the perception, expressed by some potential users (who were in fact SPARCstation users) after viewing the Scenarist demo, that a "PC-based" system would not be able to handle large-scale applications. The fact that the Scenarist design had not been constrained by the decision to place it on a 386-based platform, and that its C-based software could be converted to run on a SPARCstation was of little interest.

2. Software Subsystem

The software subsystem is the intangible system component that controls the hardware. It includes the computer operating system, high-level language compilers, program development tools and other utilities, additional commercial software packages (e.g., geographic information system, expert system, data base management system), and project-developed application-specific software. The software system also includes the data bases used by the system for storage of terrain data, unit and equipment data, scenario descriptions, and expert rules relating to placement of units and equipment.

The major features of the Scenarist software system are the following:

- o Microsoft MS-DOS operating system
- o C programming language used for programming all software developed for the Scenarist system, including all graphics, windows and mouse control code, and all geographic processing
- o Microsoft C compiler
- o No geographic information system package
- o CLIPS knowledge-based system
- o No data base management system package
- o No program development language automated software development tools

The paragraphs that follow describe each of the major software features of the Scenarist, and summarizes the reasons for the feature inclusion or exclusion.

a. Computer Operating System

The computer operating system (e.g., MS-DOS, UNIX, VMS) is the software that performs the basic operations required to run the computer, such as transfer of data within the machine, storage and retrieval of data, and communications with external devices such as video monitors, tape drives, and printers.

¶The Microsoft MS-DOS operating system is the most common operating system available for single-station 386-based microcomputers. That system was adopted for use in this project. Some consideration was given to also developing a

UNIX-based version of the system. The availability of a UNIX-based version might have appealed to some potential customers. The firm possessed a UNIX-based 386 microcomputer configured for use with the US Army's GRASS geographic information system. An advantage of this system for the Scenarist application is that it possessed two video monitors -- a high-resolution color monitor for attractive graphics, and a monochrome monitor for control. Although this system was used at some points in the system development (primarily to use the Microsoft 6.0 Programmer's Workbench CodeView debugging features), no attempt was made to develop an alternative dual-screen version for this machine.

There were three reasons for this decision. First, developing a single version avoided diluting the project resources, which were already strained in accomplishing just a single version of the Scenarist. Second, the marketability of the Scenarist was considered better for a 386-based MS-DOS microcomputer than for a dual-terminal UNIX-based workstation. Third, as a research and development effort to develop a new tool, there was no guarantee that the Scenarist development project would in fact succeed in developing a successful automated scenario generation system. From the point of view of maximizing the probability of success of the Scenarist, it was considered better to place all of the resources into developing the best design possible, than to allocate some of those resources to the development of two versions.

b. Development Software

While some components of the Scenarist might have been available as off-the-shelf packages, it was recognized that a significant amount of custom programming would nevertheless be required. Apart from graphics software, software would be needed to compute the values of factors to be used in rules, and to execute actions that resulted from the firing of rules. Also, software would be required to process military units (e.g., define units, move units, reconfigure units). It was necessary to select a high-level programming language for this part of the development effort.

¶It was decided to program the Scenarist system in the C programming language, which is the most popular programming language for microcomputer-based graphics systems. Most of the commercial or public-domain software packages that might have been included in the Scenarist were available in C, and extensive libraries of C functions are available at low cost. C compilers are available for microcomputers at low cost. The Microsoft C compiler was selected since the firm had had successful previous experience with Microsoft products and already possessed a Microsoft C compiler prior to the project.

The Scenarist could have been coded in other languages, such as FORTRAN or Ada. Ada was not selected primarily because of the high cost of Ada compilers. C offered both advantages and disadvantages over FORTRAN. A disadvantage of

C is that the language does not have standard or powerful input/output (formatting) features, as FORTRAN does, and the coding of input/output takes more time. A feature of C that is both an advantage and a disadvantage is that it covers a wider range of capabilities than FORTRAN. Both languages are "high-level" programming languages, but C contains many "low-level" features. While this offers the programmer more flexibility, it results in slower debugging. With FORTRAN the programmer is more restricted in what he can do. Because of this, he makes fewer mistakes, and FORTRAN compilers detect and correctly identify programming errors better than C compilers. Because of the "pointers" used in C, program control can be directed to points outside of the program and "crash" the system.

Another potential disadvantage of C relative to FORTRAN is that C offers the programmer much more flexibility in how procedures are coded. If care is not taken to produce easy-to-follow structured, commented code, C can be much more difficult to maintain (i.e., for a person other than the original programmer to locate an error or modify the code) than FORTRAN.

A major advantage of C over FORTRAN for the current application is the fact that it involves a considerable amount of data handling, and is heavily object-oriented. The C language includes the construct of "data structures," whereas FORTRAN does not. A data structure is a data type that is substantially more general than the data types used in FORTRAN (such as

character, integer, real, or array). A data structure is an ordered collection of variables of various types (including other data structures). A data structure corresponds to the concept of a record in a file, or an observation in sample survey work. The reason why data structures are so useful in object-oriented applications is that all of the attributes about an object may be combined into a single data structure, which may be easily handled (e.g., "passed" to a function simply by reference to the address of its starting location in the computer's memory). Similarly, the reason why data structures are useful in applications involving a lot of data handling is that an entire record may be defined as a data structure, so that entire records may conveniently be written to or read from files (e.g., written into a file or read from a file with a single, simple command).

¶Because a data structure may include all of the information about an object, the processes of developing, documenting, and maintaining algorithms involving complex collections of data is facilitated. Data structures are one of the two basic abstract elements of computer programs and of modern software engineering (the other being algorithms). Their availability substantially facilitates the description (and development and maintenance) of a data processing system in terms of data flow through "modules" (functions that operate on the data structures). Although data structures are powerful, convenient constructs, their use is not completely trouble-free. For example, if it

becomes desirable to add a variable to a data structure, the data structure definition can readily be changed in the program code. Unfortunately, however, none of the files based on the original data structure definition can now be read; the data in them must be read out using the old data structure definition and rewritten using the new data structure definition. In a FORTRAN program, the programmer could have simply defined a new variable and a new file for storing that variable. (The advantage in fixing the program could, of course, be offset by a disadvantage in maintaining the program, because the new variable is "separated" from other related variables rather than combined with them in a data structure.)

In general, C offered both advantages and disadvantages over FORTRAN, and the primary basis for the decision to choose C is that it is "the" language for microcomputers, especially those involving graphics. Because C is so widely used for microcomputers, good compilers and development environments (e.g., Microsoft Quick C, Programmer's Workbench, Borland's Turbo C Debugger) are available from alternative sources, and excellent telephone product support is available.

The project began using Microsoft C version 5.1, and ended using version 6.0. The Microsoft Quick C and Programmer's Workbench products were used throughout the development process.

¶Except for Quick C and the Programmer's Workbench, it was decided not to use any other

automated software development tools. The primary other candidate for use would have been a program development system (such as EXCELLERATOR) that implemented structured, top-down design. We decided against using a program development language for several reasons. First, the firm did not possess any of these tools at the beginning of the project, and the staff assigned to the project had no experience with these tools in an MS-DOS/microcomputer environment. A cost would have been incurred to compare available tools, select one (or more), purchase it, and learn how to use it effectively. Because of the tight development schedule, this appeared to increase the development risk rather than reduce it. Instead, the modern software engineering discipline (top-down, structured design) was implemented essentially by manual means (although a CAD program was used to produce data flow diagrams).

A second reason for not using automated software development tools was that, because of the rapid prototyping approach, it was recognized that the system design would undergo substantial and rapid change. There was a concern (perhaps unfounded) that, in spite of the advantages of these tools, their use could introduce an additional layer of development complexity that might actually impede our ability to make a large number of system design changes rapidly (especially because of the decision to embed a large amount of "foreign" software -- the CLIPS knowledge-based system -- in the Scenarist).

A comment is made about the management of the software development in the Scenarist development project. Throughout the course of the project, there were at most only two full-time development staff (and only one at times). When a design change was considered, it was discussed by the two staff members and an appropriate action was taken by one of the two staff members. The code was changed and a copy of the changed modules was given to the other person for the purpose of updating his now-obsolete version. Both persons were working simultaneously on different parts (i.e., modules) of the code, and the second person would also transfer a copy of his updated modules to the first person at frequent intervals ("update points," usually every few days). This process worked quite well because of the small size of the project -- only two persons, working in close coordination. It also worked well because few changes were made in the overall design, so that the interrelationships among program modules were rarely changed. It would not have been a workable approach for a large software development effort. It is not known whether this process would have been affected (either impeded or facilitated) had the system software been embedded in a software development tool.

We considered using windows software, but decided against it. The Microsoft Windows Software Development Kit (SDK) was not available at the beginning of the project. By the time we became aware of it, a substantial amount of code had already been produced. While the SDK appears to be a useful development environment,

it is designed for use from the beginning of a project. It is not a windowing function package that can be imbedded in an existing software system.

¶We also considered the Vermont Views package. That package is designed for inserting windows functions into existing code. We decided against using Vermont Views because the licensing agreement would have required a payment for every copy of the Scenarist delivered to a future customer. While this cost was not high for the microcomputer/MS-DOS version of the package (several hundred dollars), it was very substantial (several thousand dollars) for other platforms/systems.

Having considered and rejected several commercial windows interface packages, we decided to rely on the basic graphics interface functions that were provided as a library with the Microsoft C compiler, plus additional functions available in the public domain in numerous publications. The Microsoft graphics interface functions generally work well, and, by taking advantage of published software for windowing and mouse interface, we quickly integrated a basic (and low-cost) windows / mouse interface capability into the Scenarist.

The decision to use Microsoft compiler and development-language software (MS-DOS, C) was not based on a comparison with other vendors (e.g., Borland). It was based on prior familiarity with Microsoft products. The Microsoft product support (telephone) provided

to this project was excellent. The service was free, competent, and friendly, and the response was almost always immediate.

During the course of the software development, some problems were encountered with the Microsoft C graphics software (the "floodfill" commands worked erratically), and many hours of time were spent in debugging "pointer"-related errors, especially related to files (pointers are variables used in C to store the addresses of data structures, for easy reference to them). The Microsoft C compiler was not very helpful relative to preventing these errors, and in many cases the error condition identified at the time of program run failure was of no help at all in either identifying the true nature of the problem or suggesting the source of the error.

¶Toward the end of the project, a substantial amount of time was lost in searching for system-related "bugs" that had nothing to do with the design logic or program code. These errors arose because of the large size of the system (including both the Scenarist development code and the CLIPS code), and the requirement to use a lot of files and a lot of memory. The system would often "lock up," and the available debugging tools (code review, Microsoft Programmer's Workbench Codeview) were of little help. These problems were extremely troublesome, for two reasons: (1) they developed when the system was large, and they were very difficult to diagnose and correct; (2) they developed at the very end of the system development, when most of the project resources

and time had been expended, and no flexibility remained for reallocating project resources or revising the project schedule to accommodate them. The problems were eventually resolved by reducing the number of files and memory used by the system.

If the Scenarist development is continued, the system size will likely increase, and it would be expected that such problems would reoccur. Because of the substantial expense associated with searching for system-related errors, it is recommended that serious consideration be given to changing the compiler, even though the conversion cost would be significant. Microsoft's integrated development environment (Programmer's Workbench) was of limited value in tracking down the system-related problems we encountered. Also, it could not handle nested "include" files that were part of the CLIPS system. Also, no complete hardcopy run-time library was delivered with Microsoft C version 6.0.

One of the issues to be considered in the selection of the developmental programming language is whether to use a traditional procedural programming language (such as C or FORTRAN) or an object-oriented (procedural) language such as C++. Many artificial-intelligence and simulation applications are implemented using object-oriented programming. In object-oriented programming, entities called "objects" have attributes, often called "slots." Objects interact (or "communicate") by passing

"messages" that affect the values of the attributes. Related objects may be grouped into classes. Three important features of object-oriented programming are encapsulation (data hiding), polymorphism, and inheritance (subclassing).

Encapsulation refers to the ability to group specific types of data and the means to manipulate the data (i.e., the functions that process the data) into a class. That is, procedures ("methods") and their variables and constants are "hidden" inside the objects. Each object is an instantiation of an object type or class, just as, for example, the integer variables of a standard FORTRAN or C program are instantiations of the integer variable type.

Polymorphism refers to the ability of an object to select a procedure for processing data as a function of the data type. With the polymorphism feature of object-oriented programming, it is easy to generalize concepts involving relationships among objects, such as the distance between two objects. This is possible because the functions encapsulated in an object can sense the data type of a message sent to the object and be defined to operate in different ways on different types. For example, a "distance" function may be defined that takes two different objects as arguments, e.g., `dist(a,b)` where `a` and `b` denote the objects. The significant convenience is that with object-oriented programming this function may easily be defined to accommodate objects of different types (e.g., `dist(RED CORPS 1, BLUE`

CORPS 2) or dist(PLATOON 2, PLATOON 4) or dist(unit1, unit2)). Conventional programming languages were not designed to enable easy implementation of this type of construct. Since object-oriented languages are better-suited to representing the objects and processes that occur in simulations and knowledge-based systems, the use of such languages generally results in higher development productivity.

Inheritance refers to the ability to define new classes which assume the same characteristics as existing classes, plus added characteristics. These new classes, called subclasses, form a hierarchy. The new class contains only code and data for new or changed methods.

The special features of object-oriented programming (encapsulation, polymorphism and inheritance) enable rapid and efficient development of a top-down model design, and are particularly useful in implementing object-oriented models (such as simulations and rule-based systems).

C, like FORTRAN, is a conventional procedural language that is efficient to use for numerical processing, whereas C++ is an object-oriented (also procedural) language designed to facilitate description of the interactions among entities. C++ is hence a natural language to consider for use in implementing simulations or knowledge-based systems, which are heavily concerned with interrelationships among entities. The current implementation of the Scenarist, however, is concerned primarily with

offering the user a menu of choices for operating on maps and units. It is not a dynamic simulation. Most of the processing is concerned with execution of functions that involve numerical processing, rather than with sending vast numbers of "messages" among objects that are moving over a battlefield. Furthermore, the expert-system portion of the system (CLIPS) was already coded in Microsoft C. For these reasons, there is little reason to believe that the special features of an object-oriented language (messages, encapsulation, inheritance) would have facilitated the system development or resulted in processing speed advantages. Since the firm already possessed a Microsoft C compiler, it was initially selected for use over an object-oriented C compiler.

¶When some initial problems were encountered with the Microsoft C compiler, a Borland C++ compiler was obtained, and serious consideration was given to switching to C++ from C. The Borland C++ compiler is a strong alternative to the Microsoft C compiler for several reasons: (1) it is object-oriented; (2) Borland has a very good reputation in C products for microcomputers (Borland developed Turbo C; the similar Microsoft product Quick C was introduced after Turbo C proved so successful). The conversion might have necessitated a project schedule revision, however, since some coding had already been done in Microsoft C. (Since the Scenarist design was object oriented, it could have easily been implemented in an object-oriented language. No changes to the basic system design would have been necessitated by a switch.) Because the

perceived advantages of C++'s special features were not great, however, and because the problems extant at the time were cleared up, it was decided not to switch.

Now that the project has been completed, we realize how costly the occurrence of system-related errors is. If it is true, as some users say, that the Borland C++ product encounters fewer system errors than the Microsoft C compiler, this would be a strong reason for switching to the Borland C++ compiler. The experience near the end of the project with the Microsoft C compiler and Programmer's Workbench integrated development environment was costly to the firm, and ways of reducing system development risk are certainly welcome! (For more discussion and comparison of Microsoft C versus Borland C++, see the review by Ray Duncan on pp. 441-444 of the July 1991 issue of PC Magazine.)

If the Scenarist is extended to include dynamic simulation, additional consideration should be given to converting the system to C++. Possible extension of the Scenarist to include dynamics was not a major issue considered in the design. The system was to be able to accommodate unit movement in the sense of repositioning a unit, but it was never intended that the system would be able to simulate continuous movement of units over the battlefield.

c. Geographic Information System

A geographic information system (GIS) is a software system that stores, processes, and displays geographic data. A GIS was a candidate for inclusion in the present project because of the requirement to process digital terrain data (one of the system requirements is to develop an automated scenario generation system that can take into account digital terrain features).

GIS analysis and display capabilities include the ability to display maps, to overlay various features on maps, to manipulate map images, and to locate, extract and plot geographic features. Many person-years of development effort have gone into the development of GIS software. Capable GIS systems are available at low cost, obviating the need to develop, in this project, new software for conducting basic processing operations on digital terrain data. In addition, the development of general-purpose GIS software is not in the scope of this project.

¶Initially, it was decided to use the US Army's GRASS geographic information system as a component of the Scenarist. GRASS was selected for several reasons:

- o low price (approximately \$1,000)
- o availability of support
- o availability on a variety of platforms
- o includes "cell-based" map representations (in addition to "vector" representations),
facilitating use within analytical models

- o availability of functions to analyze and manipulate stored geographic information
- o availability of source code (written in the C programming language)

The selection of GRASS imposes general hardware and operating system requirements on the system/software configuration. These minimum requirements include:

- o UNIX operating system
- o virtual memory
- o graphics library
- o 4 Mb system memory
- o Color Graphics Monitor (256 color display) with mouse
- o Alphanumeric Monitor
- o 300 Mb hard disk
- o Tape drive (1/2" or 1/4")

There are many platforms available that meet these minimum requirements, ranging from Silicon Graphic's IRIS machine to Apple's Macintosh. Although Vista possessed a 386 machine configured as required above, the CECOM SIGINT/ESM Branch did not. After some time, it was realized that the Branch would not have available a machine of the above configuration, and the decision to use GRASS was rescinded. Instead, the geographic processing would be done using the C programming language.

d. Knowledge-Based System

A knowledge-based system (KBS), or expert system, is a software system that can emulate the behavior of human beings ("experts") in a particular subject-matter area, called a domain. A KBS was required in the present project because the contract Statement of Work specifically required that the automated scenario generation system be based on a knowledge-based system.

¶A KBS generally consists of a knowledge base that contains a representation of the experts' knowledge (e.g., rules), a global data base that contains dynamic data relating to a particular execution of the KBS, a control section that applies the information of the knowledge base to the dynamic data, and a user interface for developing and using the system.

There are several different paradigms for KBSs, involving procedures such as forward chaining, backward chaining, and learning by examples. A variety of KBS "shells" have been developed. A KBS shell is simply a software package that implements a particular type of KBS.

It was decided to use the NASA-developed and supported C-Language Integrated Production System (CLIPS) as the expert system for the Scenarist. This decision was based on a number of factors. First, CLIPS is low-cost (in fact, free to US Government contractors), and it could be provided to the Government or to Government contractors as part of the Scenarist package without their having to pay a fee. CECOM already possessed a copy of CLIPS. CLIPS is programmed in C (also in Ada), and was designed to be

imbedded in systems such as the Scenarist. As described in the System/Segment Design Document, we examined a number of other KBSS (both commercial and non-commercial systems). While some of them (particularly, Knowledge Shaper) possessed appealing features, they were expensive. Furthermore, none of them provided the source code, so that integration with the Scenarist could have been problematic. It was quite possible to achieve the goals of the Scenarist development project using CLIPS, at no cost and low development risk (because of the availability of C source code). If the Scenarist develops a market, it is possible to consider interfacing it with another KBS at a later date.

In the development of the Scenarist, a decision was made to program the rules in the C-language prior to implementing them in CLIPS. The reason for this was to verify that the CLIPS system was operating as intended. For both of the test cases considered during the project (a problem involving the placement of field artillery and air defense radars, and a problem involving the placement of TRAILBLAZER units), the system can be operated either using the C-language rule functions or the CLIPS system. In these test problems, the C-language rule functions were observed to execute somewhat faster than the CLIPS. This was expected. The basis for the decision to include CLIPS in the Scenarist was not speed, but the contractual requirement to include a knowledge-based system in the Scenarist, and the advantages of the CLIPS over other alternatives (e.g., low cost, C source

code, support, design for embedding in another system).

e. Data Base Management System

A data base management system (DBMS) is a software system for storage and retrieval of data. A DBMS may be used in the present application to store data on terrain features, unit and equipment characteristics, unit organization, and scenario characteristics. Data in the knowledge base would typically be stored within the knowledge base system.

To maximize system speed and simplicity, all of the data needed by the Scenarist were stored in files accessed directly by the Scenarist software. While there may be some role for a DBMS at a future date (e.g., to store scenarios in a library for retrieval according to specified characteristics), there was no need for one during the course of the development effort.

f. Data Base

In addition to the software described above, the software system includes a number of data files that contain data on terrain, unit and equipment characteristics, unit organization, and scenario descriptions. The data on tactical rules of placement are included in the knowledge base, controlled by the KBS software.

With respect to map data, the project used either hypothetical data (generated by hand and entered into data files via the keyboard) or sample map

data provided with the GRASS geographic information system. As mentioned, we were not successful in obtaining examples of Defense Mapping Agency data for use in this project.

With respect to unit data, all data were manually generated. An attempt was made to obtain electronically stored unit data maintained by the Battlefield Electromagnetic Environment Office (BEEO), but the cost of obtaining these data was considered too great.

Data used to define rules were extracted from published field manuals. For example, in the TRAILBLAZER application, several field manuals describing the organization and tactical deployment of the system were available at the US Army Intelligence Center and School. (These documents are referenced in the Test Report.)

¶VI. Scenarist Development Software Top-Level Design

A. Chapter Overview

This chapter presents a description of the top-level design of the developmental software component of the Scenarist. (The term "developmental software" refers to software developed during the course of the project, as distinguished from commercial off-the-shelf software, such as CLIPS or the Microsoft C compiler.) This chapter and the next (on detailed design) describe the design of each component of the software system; they do not

describe how to use these components. How to use the system is discussed in the User's Manual and in the Test Report in Appendix A. Also, this chapter and the next do not present a detailed description of the software itself (i.e., data flow diagrams, listings). The software documentation is presented in the Programmer's Manual and Variable Glossary. What this chapter does present is a description of the system design from conceptual, functional, and structural viewpoints. Knowledge of this information is not required for a user to be able to run the system, any more than a knowledge of the design of a word-processing, spreadsheet, or accounting package is necessary to use the system. This information is presented to provide the Government with an understanding of how the system works and the nature of its capabilities and limitations. It will enable the Government to make informed decisions about future development and application of the Scenarist.

This chapter presents a brief discussion of the design of the software system components. The chapter provides a description of the software system architecture (system components and their interrelationships) and an overall conceptual description of the functions of each software system component. The following chapter on detailed design provides detailed information on how these functions are accomplished.

The top-level design is presented in terms of the components that comprise the developmental software subsystem. To facilitate the

discussion of the procedures for using the Scenarist, the order in which these components are presented bears a close relationship to the order of the steps that a Scenarist user takes to operate the system. The components of the software system, and the order in which they will be discussed, is as follows:

1. Data Assembly, File Preparation, and Program Preparation

- a. Map Data

- i. Cellular Map Data (for Analytical Purposes)

- (1). Terrain-Type Map Files

- (2). Elevation Map Files

- ¶(3). Road Map Files

- (4). Cellular Map Compression Software

(for

extracting a lower-resolution cellular map

from a higher-resolution cellular map)

- ii. Vector Map Data (for Background Maps)

- (1). Area Map Objects File

- (2). Linear Map Objects File

- (3). Point Map Objects File

- iii. Software for Preparing Scenarist Map Files

from GRASS Map Files

- b. Unit Data

- i. Unit Structural Specification

- ii. Unit, Platform and Equipment Labels and Symbols

- c. Rule-Related Data

- i. Rule Formulation

- ii. Factors
- iii. Preprocessing
- iv. Rule Specification (in CLIPS rule file
or C functions)
- v. Actions
- d. Program Initialization (Project Files)

2. Scenarist Program

- a. System Control Functions (Menu Choices)
 - i. Project Selection
 - ii. Initial Map Location Point
 - iii. System Description
 - iv. Processing Military Units
 - (1). Define Unit
 - (2). Copy Unit
 - (3). Delete Unit
 - (4). Reposition Unit by User
 - (5). Reposition Subunits by User
 - (6). Reposition Subunits by Rules
 - (7). Reposition FEBA
 - (8). Display Unit
 - (9). Output Unit
 - v. Rules
 - vi. Processing Maps
 - (1). Draw Terrain Map
 - (2). Draw Elevation Map
 - (3). Draw Road Map
 - (4). Add Vector Map with Labels
 - (5). Add Vector Map without Labels
 - (6). Draw Vector Map without Labels
 - (7). Place Unit on Map
 - (8). Zoom Map
 - (9). Change Map Location
 - (10). Change Map Files
 - (11). Print Map

- vii. Scenario Generation
- viii. Hardware Data
- ix. Exit
- b. Data Input Functions
- c. Functions for Processing Units
- d. Functions for Processing Maps
- e. Data Output Functions

As is seen from the above list, there are two major components to the Scenarist software system. The first component comprises all of the activities that are necessary to set the Scenarist program up. This component includes collection of map data, unit data, and data on tactical doctrine related to the application of interest, and preparation of data files to be used by the Scenarist program. The second component is the Scenarist program. This program reads the data files, enables the user to define units, displays maps and places units on those maps in accordance with the user's instructions, and outputs data to the printer or disk files.

B. Scenarist System Concept

1. Operational Concept

The operational concept for the Scenarist is as follows. The user specifies a geographic area, and displays a background map of the area on the screen. He defines a number of military units in terms of the composition and locations of their subordinate units and equipment, and displays them on the map. He is provided with

options for repositioning units and their subordinate items (units, platforms, or equipments). He may manually reposition (move) entire units, or reposition certain of their subordinate items. He may specify rules for positioning subordinate items in a unit and request the system to reposition the subordinate items in accordance with the rules. The rules may take into account geographic features, the friendly mission, and the enemy threat. If an item's position violates a rule, the user is informed. The user may print the screen or output the unit and equipment locations to a data file.

The geographic features are defined in terms of digital terrain data, such as are contained in Digital Terrain Elevation Data (DTED) and Digital Feature Analysis Data (DFAD) maps produced by the Defense Mapping Agency (DMA). The friendly mission is specified in terms of an objective and mission for each unit. The enemy threat is specified in terms of the enemy units defined and positioned by the user. The rules for positioning units and equipment are derived from tactical doctrine and specified to an expert system.

¶(A note is in order here concerning the meaning of the expression "digital mapping data." In this report, the expression is usually used to refer to a relatively macroscopic quantitative (numerical) description of a map, such as a set of values for every cell of a rectangular grid covering a geographic area (a "cellular" representation) or a set of descriptions of

geometric objects (points, lines, areas) in an area (a "vector" representation). In this report, we are generally referring to digital map representations in which the data are somewhat aggregated, so that they may be used directly in numerical computation of functions used in rules and in rule (logical) processing. We are generally not referring to video map images ("raster data" or "image data") stored on laser disks for input to microcomputer color video monitors or television screens, even though these media may store data digitally and are certainly part of the field of digital geographic information systems. These representations either define a raster color television image or define the color of every pixel on a color computer monitor. They provide beautiful color map backgrounds, and they could certainly play a role in future versions of the Scenarist as background maps. While these mapping systems produce exceptional color map images, they do not store geographic data in a way that is convenient for object-oriented analysis (e.g., in an aggregated cellular or a vector representation of a conventional geographic information system or an object-oriented CAD system).)

The user is provided with a capability of viewing a large geographic area (containing one or more divisions), and zooming to a very small geographic area (showing individual items of equipment). The user is able to process rules using map data of varying levels of resolution. The level of resolution may differ for the different types of data (terrain type, elevation, or roads) for which data are

available. The system requires the user to provide elevation and terrain-type data, but the provision of road data are optional. The user may apply the rules to reposition subordinate items at a low level of map resolution (thereby taking into account the gross terrain features) and then proceed to apply the rules to reposition subordinate items at successively higher levels of resolution, up to the resolution limit of the available maps. The user is provided with features to enable the easy retrieval of maps of differing resolutions, and for specifying the "window" through which he wishes to view the map.

The user may relocate certain subordinate units and all platforms and equipment in a unit, and may relocate an entire unit. If a unit is moved, the system automatically repositions every subordinate unit, platform, or equipment in the unit.

¶In order to make the system easy to use, a number of the control and input features were implemented using windows, menus, and a mouse. At the beginning of the project, however, the Government project officer requested that a minimal amount of project effort be allocated to the development of sophisticated graphics and input/output features. It was desired that most of the project effort would be allocated to the problem of designing a workable scenario generation concept, rather than invested in the time-consuming development of a fancy graphics interface. The desire was expressed to develop a "core" scenario generation system that could be integrated into existing modeling systems for

which graphics modules would already be available.

To accommodate this request, the level of development of window/menu/mouse features was kept at a relatively low level. Windows, menus and mouse options were developed for high-level control of the system, but most of the keyboard data entry was done in a conventional fashion, using keyboard (rather than mouse) input and text output (rather than graphics text output) to the screen.

There is substantial potential in the current version of the Scenarist for enhancing the user interface through use of the mouse. The most obvious mouse-function extension is to allow the user to reposition items by "clicking" on them and "dragging" them across the screen. In the current version, the user may manually relocate only small-area subordinate units or "point" subordinate items in a unit (minor subunits, platforms, and equipment), not major "area" subordinate units (such as front-line brigades). To relocate major area subordinate units it is currently necessary to erase and redefine the entire unit. With additional development of the user interface, these changes could be made much easier. Such developments would be appropriate after the Scenarist demonstrates the value of its functional capability to generate scenarios. In summary, the allocation of project effort was oriented heavily toward design and implementation of a useful functional capability for automated scenario generation, with minimal

resources expended on the development of the user interface and graphics.

¶ Limited success was realized with respect to the desire to develop a "core" scenario generation system that could be readily integrated into existing systems having a need for an automated scenario generation capability. To a considerable extent, the system-specific input/output and mouse interface functions have been isolated in separate modules, but many of the system modules call system-specific input/output functions. All of the system-specific input/output functions (printer, file, graphics, window, and mouse functions) are coded in Microsoft C, MS-DOS, or Phoenix AT BIOS. If the system is to be converted to for use with a different compiler or operating system (e.g., Borland C++, UNIX), these lower-level input/output functions would have to be recoded.

If future development work is undertaken on the Scenarist, all system-specific input/output calls should be isolated in one or more functions, and all input/output calls in other modules should access non-system-specific (intermediate) input/output functions. This will enable easier porting of the Scenarist to other systems in the future. Greater isolation of the low-level input/output functions could have been accomplished during the Scenarist development effort, but only at the expense of taking time away from attention to major design issues. Demonstration of the validity and utility of the basic Scenarist approach was

considered to be a major issue to be addressed by the project. Without accomplishing this, the Scenarist would be of little use on any computer system, and the issue of porting it to another system would be moot. For this reason, most of the system design effort was allocated to development of a sound conceptual approach and demonstration of a basic functional capability, rather than on facilitating future porting of the Scenarist to other operating systems. Early in the project, consideration was given to developing a UNIX-based version of the Scenarist in parallel with the MS-DOS-based version. Plans to develop this parallel version were dropped, after it was realized how ambitious the basic goal of the project -- development of an automated scenario generation system (on any platform and operating system!) -- was.

2. Design Concept

It was desired that the system would be able to accommodate a high level of detail, to ensure a high degree of face validity both in displaying units and specifying rules. At the same time, it was desired that the system would be parametric in nature, so that functions such as defining units and repositioning units could be accomplished easily and quickly. In the paragraphs that follow, we shall describe some of the design features.

a. Parametric Specification of Military Units

(1). Specification of Unit Structure

¶A design was desired that would provide the user the option of specifying a high level of detail internal to military units. At the same time, it was desired that the system would be able to relocate units quickly and automatically, i.e., to reposition all of a unit's subordinate items automatically if the unit were moved. These goals were accomplished in two ways. First, we developed the concept of a "generic" item. A generic item (unit, platform, or equipment) is an abstraction of a real military unit, platform, or equipment, defined as follows:

1. The echelon level of the item may be any one of thirteen

different echelon levels (side, army, corps, division, brigade, regiment, battalion, company, platoon, section, squad, platform, or equipment). The first 11 echelons refer to units; the last two refer to platforms and equipment.

2. The item has an item type (e.g., a mechanized brigade)

3. The item has a "geographic type," or "geogtype", which

may be one of seven different types (defined in the

next chapter). The seven geographic types are divided

into two classes -- area items and nonarea (point) items.

4. Area items are quadrilaterally shaped.

5. The positions of all of the subordinate items defined for the item are specified in terms of normalized coordinates relative to the item's boundaries. The boundaries are the four sides of the unit square defined by the points (0,0), (1,0), (1,1), and (0,1), with the front being the side from (0,0) to (1,0) and the rear being the side from (1,1) to (0,1).

Next, we defined a "specific item." A specific item has all of the attributes of a generic item, plus the following:

1. The item has an identifying number.
2. A map location is specified for the item (i.e., the four corners of an area item are specified, or the point location for a point item is specified, in real (map) coordinates)
3. The item may have a parent (superior unit).
4. The item may have an objective and mission.

¶ Specific items are defined for all echelon levels except echelons 12 and 13 (platforms and equipment). This implies that the subordinate items of a platform or equipment (i.e., other platforms and equipments) cannot be relocated (their relative positions are the same for all

items of this type) and that no map position is explicitly specified for these items (i.e., their position on a map is determined from their relative position within their parent unit, when the parent unit is placed on the map). The reason for this restriction is to save storage space. Since all platforms and equipments of a specific type are defined "generically," their description is stored only once in the data base. If they could be reconfigured or could be assigned map coordinates as independent units having no parent units (i.e., they were allowed to be defined specifically rather than generically), their specific characteristics would have to be specified and stored, for every platform and equipment. Not only is this not necessary from an application viewpoint, but it would consume a vast amount of storage space (since there are so many platforms and equipments).

In placing a specific unit on a map, it is necessary only to specify the locations of its four corners (or single location, for point items). With this specification, it is relatively easy to relocate items on a map, since only four (or one) of the data items associated with it change. (In addition, of course, the positions of all subordinate items of a moved item must be updated. This is done automatically by the system.)

In order to allow for rapid processing, the system was designed so that the user need specify only those subordinate items of a unit that are affected by the placement rules, or whose

positions may affect the placement of other items. This approach not only saves a vast amount of storage space, but it simplifies the data input process. The user must specify only those items that are required for the application for which the Scenarist is providing scenarios.

At one point in the Scenarist design process, consideration was given to designing the Scenarist to extract needed unit data from Tables of Organization and Equipment (TO&E) data maintained by the US Army Electronic Proving Ground's (USAEPG's) Battlefield Electromagnetic Environments Office (BEEO). This approach was dropped from consideration when difficulties were encountered (delays, cost) in obtaining the TO&E data. If the Scenarist is developed further, additional consideration should be given to extracting unit data from existing TO&E sources. The interface will not be a simple one, however, because of the necessity to specify subordinate units in a form acceptable to the Scenarist (mainly, to determine geographic types for subordinate units and to specify the relative positions of all subordinate units in the canonical configuration).

(2). Definition of a Canonical Unit

¶A key concept in the Scenarist is the concept of a "canonical unit." A canonical unit is a unit whose configuration (relative locations of subordinate items) is in accordance with tactical doctrine, but independent of terrain features, friendly mission, or enemy threat. In using the Scenarist, the user must define a

canonical unit for every unit type to be used by the system. To define a canonical unit, the user specifies the relative locations of all subordinate items of interest in accordance with tactical doctrine, but in the absence of any information about terrain features, friendly mission, or enemy threat. The canonical-unit configuration should represent a reasonable laydown of the unit on a flat, featureless battlefield on which no other units (either friendly or enemy) are located (i.e., there is no information about the terrain, friendly mission or enemy threat).

The Scenarist works essentially by providing the user with means for changing the configuration of units. The starting point is the canonical unit. The process of scenario generation begins by defining one or more specific canonical units and displaying them on a map. The usual procedure for defining a canonical specific unit is to first define a canonical generic unit and then "copy" it to form a specific unit, by providing information about its parent unit (if it has one), its map location (unless it has a parent unit, in which case its map location is already determined from its relative position within the parent unit), its mission and objectives. Having defined one or more canonical specific units, the user may display them on a map.

At this point, the user may reconfigure the units to account for terrain features, friendly mission, and enemy threat. If desired, the user could make "manual" changes to the locations of

the unit's subordinate items. The main point to development of the Scenarist, however, is for the system to reposition the subordinate items automatically in accordance with the user-specified rules. After repositioning the subordinate units in accordance with the rules, the user may review the rule-based placement. If he is not satisfied with the results of the rule-based placement, he may make manual adjustments to the configuration or he may modify the rules and re-execute the rule-based placement process.

b. Multiple Levels of Map Resolution

A major design problem facing the Scenarist development effort was how to conduct the analysis in a fashion that was both "extensive" and "intensive." The term "extensive" refers to the ability to examine large geographic areas, i.e., to handle the deployment of large units (e.g., divisions) over areas measuring scores of kilometers on a side. The term "intensive" refers to the ability to examine a very small area, i.e., to examine the placement of an individual item of equipment on an area measuring a few hundred meters on a side.

¶The problem associated with attempting to develop a system that can handle both extensive and intensive analyses arises because of the desire to use cellular digital mapping data. Cellular digital mapping data are available at resolutions on the order of 10-30 meters. An area measuring 100 km by 100 km at 30-meter resolution contains $(100,000/30) \times (100,000/30)$

= 11,111,111 cells. If it is desired to store three two-byte integer values (e.g., integer representations of terrain-type, elevation, and roads) in each cell, a single map would require 67 million bytes of storage, or 67 megabytes (MB).

While a file of this size could be stored on a microcomputer (having a large hard disk), storage is not the only problem. In order to use the mapping data in the Scenarist, it is necessary to access the mapping data, many times. From a practical viewpoint, it is very desirable, perhaps necessary, to read all of the mapping data needed in an analysis into the computer's direct-access memory -- an area of size from about 640 kilobytes to 4 megabytes for a 386-based microcomputer. It is necessary to have the data in memory because memory access is fast, whereas disk access is slow. Even if the entire 4 MB area were available for map storage, this would accommodate a 30-meter-resolution, 3-variable map of size only $\sqrt{4,000,000/6} \times 30 / 1000 = 24.5$ km on a side.

While a map of this size could accommodate some modest-sized units (e.g., a brigade), there is an even more serious problem associated with the use of high-resolution digital maps for the Scenarist application, viz., processing speed. In generating a scenario, it was anticipated that a substantial number of computations would be required in the search for a suitable location for a subordinate unit, platform, or item of equipment. Some of these computations (e.g., determination of terrain-type) could be done

rapidly, but some are time-consuming (e.g., determination of whether a line-of-sight (LOS) condition exists between two items). The problem is that in the search for a suitable location these types of computations have to be done a very large number of times as alternative locations are examined for suitability. Moreover, they have to be done for a large number of items (subunits, platforms, equipments), and re-done for an item if the positions of other items related to it are changed. Finally, knowledge-based systems are generally slow, and the rules would have to be processed many times simply to find a suitable location for a single item.

Given the nature of the scenario-development problem, it was viewed that the amount of processing to be done could not be handled even on a very fast workstation (much less on a 386-based microcomputer), using a high-resolution map for the entire battle area. The major design issue facing the project, then, was how to design a system that could be both extensive and intensive in nature, i.e., cover a large map area and allow examination of small areas using high-resolution data.

¶The approach adopted was to use maps of varying resolutions, and to "match" the map resolution to the size of the geographic area being considered. Analysis of large units on large maps would be done using maps of low resolution. Analysis of small units, platforms, or equipment would be done using maps of high resolution. With this approach, decisions about

repositioning items is done using geographic-feature data at a level of resolution that is appropriately related to the size of the unit. The process would begin by examining large-scale units on a large, low-resolution map. The subordinate items of the units would be positioned in accordance with the rules. Then, the user would switch to a smaller map of higher resolution and repeat the process. This process would be repeated until the user had used the highest-resolution map available.

With regard to matching the map resolution to the unit size, the map resolution should be such that the unit "covers" a "reasonable" number of map cells, e.g., the unit's side is about 10-15 map cell widths in length. In order for the user to be able to see the entire unit in the map context, this implies that the map should be somewhat larger than 10-15 cells on a side. It was decided that all maps used in Scenarist processing would be 32 cells on a side. If a map of the appropriate resolution was available having more than 32 cells on a side, a 32 x 32 "window" would be extracted from it. If a map of the appropriate resolution was available having fewer than 32 cells, then the available portion of the map would be displayed on the screen.

The system for repositioning subordinate items in a unit was implemented in the normalized coordinate system relative to the unit boundaries. In the Scenarist processing, a rectangular gridwork is defined over a unit, in a normalized coordinate system in which the unit

is represented as a square with corners at (0,0), (1,0), (1,1), and (0,1). The unit front extends from (0,0) to (1,0). The grid cell size is determined such that, in the map coordinate system, the cell size is equal to the cell size of the highest-resolution map (of the three map types terrain type, elevation, and roads). The process of finding a suitable location for an item that is not suitably located (i.e., whose location does not satisfy all of the rules applicable to that type of item) involves examining the cells (of the unit's cell grid) near the current location, in the search for a suitable cell. If one is found within a specified radius, the item is relocated to that cell.

¶The Scenarist has been designed to accept maps of differing resolutions, i.e., the terrain-type, elevation, and road maps do not have to be of the same resolution. In a particular analysis, however, maps of similar resolutions should be used if they are available. The amount of processing is governed by the resolution of the highest-resolution map of the three maps. If one of the two other maps is replaced by a map of lower resolution, the processing time is not reduced. Since the processing time is the same whether the higher-resolution or the lower-resolution map is used, there is no point in ignoring the finer details of the higher-resolution map -- the analysis loses some validity at no saving in processing speed.

With the adopted approach, the analysis performed uses only the data available in the 32 x 32 (or smaller) maps extracted from larger map files and placed on the screen. (Note: if the user zooms on a map, the system still uses the full 32 x 32 map for the analysis.) Rule processing is performed only for subitems located on the maps that are currently in memory. If in the processing of constraints reference is made to a point outside of the 32 x 32 map area, no data are available for that point (even though data for that point may have been available in the map file from which the 32 x 32 map was extracted). While this presents no problem in the processing of rules that involve "local" data (in the vicinity of an item, and on the 32 x 32 map), there is no point in processing rules that involve reference to points outside of the 32 x 32 map area. In particular, LOS from an item on the map to an item off the map cannot be determined. For this reason, it is important to process rules that simultaneously involve all of the subordinate items of a unit on a 32 x 32 map that covers the entire unit (e.g., rules that involve LOS among all subordinate items). Processing of rules that involve only local data around a single item may be done on a 32 x 32 map that includes only that single item. (Additional discussion of this point will be made later, when the concepts of "local constraints" and "global constraints" is presented.)

In summary, with respect to map resolution, the design possessed the following features: (1) ability to display maps of a scale sufficiently large to contain several divisions and of a scale

sufficiently small to show the terrain features in the immediate vicinity of an individual item of equipment; (2) ability to accommodate maps of differing resolutions for each data topic of interest (terrain type, elevation, and roads) and to use maps of differing resolutions for the three data topics in the same analysis; (3) map resolution matched to unit size; (4) ability to switch easily to maps of different resolution; (5) ability to zoom on a map.

Note that the preceding discussion of issues of multiple map resolutions relate to the analysis of map data (i.e., the definition of factors from map data for use in rules), not to the display of background maps. Concerning the use of background maps that are not to be used in the rule-based analysis, it is feasible to store high-resolution large-area map images (stored in files or CD-ROM laser disks).

¶Some additional remarks will be made concerning the sequence of operations in using the Scenarist system, relative to the issue of maps of varying resolutions. The user should begin processing on a map of sufficiently large scale that the unit being processed fits entirely on the map. It is important that the unit fit entirely on the map in order for the global rules to have a significant effect on item positioning. (If part of the unit lies off the map, subitems in that part of the unit will correspond to "no data" for data derived from the map. Any global rules referring to those items in that part will be processed using "no-data" codes.) Once the user

has made a run in which the unit fits entirely on the map, the global rules will have applied.

After completing rule processing for a map of sufficiently large scale that the unit fits entirely on the map, the user may proceed to conduct rule processing on a map of higher resolution -- e.g., on a map having a scale half the size of the first map. The user may proceed to process rules on maps of successively smaller scales, to as small a scale as is appropriate for the application at hand. At very small scales, it may be the case that only a single subitem is on the map. In this case, only the local rules will apply, since "no-data" codes will be returned for off-the-map references. This situation is acceptable, however, since the global rules will already have been applied when the user conducted rule processing with a higher-resolution map. In fact, this situation is desirable, since it is in general inefficient to process global rules using very-high-resolution data. In effect, the Scenarist enables the user to "match" the map resolution to the "scope" of the rules.

As the user moves to maps of higher resolution, the processing time required to extract the 32 x 32 maps from the map files becomes longer. The rule processing time does not increase significantly, however, because the processing is done only for subitems on the map, and the number of subitems on the map decreases as the map scale decreases.

c. Allocation of Processing between CLIPS KBS and Developmental C-Language Functions

(1). Rationale for Allocation

¶The CLIPS knowledge-based system (KBS) is designed to store and execute rules, i.e., for "logical" processing. It is not intended for numerical processing or graphics. In the system design, system requirements related to storage and processing of rules were allocated to CLIPS, and system requirements related to numerical and graphics processing were allocated to the developmental software. In spite of the general intent to effect the preceding allocation, situations arise in which a function could reasonably be implemented either in CLIPS or in developmental software.

For example, suppose that a rule specifies that an item may not be located in a lake, and that a certain item is in fact located in a lake (when the canonical laydown is placed on a map). This rule is said to "fire." Now, an action should be taken -- a search should be initiated for a suitable location out of the lake. Under one design approach, this search could be implemented by a search function coded in C in the developmental software of the Scenarist. An alternative design approach, however, would be to design a set of rules that could be implemented in CLIPS to identify a new location for the item.

In a situation like this, the Scenarist design approach would be to implement the search function in the developmental software, not in

CLIPS. In general, CLIPS was used to implement basic tactical rules, not to accomplish "preprocessing" that could be readily programmed in C, or to implement "actions" that resulted from the firing of a rule. There were several reasons for this design decision. First, it was desired that the user be required to identify rules concerned with tactical doctrine, not with implementing search strategies. Second, because of the reputation of KBSs for slow speed, it was desired to minimize the amount of processing done in the CLIPS KBS. Finally, prior to the Scenarist development project we had no previous experience with CLIPS, and it seemed prudent (i.e., risk reducing) to minimize the amount of processing done in CLIPS.

A conscious attempt was made to allocate to CLIPS the task of storing rules and performing logical operations that could conceivably take advantage of an inference engine (an inference engine determines the order in which rules are processed). Numerical processing (e.g., determining accessibility, or searching for a suitable location) that had little to do with tactical doctrine or little need for the special inferential capabilities of a KBS were not allocated to CLIPS.

(2). Preprocessing and Action Functions

(a). Preprocessing

¶Tactical doctrine rules are formulated in terms of certain concepts, or variables, or factors, that can be determined from observation of the

battlefield situation. The process of formulating rules for use in the Scenarist involves identifying, for every concept or variable or factor in a tactical rule, closely related variables whose values may be determined from the variables maintained in the Scenarist system. In some cases this is straightforward. For example, a terrain-type map specifies whether a certain cell falls in a body of water. A rule that specifies that a unit may not be located in a body of water is easy to implement. A variable, WATER, could be defined whose value is 1 if the terrain type of the cell in which the item is located is water, and 0 otherwise. The rule could be specified as: "If WATER = 1, location is unsuitable." Alternatively, a variable TERRAIN could be defined with a number of different values, each value representing a different type of terrain. If that the value "5" corresponds to water, the rule could be specified as: "If TERRAIN = 5, location is unsuitable."

As additional examples of factors used in rules, it may be necessary to determine the distance of an item from the division's front, or to determine whether a line-of-sight condition exists between two units. In these cases, C-language functions must be coded to determine these quantities from the unit definition and the map. These quantities are computed on an "as-needed" basis, in the consideration of the suitability of various locations for the placement of an item.

In some cases, the "operationalization" of a tactical concept may be somewhat difficult. For

example, in the TRAILBLAZER application, the concept of "accessibility" comes into play. In order for a TRAILBLAZER unit to be positioned at a certain location, the location must be accessible. The TRAILBLAZER unit cannot negotiate grades of greater than 30 degrees. In order to determine the suitability of a location for a TRAILBLAZER unit, processing must be done to determine whether the location is on a road or can be reached by road from the division rear area without exceeding a 30 degree grade. The determination of the accessibility of a location requires a considerable amount of processing, and it is efficient to determine the accessibility for every cell of a map prior to beginning rule processing, rather than to attempt to determine it for specific locations on an as-needed basis (as was done in the case of the WATER rule discussed above). In other words, it is efficient to conduct the "preprocessing" required to determine accessibility for all cell locations of a map, and store the accessibility values in the map for rapid access during rule processing.

In the Scenarist, all of the functions required to compute the factors needed in rules (whether computed "on-line" as in the case of terrain type, distance to FEBA, or LOS, or through the use of preprocessing) are coded in various C-language functions. The current version of C contains a number of these functions, but future applications will undoubtedly require additional functions. These must be coded by the user.

¶(b). Actions

i. Actions Resulting from Rule Processing (Location Suitability Determination)

During the development of the Scenarist, the rules were formulated around a concept in which the location of an item was either suitable or unsuitable. That is, "suitability" was implemented as a discrete concept, rather than a continuous one. The action of the CLIPS system after processing the rules was to return a "suitability" value, that was either zero (location unsuitable) or one (location suitable). The "actions" to be taken in response to the rule processing were in turn formulated in terms of this discrete concept of locational suitability.

There is no reason why suitability cannot be implemented in terms of more than two categories (unsuitable and suitable), or as a continuous concept (e.g., a decimal value between 0 and 1). The user need simply formulate a rule that determines such a value based on the factors provided to it, and pass this value back to the Scenarist system for consideration in the action functions.

During the initial conceptualization of the Scenarist design, consideration was given to a probabilistic approach to item location. Under that concept, a "suitability surface" would have been determined, that defined the locational suitability for a particular type of item over the area occupied by a unit. This approach was

rejected in favor of the "canonical-unit" approach, however, because it was not clear how to handle joint probabilistic interrelationships among units (i.e., how to specify the joint probability distribution of the locations of units). The current version of the Scenarist is intended to produce a single "deterministic" laydown, rather than a probability sample of several laydowns. There is, however, no reason why the Scenarist cannot be extended to produce scenario samples.

There are at least two obvious ways to implement probability sampling. First, the user could define a set of several canonical configurations for a specific type of unit and specify (subjective) probabilities for each member of the set. Then, whenever a unit of a specific type was needed, one could be selected by probability sampling. Another approach is to specify a probability scheme for selecting a location, in the action function. This approach would be best suited for introducing a small amount of "local" variation in the locations of units, whereas the sampling of canonical units would be appropriate for introducing a large amount of stochastic variation.

¶In the original Scenarist proposal, it was proposed to generate, by probability sampling (simulation), a large number of scenarios having certain general characteristics. These characteristics could be parameters (such as the military organization structure, or the geographic land type) of sampling distributions used in the generation of the scenarios, or they

could be other observed characteristics observed from the generated scenario (e.g., terrain roughness, average distance between emitters). It was suggested to store the generated scenarios in a data base and develop a scheme for sampling scenarios having specified characteristics from the data base (e.g., using controlled selection or some other probability sampling procedure).

As the project got under way, it became apparent that all of the project resources would be required simply to develop a deterministic version of the Scenarist. No resources were available to determine a stochastic version, or to develop a scenario sampling scheme such as was discussed in the proposal. If future experience with the Scenarist is successful, consideration could be given to these extensions. Such extensions would be quite useful for test and evaluation applications (where the availability of a probability sample of scenarios would broaden the scope of inference of tests (based on a sample of scenarios rather than a single scenario) or training applications (where a large number of different scenarios would be useful to avoid trainees' learning the characteristics of a single scenario)).

ii. Action Functions Implemented after CLIPS Processing

After one or more rules have "fired," CLIPS specifies an action to be taken. As discussed above, in the current version of the Scenarist the action resulting from the CLIPS rule processing is simply a determination of the

suitability of a location. Once the suitability value has been determined, the Scenarist executes an "action function" that implements a search for a suitable location. In the current version of the Scenarist, the action functions have been kept simple. Rules are classified in two types -- those dealing with "local" constraints, and those dealing with "global" constraints. A local constraint (or local rule) is a rule (about the suitability of the location of an item) whose factors depend only on the item's attributes and the map data. A global constraint (or global rule) is a rule (about the suitability of an item) whose factors involve other items. In the Scenarist processing all local rules are processed, and then all rules (local and global) are processed.

¶During the processing of a particular class of rules (local or global), a particular type of "action" is taken. The actions made in response to local rule processing are to initiate, continue, or terminate a "spiral" search of cells in the neighborhood of an unsuitably located item, in the search for a suitable location. All of the unit's subordinate items (to which rules apply) are processed, in a single cycle, in sequence. Once a unit is suitably placed in accordance with the local rules, its suitability will not change with respect to the local rules if the locations of other items are changed (by the definition of a local rule).

During the processing of global rules, an appropriate action (defined by the user) is taken after processing each item. All of the unit's

subordinate items are processed. Then they are all reprocessed some additional number of times. The reason for the reprocessing is that a global rule may be satisfied at one time for a particular unit but violated at a later time if some other unit is moved. This could not happen for local rules, by definition. The actions specified for global rules should be such that there is a reasonable prospect that the global rules will be satisfied after the actions have been applied several times to all of the subordinate items of the unit. For example, in the TRAILBLAZER application, the action in response to failure of the global rule requiring certain LOS conditions among TRAILBLAZER units was to move the unit to a neighboring cell of higher (or equal) elevation.

C. Data Assembly, File Preparation, and Program Preparation

1. Map Data

There are two kinds of maps used in the Scenarist processing -- cellular maps and vector maps. Cellular maps are defined as a matrix of numbers defined for each cell of a rectangular grid over a rectangular geographic area. These maps contain data that is used by the Scenarist in its determination of a suitable location for a unit, platform, or equipment. (Since the cellular maps are defined in terms of numerical ("digital") data, these maps are also referred to as digital maps. The term "digital" is ambiguous. It is also used to refer to vector and raster (image) maps. Because of the

ambiguity of the term "digital," we will generally use the more specific terms "cellular map" and "vector map" in this report.) The current version of the Scenarist accepts three types of cellular maps, one containing terrain-type data, one containing elevation data, and one containing road data.

¶The second type of maps used by the Scenarist are called vector maps. These maps define a map background, to make the map display on the system video terminal screen more comprehensible to the system user. These maps are defined in terms of geographic objects -- polygons, splines (a series of straight-line segments connected end to end), and points. There are three types of vector maps used in the Scenarist --area-object maps, linear-object maps, and point-object maps. An area-object map specifies a number of polygons to be drawn on the screen. These polygons represent area objects, such as lakes and towns. A linear-object map specifies a number of splines, such as rivers, roads, and political boundaries. A point-object map specifies a number of point locations, such as a location reference point for a small village or a marshalling point. All of the objects of the vector maps may have labels which may be printed on the map at the user's discretion.

The vector maps are used simply to provide a pleasant map background and frame of reference for the user. Their use is optional. The Scenarist model is not designed to access the data of the vector maps in its analytical processing.

a. Cellular Map Data

The cellular mapping data used by the Scenarist in its rule-based processing fall generally into two categories -- elevation data and other geographic features. The Defense Mapping Agency provides these data in its DTED and DFAD data bases. In the design of the Scenarist, it was necessary to decide what types of mapping data would be used. Because of the design decision to keep at most 32 x 32 cell maps in memory, the system could accommodate a number of maps. It was possible to store all data in integer format, so that each data item describing a cell used only two bytes of memory. Hence, each map consumes only $32 \times 32 \times 2 = 2048$ bytes of memory.

Although available memory would have allowed for storage of a number of map data topics, it was necessary to decide on what those data topics were, in order to specify the specific variables and functions that would be the factors used in the rules. Initially, we decided on a "minimal" set of maps -- just two maps, terrain type and elevation. In the course of the TRAILBLAZER test, however, accessibility became a key factor, and it was decided to add a road availability map as well. The use of the road availability map is optional. If a user has an application that does not need road data, the system will run using only elevation and terrain type (both of these, however, are required).

In addition to the two or three maps accepted as data input, the user may generate other maps for use in rule processing. For example, in the TRAILBLAZER demonstration, it was decided to create a "TRAILBLAZER accessibility map," which specified the accessibility of a TRAILBLAZER unit to each cell of a map.

¶In general, the user may generate as many additional maps as are desired for a particular application. The maps are stored in two arrays (one for discrete (categorical) data such as terrain type and one for continuous data such as elevation), and the dimensions of these arrays may easily be changed (from the current values of three discrete-data maps and one continuous-data map to larger numbers).

The additional maps may be read from map files (as was the case with the road availability data) or they may be generated from existing maps (as was the case for the TRAILBLAZER accessibility map, which was derived from data in the elevation, road availability, and terrain-type maps). The program code has to be developed to generate additional maps or to read in additional maps from map files. However this is done, the data in these maps may then be accessed for use in the rules.

Early in the Scenarist development project, it was planned to use DMA mapping data directly in the system. We were unable, however, to obtain any DMA mapping data on a medium accessible by the firm's 386 computers (3-1/2" diskettes, 1/4" tape, compact laser disks (CD-ROM), or

Bernoulli-Box cartridges). Without sample data, we were unable to develop this capability. Instead, it was agreed with the CECOM project officer to use digital mapping data sample data sets that were provided with the US Army's GRASS geographic information system (GIS) for development and testing. These data are similar in format to DMA mapping data, but they are not DMA mapping data. The GRASS Spearfish, SD, mapping data included about a dozen cellular data topics (including the elevation, terrain type, and roads data topics used by the Scenarist) and two vector data topics (roads and streams).

¶The format of the cellular map files is relatively simple. The file contains a "header" -- a number of lines that identify the map, the data type (discrete or continuous) the number of data categories (for discrete data), the Universal Transverse Mercator (UTM) zone and band, the coordinates of the map top left corner (in UTM coordinates), the cell width, and the number rows and columns. The Scenarist system does not include software for formatting DMA or other map data files in the format required by the Scenarist. During the Scenarist development, some cellular map files were manually generated using the Microsoft "edlin" line editor, and some were extracted from the GRASS sample map files. The manually generated maps were developed simply by obtaining a map for the area of interest (the Beqaa Valley region of Lebanon), drawing a rectangular grid of gridsize 2000 meters over the map, and observing the terrain type from the map. Hypothetical elevation data were used; the elevation data

correlated with the gross terrain features (ocean, plains, mountains), but were not true or accurate values. Hypothetical elevation data were used simply to avoid expending resources in locating correct elevation data, which was not readily available and was not necessary for testing purposes.

Consideration was given to developing some software to assist a user in generating a cellular map file. This software would have been a program to request the user to provide the required header data and then to provide the value for each cell of the map. It was decided not to allocate any resources to the development of this software. For a one-time application, the (labor) cost of using the edlin line editor was less than the cost of developing the software. Also, such software would probably be little used, since the primary intention is to use the Scenarist to access large-scale cell maps that can be extracted from existing sources (e.g., DMA maps).

Some software was developed, however, to form lower-resolution maps from higher-resolution maps. The process of forming a lower-resolution map from a higher-resolution map was referred to in the project either as map "compression" or map "aggregation." This software can aggregate either discrete-data or continuous-data maps. It operates by forming a single cell from a square block of neighboring cells. The value of the single cell is estimated from the values of the block of cells, using standard statistical

procedures (e.g., computation of a mean, mode, or "no data" code).

The data input required of the user by the program (s03xcomp.c) to create a lower-resolution map from a higher-resolution map is the following:

1. The name of the higher-resolution map file (i.e., the input file)
2. The name of the lower-resolution map file (i.e., the output file)
3. Whether it is desired to compress a discrete-variable (terrain-type, road) map or a continuous-variable map (elevation). (Note: the descriptors "discrete" and "continuous" here refer to the measurement nature of the variable, not to the format in which the data are stored in the file. In the current version of the Scenarist, all maps are stored in integer format.)
4. If a discrete-variable map is to be compressed:
 - a. The number of adjacent cells to aggregate (i.e., the "compression factor")
 - b. Whether the code "0" (zero) in the file indicates "no data" or is an acceptable data value
 - c. Whether it is desired to compute a mode or an indicator-variable value from a block of cells. (An indicator variable indicates the presence or absence of specified attribute values: 1 for present, 2 for not present.) If an indicator variable is to be computed:
 - (1). The number, nvals, of attribute values to check for

(2). The nvals attribute values to check for

The basic approach of the Scenarist development project was to put effort into the development of the basic system functions required to generate scenarios, not into the development of software to process map data. As noted earlier, the major issue to be decided at the completion of the Scenarist development project is the validity and utility of the generated scenarios. If the system is accepted from these points of view, it then makes sense to allocate resources to developing software to make the system easier to use.

b. Vector Map Data

The Scenarist uses vector map data to draw background maps. Although cellular maps are more convenient for analysis, vector maps are better for visual presentation. They have a "smooth," professional, easy-to-view look, rather than the "blocky" or "grainy" appearance of a cellular map.

In the initial design of the Scenarist, it was decided to use an object-oriented specification for the vector maps. Three types of geographic map objects were defined -- area objects (such as lakes, towns, land-type areas), linear objects (rivers, roads, unclosed portions of geographic boundaries), and point objects (reference points, such as peaks or small villages). With the object-oriented specification, the geographic features (objects

with attributes) specified in the vector maps could have been used in the analysis. It was decided, however, not to use the vector-map objects in the analysis. The reason for this decision was simply that project resources were quite limited, and vector data are somewhat more difficult to work with in analytical problems than cellular data. A future extension of the Scenarist might reconsider using object-oriented vector map data in the analysis (in addition to the cellular data).

In the early development of the Scenarist, the area-object, linear-object, and point-object vector map files were developed by hand for the Beqaa Valley region, using the Microsoft edlin line editor. These maps contained a wide variety of geographic objects (e.g., cities, towns, roads, rivers, political boundaries, place names). In the TRAILBLAZER test, only two types of vector data were available for the Spearfish, SD, geographic area used in the test -- roads and streams. During the test, we displayed only roads on the map.

¶c. Image Data

During the Scenarist development, we became aware that DMA provides image ("raster") mapping data on 3-1/2" high-density diskettes. If future development of the Scenarist occurs, it would be desirable to create a capability to use these diskettes to generate the Scenarist background maps.

As was the case with cellular maps, consideration was given to developing software to assist the user in the manual preparation of the vector map files used by the Scenarist. No resources were allocated to the development of this software, for the same reasons (given above) that no software was developed to assist the user in the manual preparation of cellular maps.

d. Software for Preparing Scenarist Map Files from GRASS Map Files

It was not a contract requirement or project objective to develop general-purpose software for assisting the user in formatting map data from various sources for input into the Scenarist system. However, in order for the Scenarist to work, map data had to be available in some format. Had sample DMA data been available on diskette or CD-ROM or Bernoulli-Box cartridges (the media available to Vista), the Scenarist would have been designed to accept such data, or software would have been to "preprocess" this data into a format acceptable by the Scenarist.

As it developed, the initial prototype of the Scenarist was developed using manually prepared map data. The data were put in a file in a particular format using the Microsoft edlin line editor. Later, when sample map data were extracted from the GRASS system, they were placed in files in this same format.

The format of the Scenarist cellular map files included a header that included the following information:

1. The file name
2. The map name
3. The data topic
4. The data source
5. The file creation date
6. The date of the last change to the file
7. If the data are continuous, the name of the unit of measurement (e.g., meters). If the data are discrete, the number of data categories and the names of the data categories.
8. The UTM zone and band
9. The location of the map top left corner, in UTM coordinates
- ¶10. The cell width
11. The number of rows and columns
12. The row format (in FORTRAN, for information only)

The data input required by the program (WIN7.FOR) that prepared Scenarist cellular map files from GRASS cellular map files required the following input from the user:

1. The data topic (elevation, soils, slope, aspect, road, streams, landuse, quad, geology, vegetation, railroads, or terrain)
2. The east-west and north-south coordinates of the window top left point. (The cellular map extracted from the GRASS map file and converted to Scenarist format was referred to as a "window." This window was square, i.e., the number of east-west cells ("columns") was equal to the number of north-south cells ("rows").)
3. The width of the square window in meters

4. The name of the Scenarist map file into which the data are to be written

The WIN7.FOR program prepares a file header in the required Scenarist format, and writes both the header and the data into the specified file.

The data input required by the program (WINL7.FOR) that prepared the Scenarist vector map files from GRASS vector map files required the following input from the user:

1. The data topic (streams or roads)
2. The east-west and north-south coordinates of the window top left point
3. The width of the square window in meters
4. The name of the Scenarist map file into which the data are to be written

The WINL7.FOR program prepares a map file header that contains the following information:

1. File name
2. Map name
3. Data topic
4. Data source
5. File initial creation date
6. Date of last file change
7. UTM zone and band
8. Map top left corner, in UTM coordinates

The WINL7.FOR program writes the map header and the vector data (in the format required by the Scenarist program) into the specified Scenarist map file. Since no file header is used by the Scenarist for vector map files, the header should be removed (e.g., by using the edlin line editor) before using the vector map file.

¶Some comments are in order concerning the non-use of headers in the Scenarist vector map files. Headers are used in cellular-data map files. The main reason for this is that the descriptive information about a cellular map is printed on the screen to assist the user in changing map files and selecting map location points (the places from which 32-cell by 32-cell "submaps" will be selected from the map file). While the user will generally use several different-resolution cellular map files in a particular Scenarist application, he will typically use at most a single area-object file, linear-object file, and point-object file. For this reason, there is no capability in the Scenarist for changing vector map files, and no need for header information to assist the user in changing such files.

The user of headers causes problems in files that are intended to store large numbers of similar-format records. The use of headers in cellular map files does not cause problem in this regard because a cellular map file does not contain a large number of similar records. Instead, it contains a single array defining a map (following the header). On the other hand, vector map files may contain large numbers of records (each one associated with a geographic object). These files may be read many times during the course of a Scenarist program run. The user may modify them frequently (at least initially), and it is likely that software will be developed to generate and modify them interactively.

In other words, a vector map file represents a data base filled with a large number of similar records. Including a header in such a file causes problems in data access, because the file header is of a different structure from all of the following records. With the header present, the process of accessing the other records is more difficult than if all of the records in the file are of the same type. All of the data records stored in a vector map file contain the same sequence of variables. These records are not necessarily of the same size, however, because the files may be prepared in "free format" using a line editor, in which case the user may input spaces, carriage returns, and line feeds between each variable. Although the data records of a vector map file are currently not necessarily all of the same size, they will be if software (such as WINL7.FOR) is used to generate these files (from user input over a keyboard). At that time, no header would be used (just as no header is used for the Scenarist unit files, which are generated by the Scenarist program rather than manually created by the user using a line editor). Because the use of headers would be dropped at that time, it was decided not to include them at all (to minimize the need for later modifications to the program code).

¶The major disadvantage of this (no-header) approach is that without descriptive headers, the vector map files are not "self-documenting." Of course, that is true for all of the other Scenarist data files, such as the unit files. Since those files are not self-documenting, the

user must maintain a log identifying the contents of each file. The process of keeping track of the various files is facilitated by using the file naming convention described later in this report (and also in the Scenarist program documentation and the Test Report). Even though the cellular map files contain descriptive headers, the user should maintain a log even for those files, since the file contents are not known without reading the file.

The programs WIN7.FOR and WINL7.FOR were written in FORTRAN rather than C since the staff member assigned to the task of developing these programs was much more familiar with FORTRAN than C. Since this task was the only project task assigned to that individual, it was considered more efficient to use FORTRAN than C for this task. The task of identifying map source data and assembling the data into a map file in the Scenarist map file format is not addressed by the Scenarist program. This task will typically be accomplished by software that the user already possesses for processing map data, or by new software that he will develop in whatever language he is comfortable with. That language may or may not be C. Except for the esthetic aspect of doing all Scenarist project coding in C, there was no overriding reason to use C rather than FORTRAN in the task of preparing Scenarist map files from the GRASS map files.

2. Unit Data

a. Unit Structural Specification

The Scenarist embodies a model-based approach to scenario generation. The foundation for this approach is a parametric representation of a military unit. The term "parametric representation" means that all military units are have a common structure, that is defined in terms of a relatively small number of variables, called parameters. The parameters specify the number and types of subordinate items in the unit. The subordinate items are either other military units, platforms, or equipments.

The following discussion of unit structural specification provides some details on the way in which units are defined in the Scenarist model. This information is actually part of the detailed design, but is included here to facilitate the discussion of what unit data are needed by the Scenarist.

¶The term "military item" refers to a military unit, platform, or equipment. Every military item has an "echelon," which is one of thirteen levels, numbered 1-13, respectively:

1. side
2. army
3. corps
4. division
5. brigade (or brigade group)
6. regiment (or group)
7. battalion (or squadron)
8. company (or battery or group)
9. platoon (or detachment)
10. section
11. squad
12. platform

13. equipment.

These echelon levels correspond to standard military usage, except for the association of an echelon level with side, platform, and equipment. The echelon level of a subordinate item must be of lower echelon level than its parent unit. (An item's parent unit is the superior unit to which it belongs, i.e., of which it is a subordinate unit.)

"Side" is considered an echelon level for convenience in coding and processing units. "Platforms" includes platforms (in the usual military sense), systems, and equipment suites, which may be subdivided into component elements. "Equipments" includes individual equipment items, which are not subdivided further.

Every unit has a "type" -- a positive integer descriptor (1, 2, 3, ..). The type is intended to distinguish units of different structure, but of the same echelon (e.g., a mechanized infantry battalion vs. a tank battalion). In the process of defining a unit, the user would typically define a generic unit of a particular type, and make "copies" of it to save time when defining specific units of that type. (Note: The value "0" may be used for unit type. It is recommended, however, that the value zero be used only for generic units. The reason for this recommendation is that it is planned to develop a capability for displaying a generic unit of type zero on the screen as a guide to the user when he is defining a unit of the same echelon structure.)

Every specific unit has a "number," used to distinguish it from other specific units of the same side, echelon, and type. The number is a positive integer (1, 2, 3,...). A specific unit may also have a user-supplied identification number ("idno"). The idno may be used by the user to retrieve units for display. It is not used for any other purpose by the Scenarist.

Generic units do not have numbers or idno's.

¶Every unit has a code and a parentcode, used by the Scenarist to keep track of units and their relationship to other units (parent unit and subordinate units). The code (and the parentcode) is an 11-component vector containing the unit numbers for each of the echelon levels of the units to which the unit is subordinate, its own unit number, and zeros for lower echelon levels. That is, the code is the vector (side number, army number, corps number, division number, brigade number, regiment number, battalion number, company/battery number, platoon number, section number, squad number), where all vector components corresponding to echelon levels below the unit's echelon level are zero. Some echelon levels may not be present. For example, the code of a division of side 1 (BLUE), army 1, might be (1,1,1,0,0,0,0,0,0,0,0). The code of a two regiments in this division might be (1,1,1,1,0,0,0,0,0,0,0) and (1,1,1,2,0,0,0,0,0,0,0). The code of a section in the second brigade might be (1,1,1,2,0,0,0,0,0,1,0).

Every subordinate item of a unit has a "geographic type." There are seven geographic types, or "geogtypes", numbered 1-7, respectively:

1. on-line (front) subunits
2. reserve (rear) subunits
3. all-area subunits
4. major subarea subunits
5. large minor subarea subunits
6. small minor subarea subunits
7. point (non-area) subitems (platforms, equipments).

An item's geogtype indicates the geographic type of a subunit, with respect to the specification of the unit's geographic layout by the user when defining a unit.

The definitions of the geogtypes and some other related quantities are given below. All area units are considered to occupy quadrilaterally shaped geographic areas. No point subitems were involved in the developmental tests of the Scenarist, and so the attributes of these items were left undefined.

Figures 3-8 present graphical illustrations of the various unit geographic types.

Unit corners. For specific units, the map coordinates of the four corners of the unit (i.e., the four corners of the quadrilaterally shaped area defining the geographic area associated with the unit). For generic units, the corners are (0,0), (1,0), (1,1), and (0,1).

Unit boundaries. The four sides of the quadrilateral defining the unit's area. The unit is oriented in a particular direction, with one boundary called the front, its opposite boundary called the rear, and the other two boundaries called the flanks or sides.

Front/rear boundary points. Two points, one on each side (flank) of the unit, that define a line separating the unit into two (quadrilaterally shaped) areas, called the front and rear areas of a unit.

On-line (front) subordinate unit (geogtype 1). A subordinate unit ("subunit") occupying a (quadrilaterally shaped) slice of the unit's front area.

Front-unit boundary points. Two points, one on the unit's front and one on the front/rear boundary, that define the side of a front subunit.

Reserve (rear) subordinate unit (geogtype 2). A subunit occupying a slice of the unit's rear area.

Rear-unit boundary points. Two points, one on the unit's rear and one on the front/rear boundary, that define the side of a rear subunit.

Domain. The area covered by a unit (or subunit).

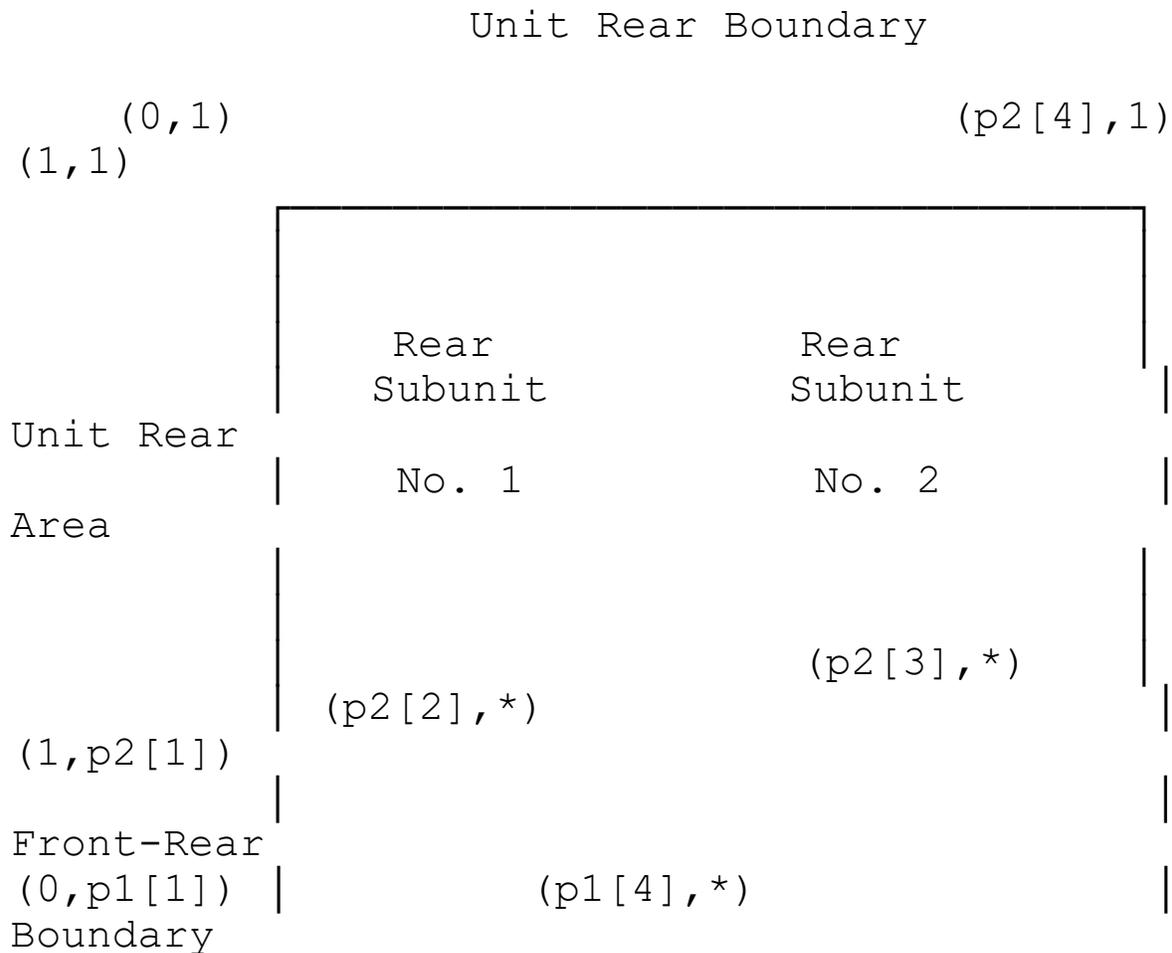
All-area subordinate unit (geogtype 3). A subunit whose domain is the entire unit area

(i.e., a subunit that "covers" the entire unit area).

Symbol location point. The point at which a subunit's symbol will be displayed.

Major subarea subordinate unit (geogtype 4). A subunit whose domain is a quadrilateral of arbitrary location and orientation within the unit area.

Figure 3. Illustration of Unit Geographic Types 1 and 2 (Front and Rear Subunits)



Note: All parameter values represent coordinates in the unit coordinate system. These parameters are converted to map coordinates by the Scenarist program.

Figure 3 (cont.). Illustration of Unit Geographic Types 1 and 2 (Front and Rear Subunits)

Front/Rear Boundary Points: Two points, one on each side (flank) of the unit, that define the front-rear boundary. In the illustration, the points $(0, p1[1])$ and $(1, p2[1])$. The user inputs the values $p1[1]$ and $p2[1]$. (Note: the indices $[1], [2], \dots$, in the illustration are used simply to distinguish the several sets of $(p1, p2)$ points that are requested by the program in the process of defining a unit.)

Front-Unit Boundary Points: Two points, one on the parent-unit's front boundary and one on the front/rear boundary, that define the side of a front subunit. This side is a common boundary between two front subunits. The number of sets of these two points that must be specified by the user is one less than the number of front subunits. In the example, there are two sets of such points, $((p1[2], 0), (p2[2], *))$ and $((p1[3], 0), (p2[3], *))$ (where $*$ denotes an unspecified y-coordinate on the front-rear boundary). The user inputs the two sets of numbers $(p1[2], p2[2])$ and $(p1[3], p2[3])$.

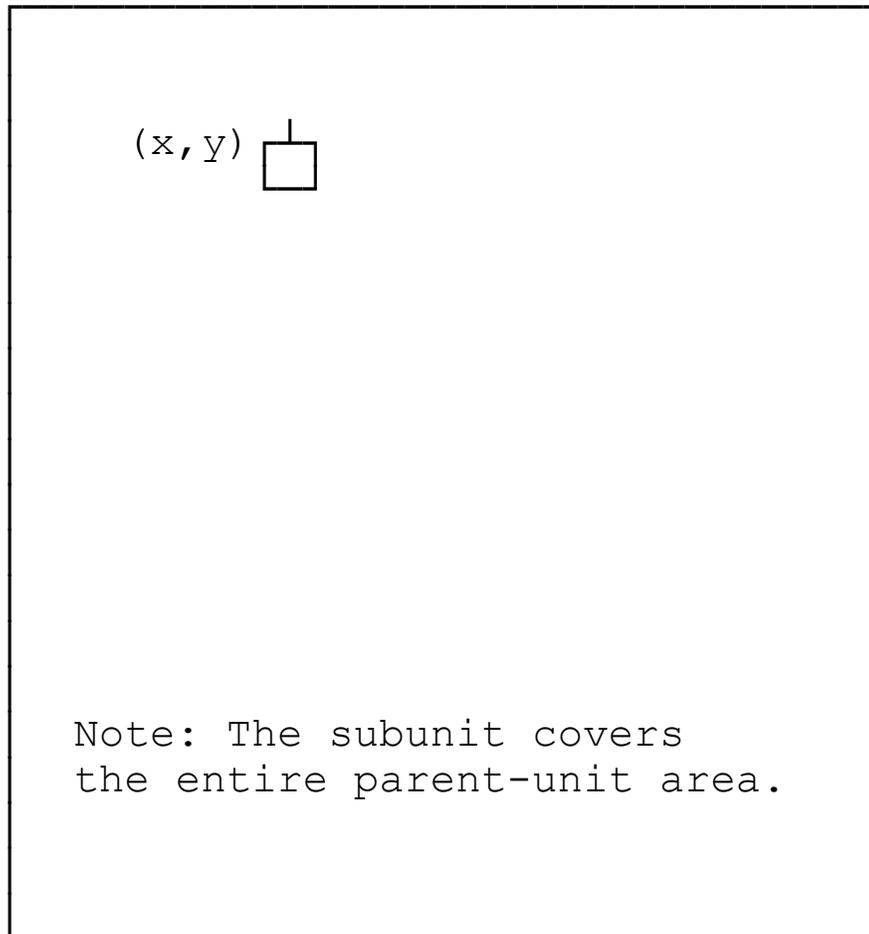
Rear-Unit Boundary Points: Two points, one on the parent-unit's front/rear boundary and one on the rear boundary, that define the side of a rear

subunit. This side is a common boundary between two rear subunits. The number of sets of these two points that must be specified by the user is one less than the number of rear subunits. In the example, there is one set of such points, $((p1[4], *), (p2[4], 1))$.

Figure 4. Illustration of Unit Geographic Type 3 (All-Area Subunit)

Unit Rear Boundary

$(0, 1)$
 $(1, 1)$



(0,0)
(1,0)



Unit Front Boundary

Definitions:

Geographic Type 3: An all-area subordinate unit. A subunit that covers the entire parent-unit area.

Parameters:

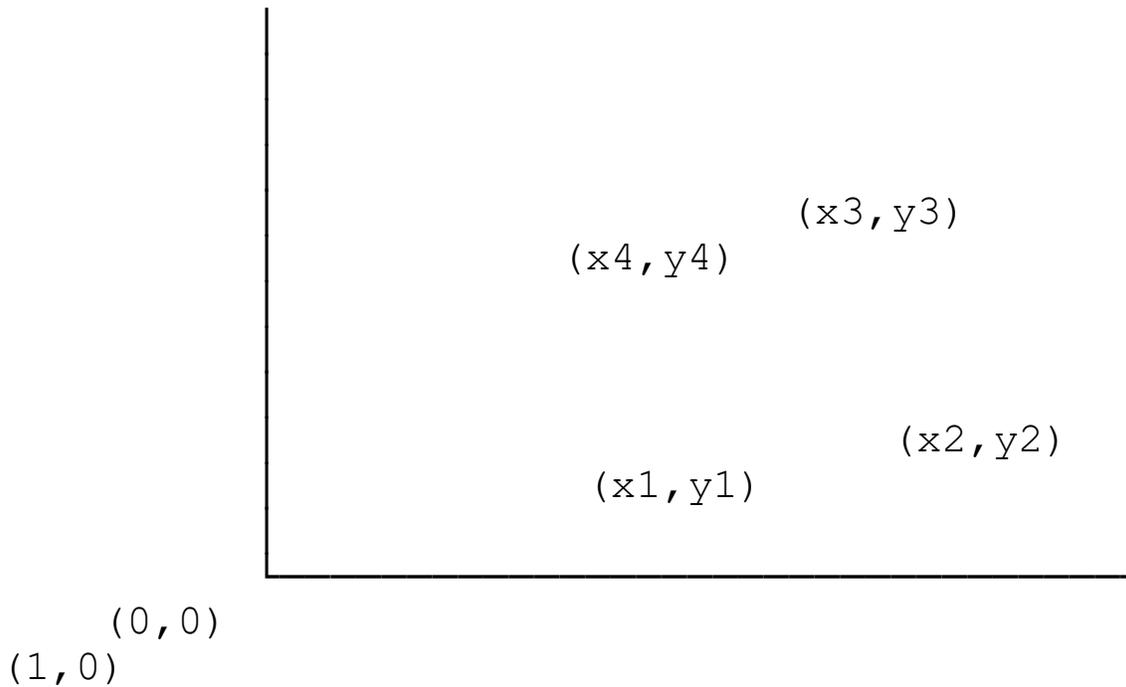
Symbol Location Point: The point at which the subunit's symbol will be displayed. In the illustration, (x,y).

Figure 5. Illustration of Unit Geographic Type 4 (Major Subarea Subunit)

Unit Rear Boundary

(0,1)
(1,1)





Unit Front Boundary

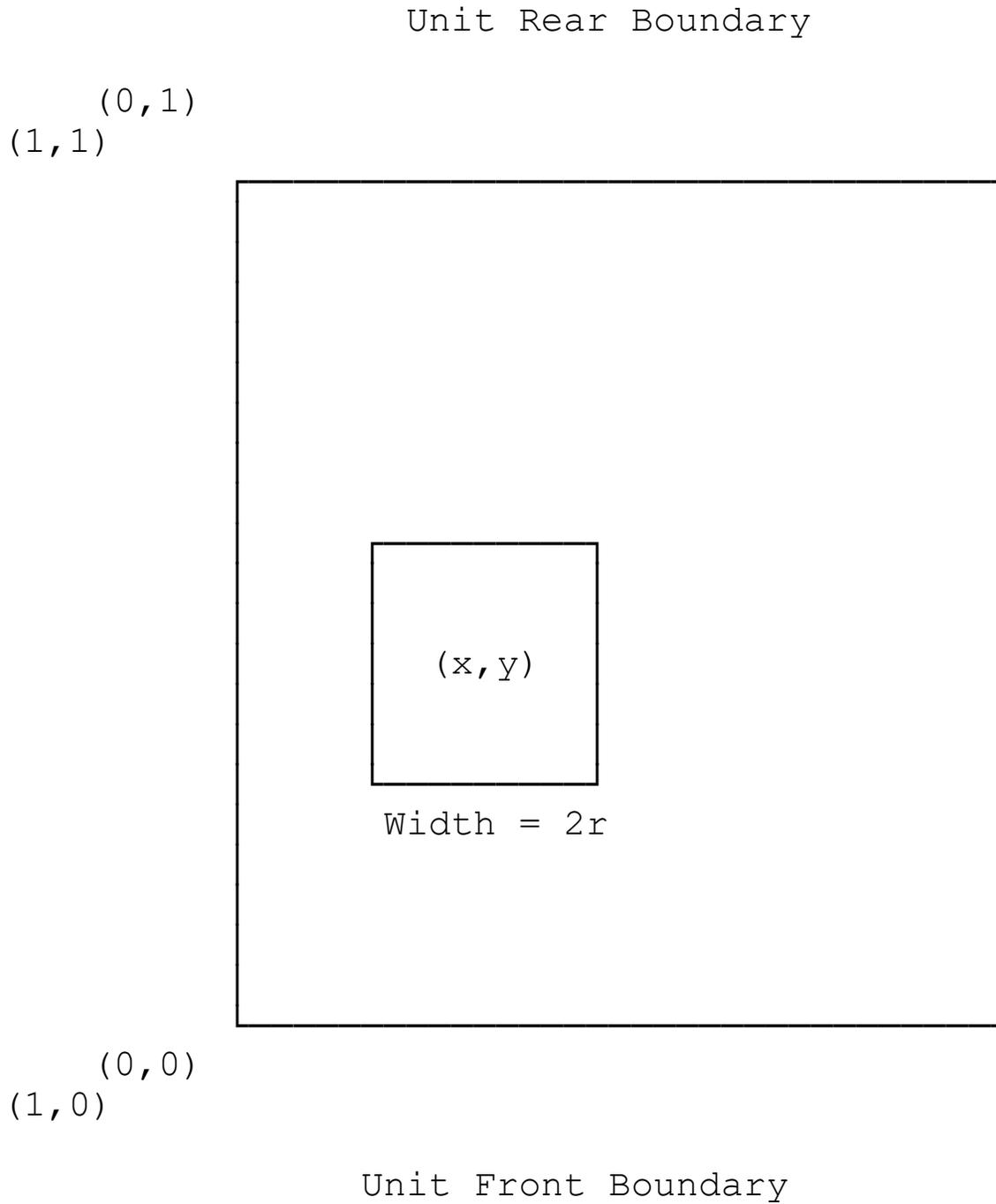
Definitions:

Geographic Type 4: A major subarea subordinate unit. A subunit whose domain is a quadrilateral of arbitrary location and orientation within the parent-unit area. (A subunit's "domain" is the area covered by the subunit.)

Parameters:

Corners: For each major subarea subunit, the user must specify the coordinates of the four corners, clockwise, starting from the corner at the right front of the unit. In the illustration, (x1,y1), (x2,y2), (x3,y3), and (x4,y4).

Figure 6. Illustration of Unit Geographic Type 5 (Large Minor Subarea Subunit)



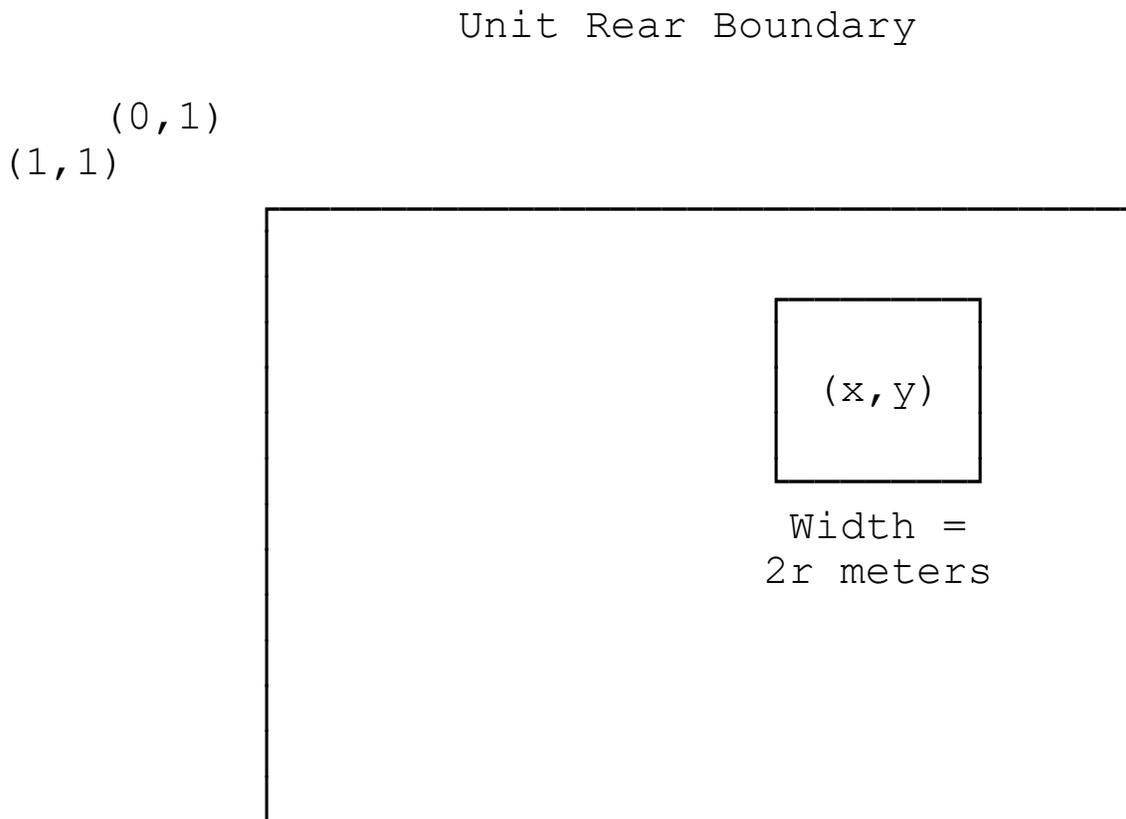
Definitions:

Geographic Type 5: A large minor subarea subordinate unit. A subunit whose domain is a square whose half-width ("radius") is specified as a fraction of the parent-unit width.

Parameters:

Location Point, Radius: The location, (x, y) , of the center of the subunit, and its half-width (radius), r , specified as a fraction of the parent-unit width (i.e., in "relative" units).

Figure 7. Illustration of Unit Geographic Type 6 (Small Minor Subarea Subunit)





Unit Front Boundary

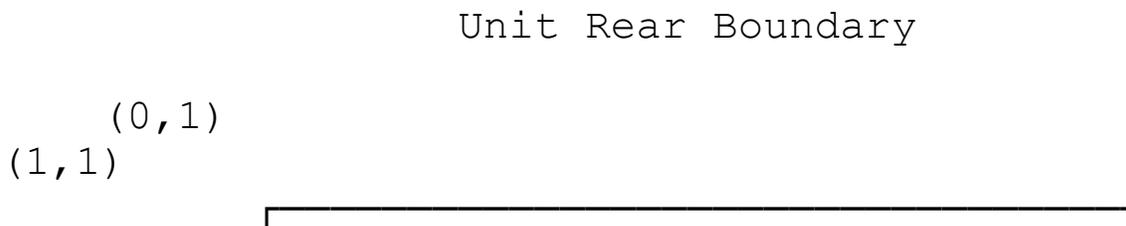
Definitions:

Geographic Type 6: A small minor subarea subordinate unit. A subunit whose domain is a square whose half-width ("radius") is specified in meters (i.e., in "absolute" units).

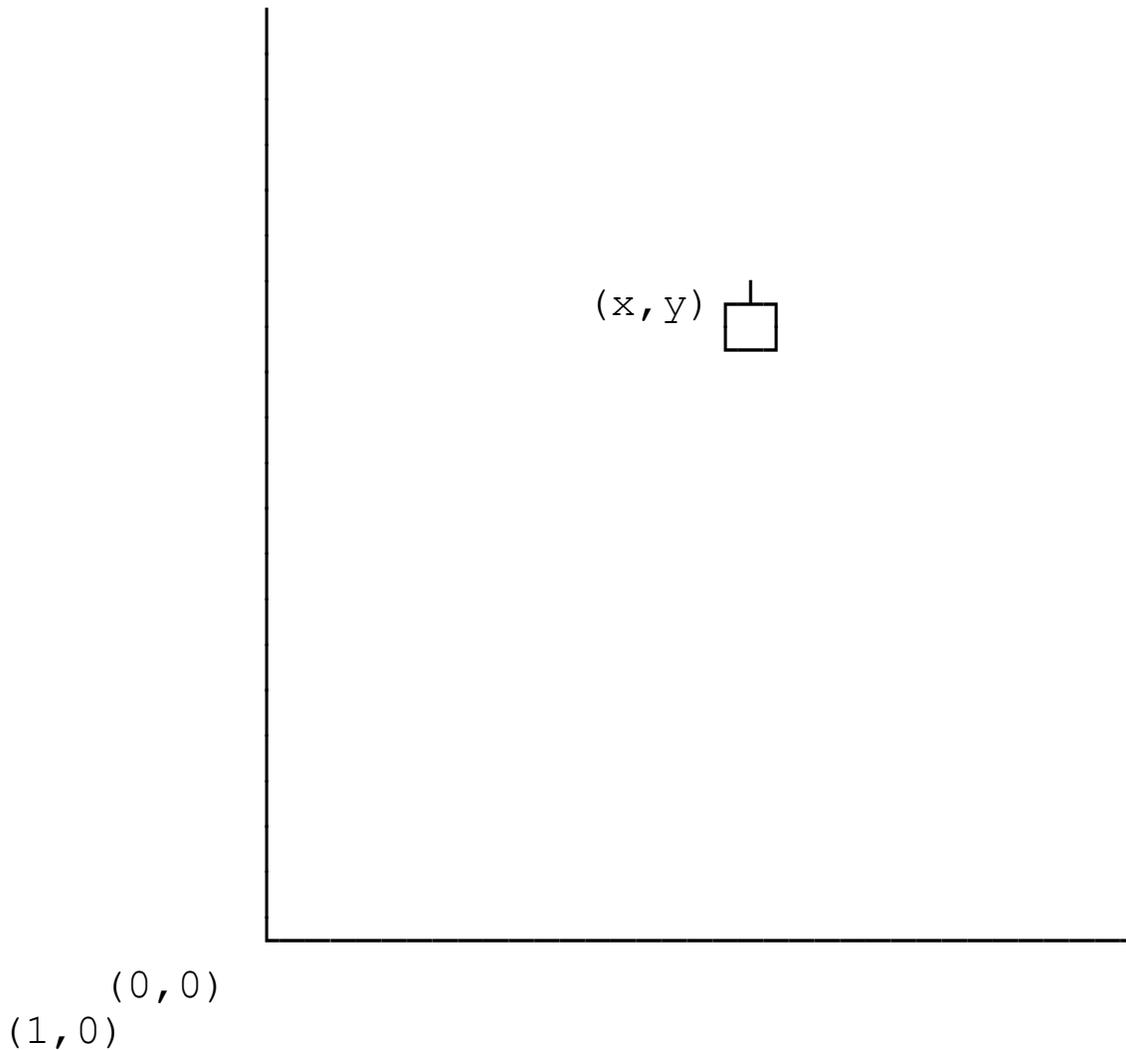
Parameters:

Location Point, Radius: The location, (x,y) , of the center of the subunit, and its half-width (radius), r , specified in meters.

Figure 8. Illustration of Unit Geographic Type 7 (Point Subitem)



Unit Rear Boundary



Unit Front Boundary

Definitions:

Geographic Type 7: A point (non-area) subordinate unit. A subunit whose area is unspecified.

Parameters:

Location Point: The location, (x,y), of the center of the subunit.

¶

Large minor subarea subordinate unit (geogtype 5). A subunit whose domain is a square whose half-width ("radius") is specified as a fraction of the unit width (i.e., whose half-width is relative to the unit size).

Small minor subarea subordinate unit (geogtype 6). A subunit whose domain is a square whose half-width (radius) is specified in meters (i.e., whose half-width is absolute).

Point subordinate items, or non-area items (geogtype 7). Platforms and equipment.

The primary purpose of the geogtype is to classify the subordinate items of a unit with respect to which of them may be repositioned by the user or by the rules of the knowledge base. (The geogtype is also referred to in drawing a unit.) In the current version of the Scenarist, only subitems of geogtypes 6 and 7 may be repositioned (i.e., absolute-radius subunits, platforms, and equipment).

One of the reasons for restricting repositioning (whether user-controlled or rule-controlled) to subitems of geogtypes 6 and 7 was that CECOM's interest in scenarios was to position radar units and equipment, and these are small units. It is easier to develop methodology for repositioning small, specialized items that contain few subordinate items and whose extent may be

represented by points or absolute-radius circles, than to develop methodology for repositioning units whose composition is complex and whose boundaries are arbitrary.

During the course of the project, some consideration was given to developing software that would allow the user to reconfigure the larger subunits of a unit (e.g., to respecify the front/rear boundary, or move a major area subunit). In the present version, if the user wishes to make such changes, he must erase the unit and redefine it ab initio ("from scratch"). This is not a desirable feature, but project resources were not available to develop general-purpose "unit editing" software. If the Scenarist undergoes additional development, this development should receive high priority. It is not a simple design change, because if the configuration of a unit is changed, all of the subunits must be correspondingly changed. Accomplishing this was not too difficult for items of geogtype 6 and 7, and would not be difficult for items of geogtypes 3, 4, and 5. It would be more difficult for items of geogtypes 1 and 2, because these types of subunits share common boundaries.

¶If the Scenarist is extended to have a general unit-editing capability, consideration should be given also to extending the system so that subunits of all geogtypes could be repositioned not only by the user but also by rules. Because of the "shared-boundary" problem, however, development of a rule-based system capability to

relocate subunits of geogtypes 1 and 2 would be difficult.

Some additional information about geographic types is the following. Units (echelons 1-11) may be of geographic types 1-6. Only units may possess specific subunits. Type 7 is reserved for use by platforms (echelon 12) and equipments (echelon 13) only. "Point subitems" (geogtype 7) are generic items (i.e., their attributes are a function of the side, echelon, and type, not the individual item). (The term "point subitem" is a little misleading: e.g., a platform may possess an area, but it is a generic characteristic.) Platform point subitems may have generic subunits, but not specific subunits. Equipment point items have no subunits (either generic or specific). The generic characteristics of platforms, including a list of up to 9 equipments and other attributes, are stored in the platform file. Note that the location is not included (it is a specific characteristic, stored only in the parent unit.) The generic characteristics of equipments are stored in the equipment file (no location data).

During the generation of a scenario, the system keeps track of how the location of an item has been determined. This information is recorded in a variable called the "placement code." The placement code has the following values:

- 0: Canonical placement (the placement code is initialized to 0 for all defined units)
- 1: User-reviewed, unchanged
- 2: User-suggested placement

- 3: User-mandated placement
- 4: Position reviewed by rules, whether moved or not
- 5: Position reviewed by rules, moved to satisfy local constraints
- 6: Subitem of geogtype 6 was moved in a search to satisfy global constraints
- 7: Subitem of geogtype 7 (platform/equipment) was moved in a search to satisfy global constraints

During the development of the Scenarist, unit data were derived from two sources. For the Beqaa Valley example, hypothetical unit were generated. A generic division was generated and copied to form specific BLUE and RED divisions. This division had two front-line brigades, and the brigades contained a number of radar sections. These units were not real units (in which the radar sections would have been division assets). They were hypothetical units devised for test purposes.

¶For the TRAILBLAZER application, information about unit composition and the organizational structure of the units was obtained from field manuals (referenced in the Test Report). The material was examined, and decisions made about what subunits of a division should be represented in the Scenarist model. The subunits to be included were any subunits whose positions affected the placement of TRAILBLAZER units/equipment.

During the test, the Government project officer expressed concern that only selected units and equipment had been included in the model. The concern was expressed that unless all of the subordinate items of a unit were included, a TRAILBLAZER unit might be placed on top of some other unit. Several comments are in order here. First, the Scenarist system repositions the subordinate items of a unit only in response to user instructions or the rules. The decision to move a subitem may involve consideration of the positions of other units, but once a subunit is moved, either by the user or by the rules, it is moved independently of all other subunits. In order to make certain that the subunit's new position does not interfere with some other subunit, that information must be included in the rules. Alternatively, this information could be included in the "action" functions that are processed in response to the firing of the rules. The point is, that information has to be embodied somewhere in the model, in order for the model to act on it.

The second point to address is the issue of what subitems must be included in the model in order for the generated scenarios to be useful (i.e., adequate) for the application at hand. If all that a user needs to know is the positions of certain radars, then only those subitems need be included whose positions affect the placement of radars. The positions of other units (e.g., a finance company) are not relevant.

If, on the other hand, it was desired to use the Scenarist to generate complete operational

laydowns, then all of the subitems of every unit would have to be included in the model, and rules for placing all of them (relative to each other) would have to be specified. It was not intended in the Scenarist proposal to develop a system for generating complete laydowns. The examples discussed in the proposal related to several uses of scenarios, but those uses were special-purpose uses such as test and evaluation, training, force structure analysis, and the like. The development of a system for producing a complete laydown was not one of the intended applications.

¶A major consideration kept in mind in the Scenarist development project was that in the development of any mathematical model or system, no model can solve all problems; rather, a model is appropriate for solving a certain class of problems. The type and level of detail to be included in a model should be the minimal level necessary in order for the model to be adequate for addressing the applications for which it is intended. This principle of "parsimony" requires that no more detail be included in a model than is necessary. In the development of the Scenarist, the design was guided mainly by the goal of developing a tool that could address scenario generation applications in test and evaluation. At the same time, it was hoped (and is believed) that with extension, it could also be applied to address scenario generation applications in other applications areas.

During the course of the Scenarist development, efforts were made to obtain detailed TO&E data

files (stored on Bernoulli-Box cartridges) from the Battlefield Electromagnetic Environments Office, with the intention of extracting data on unit compositions and organization from these files. As mentioned, this effort was unsuccessful. Had it been, we would have examined the issue of developing software to extract the needed information. Since we never obtained the needed data, this issue was never addressed.

Once data are available from some source -- either field manuals, TO&E files, or some other source -- the major issue to be addressed in a Scenarist application is what geogtype should be assigned to each subitem. In some cases, the choice is apparent. For example, a front-line brigade would be classified with geogtype 1 (front subunit). In other cases, some thought is required. The key issue to address is: over what geographic area will all of the subordinate items of a unit be spread? The answer to this question indicates the geogtype. For example, in the TRAILBLAZER application, the TRAILBLAZER units were squad/teams, included in a TRAILBLAZER section, included in a SIGINT processing platoon, included in an electronic warfare company, included in a military intelligence battalion, included in a division. The key issue to realize here was that the five TRAILBLAZER squads could be placed anywhere in the front area of the division. Hence, their parent unit -- the TRAILBLAZER section -- had to represent them as "all-area subordinate units." They themselves could be represented as small units covering a fixed-radius area -- i.e., as

"small minor subarea subordinate units." In order for the TRAILBLAZER section to be an all-area subunit, it was in turn necessary for the SIGINT processing platoon (SPP), the EW company and the MI battalion also to be all-area units. Had, for example, the MI battalion been defined as a major area subunit of a certain size (relative to the division), then all of its subordinate items (the EW company, the SPP, the TRAILBLAZER section and the TRAILBLAZER squads) would have been constrained to this area, rather than free to be located throughout the division area.

¶The process of identifying the geogtypes of subordinate units requires an educated decision on the part of someone familiar with the Scenarist structure. It is unlikely that this decision could be readily automated. In other words, the extraction of unit data from the BEEO TO&E files would have been a semi-automated process, combining human judgment with machine processing.

The Beqaa Valley and TRAILBLAZER applications did not require the placement of platforms or equipments. The smallest-echelon item placed was a radar section. For this reason, no data were placed in the "platform" or "equipment" files, and no code was developed to read these files (since the code would have depended on what information was to be stored about those items).

b. Unit, Platform and Equipment Labels and Symbols

To assist the user in interpreting what is displayed on the screen, the Scenarist uses standard military symbology. There are two types of unit representations (drawings) placed on the screen. First, a unit may be drawn in extensive detail, showing its four boundaries and showing the placement of subordinate units within it. Unit boundaries are drawn for subunits of geogtypes 1-6, and symbols are drawn for subitems of type 7 (platforms and equipment).

The subitems of type 7 are treated generically in the Scenarist. Individual-specific data are not maintained in the model for such items. Accordingly, a different symbol may be programmed for each side x echelon x type category.

Once the units and subitem symbols have been drawn, descriptors may be placed near them. There are two types of descriptors. A "name" is stored for each defined unit (whether generic or specific). A "label" is stored along with each symbol (i.e., a label is generic and is defined for each side x echelon x type category). On the drawn unit, the unit name is printed at the corner of the unit, and a label is printed next to each subordinate-unit symbol.

The user specifies unit names as part of the process of defining a unit. The symbols used during the Scenarist development are coded in a C-language function, and the matrix. The labels and symbol numbers corresponding to each side x echelon x type combination are stored in a data file. In a new application, the user would have

to program any new symbols used, and include their labels and reference numbers in the symbol/label data file.

3. Rule-Related Data

¶a. Rule Formulation

The heart of the Scenarist is the rules for determining the positions of subitems in a unit, taking into account tactical doctrine, terrain features, friendly mission, and enemy threat. In order to specify rules, it is necessary to review documents on the doctrine for deploying the units, platforms, or equipment of interest in a particular application and included in the model. To specify a rule, it is necessary to identify the factors involved in the rule and to construct an "if-then" statement that states what conclusion results if certain facts are asserted. In the field manuals, some rules may be simply stated in terms of easily measured data. As was discussed earlier, some of the factors involved in a rule may involve concepts that are difficult to operationalize, and some creativity is helpful in this regard.

b. Factors

Once all of the rules have been specified, all of the factors involved in rules should be listed. In some cases, functions may already be available in the current version of the Scenarist to compute values of these factors. The functions that are currently available are:

1. _terrain: determines terrain type at a point

2. `_elevation`: determines the elevation at a point
a location to the FEBA
3. `_distancetofeba`: computes the distance from a point
to the FEBA
4. `_horizonangle`: computes the horizon angle looking from
a point to the FEBA
5. `_accessibility`: determines whether a point is
accessible from the rear of the unit by road,
or
by moving over cells from a road without
exceeding
a 30 degree slope
6. `_lostotarget`: determines whether a
line-of-sight
condition exists between an item and its
objective
7. `_LOS`: determines whether a LOS condition
exists
between two specified points
8. `_road`: determines whether a road exists in
the cell
in which a point is located
9. `_slopetorearcell`: computes the angle from
a point
to the point one unit cell toward the unit
rear
10. `_lostootherunits`: determines whether
there is LOS
from a point (the location of a TRAILBLAZER
squad/
team) to the locations of at least two of the
(four

other) TRAILBLAZER squads of the same
TRAILBLAZER
section

11. `_lostoheadquarters`: determines whether
there is

LOS from the locations of at least two
TRAILBLAZER

squads to the symbol location point of the
TRAILBLAZER

¶section containing the units

12. `_disttootherunits`: determines the minimum
distance from

the a particular TRAILBLAZER squad to the
locations

of the other (four) TRAILBLAZER squads of
the same

TRAILBLAZER section

13. `_distancetofront`: determines the distance
from a point

to the unit front

14. `_inforwardarea`: determines whether a point
is located

in the front half of a unit

The above functions were programmed to provide values for the factors involved in rules for the Beqaa Valley and TRAILBLAZER applications considered during the development of the Scenarist. While some of the functions are quite general in nature (e.g., `_elevation`), many of them are quite specific to the particular application. This experience suggests that in any new application of the Scenarist, the user should be prepared to program a number of such functions.

c. Preprocessing

As was discussed earlier in this chapter, it is more efficient to "preprocess" some factors for all of the cells of a map, prior to beginning processing of the rules for the subunits of a unit. In the Beqaa Valley and TRAILBLAZER applications, it was seen useful to preprocess the `_accessibility` factor. A C-language function (`_createaccessibilitymap`) was developed that generated a map that specified the accessibility (0 if accessibility cannot be determined (e.g., because road data are not available); 1 if the cell is not accessible; and 2 if the cell is accessible).

Whether a user should develop other preprocessing functions will depend on the particular application. The issue rests on the amount of processing required to precompute the factor values at once for all cells of a map area compared to the amount of computation required to compute the factor values "on-line," i.e., only for those cells for which they are needed, during the rule processing.

¶An example may clarify this issue. In determining accessibility of a given point in the TRAILBLAZER application, it was necessary to initiate processing at some point on the edge of the map, and determine whether the point (cell) could be accessed by a road or without violating the accessibility criterion (30 degree grade). Since a particular cell may be accessed in many ways, it is conceivable that during the process

of determining the accessibility of one cell the accessibility of all of the other cells would be determined. In any event, the determination of the accessibility of a nearby cell would require just about the same amount of computation (starting, once again, from a point at the edge of the map). It involves hardly any more computation to determine the accessibility for all cells of the map than to determine the accessibility for a single cell. For this reason, it is far more efficient to precompute an accessibility map containing the accessibility for every cell of the map, than to compute it on an as-needed basis for individual cells during the rule processing.

In this example, the issue is that it requires about the same amount of computation to compute accessibility for a single cell of a map as for all of the cells of the map.

If the amount of computation required to evaluate a factor is approximately proportional to the number of cells for which the factor is needed, there is generally no point in precomputing a map of factor values. Instead, the factor values should be computed "on-line." This is certainly true for a factor used only in local rules. The reason for this is that it is unlikely that the factor values will be needed for every cell of the map.

For global rules, however, the situation is less clear-cut. The rules may be applied for a number of iterations, and a factor value may be needed several times for the same cell. In the current

version of the Scenarist, the maximum number of iterations for processing global rules is three, and factor values are needed only in small neighborhoods of each subitem (i.e., for eight neighboring cells). It is not efficient to precompute factor values in this case since it is unlikely that the number of function calls required to evaluate the needed factor values on-line would be greater than the number of cells in the map. If the Scenarist is modified to allow for a much larger number of iterations, this situation could change. If this is done, consideration should be given to modifying the system design so that every time a factor value is computed, it is stored in an array so that it never has to be computed again. With this design change, the user would never have to decide whether to precompute a factor or not. A drawback to this approach is, however, that a 32 x 32 cell array is needed to store the values for each precomputed factor.

d. Rule Specification

The rules are entered in a CLIPS rule file, called clipxxxx.fil (the suffix xxxx is an application identifier), in the format required by the CLIPS knowledge-based system. For the user's first application, it may be desirable to program a small number of rules both in C and in CLIPS to insure that the system has been set up properly.

¶The interface between CLIPS and the remainder of the Scenarist is accomplished in a C-language functions called _clipssuitabilityxxxx (where the suffix xxxx, as above, is an application

identifier). In this function, the factor values are transferred to CLIPS, and the result of the CLIPS processing is received back from CLIPS. As mentioned, the information received back from CLIPS is the value of a variable, `clipssuitability`, that is 0 if the location is unsuitable and 1 if the location is suitable.

During the development of the Scenarist, C-language functions were programmed to accomplish processing equivalent to that done in CLIPS. This was done to verify that CLIPS was working properly. Those functions were called `_suitabilityxxxx` (where the `xxxx` is an application identifier).

e. Actions

As was discussed earlier, the interface between CLIPS and the remainder of the Scenarist system requires that the user separate the rules into two categories, depending on whether they address local constraints or global constraints. The Scenarist operates by applying the local rules to the subitems of a unit and then by applying both the local and global rules to the subitems. Whether a rule is a local rule or a global rule is indicated by a variable called "localglobal." The user should include in every global rule a test to check whether the value of `localglobal` is equal to 1. The system operates by first setting the value of `localglobal` equal to 0 and processing the rules. In this case, only the local rules will be applied (since, in order for a global rule to be fired the value of `localglobal` must be 1). The system then sets the

value of local/global equal to 1, and reprocesses the rules. Local rules have no check on the value of local/global. Hence, this time all rules will be applied.

¶For local rules, if an item's initial (canonical or user-suggested) location is unsuitable the system executes a spiral search for a suitable location. For global rules, the action taken if an item's location is unsuitable is up to the user to decide and program. The action taken in the case of the TRAILBLAZER application was discussed earlier, i.e., the item is moved to a neighboring cell of higher elevation. What action should be taken for other applications depends on the application. If the action involves straightforward numerical computation, this is appropriately done in a C-language action function. If considerable logical processing is involved, consideration may be given to moving this processing into the CLIPS environment. This is easy to accomplish if all of the needed factor values can be precomputed and passed to CLIPS. If, however, substantial numerical processing is involved (e.g., a cell-by-cell search for a new location, with substantial processing required for each new cell examined) this would not be appropriate.

At the present time, it is not clear whether the problem of automated scenario generation is more effectively handled if it is considered as an optimization problem (i.e., search for a suitable location) or a rule-processing problem (i.e., processing a large number of rules governing the positioning of subunits). Future

applications should shed light on this issue. The issue rests on how difficult it is to handle global rules, i.e., to identify reasonable "action" functions to execute if a global rule fails. Global rules involve, implicitly, the solution of a complex (non-separable) optimization problem. This issue arose and was addressed in the TRAILBLAZER application. Without defining an optimality criterion and examining at least a modest number of laydowns with respect to this criterion, however, it is not known how "good" the quality of the laydowns produced by the Scenarist is. The use of KBSs is not an efficient means of solving complex optimization problems, if those problems can be formulated in ways that are addressable by the techniques of optimization theory.

4. Program Initialization (Project Files)

Once the user has defined the map files, unit files, symbol file, and CLIPS rule file (and a FEBA file, to be discussed in the next chapter), it is necessary to set up a "project file" that is read by the Scenarist program when it begins execution. The purpose of the project file is to save the user the trouble of inputting a lot of initialization information (names of files) over the keyboard. The project file contains the names of the map files, unit files, symbol file, CLIPS rule file (`_clipxxxx.fil`), and the names of the C-language rule function (`_suitabilityxxxx`), the preprocessing function (`_preprocessingxxxx`), the action function (`_actionxxxx`) and the Scenarist/CLIPS interface function (`_clipssuitabilityxxxx`).

D. Scenarist Program

1. System Control Functions (Menu Choices)

The Scenarist was designed with a multi-level menu system. The menus provide the user with selections of functions to execute. In general, the selection of the menu choices is done using a mouse, but the data required by the various functions are input through the keyboard.

Figure 9 presents a list of the functions available in the Scenarist through its menu options. Related functions are grouped together into a number of functional areas ("main menu choices"). When the program begins execution, control passes automatically through two of these functional areas (project selection and initial map location point selection) and then transfers to a "main menu" from which the user may select any of the remaining functional areas.

The Scenarist system is comprised of a "main" C-language program and a large number of C-language functions (subroutines). Related functions are grouped together into files referred to as "modules." The modules in turn are grouped into four major functional groupings, called Computer Software Configuration Items, or CSCIs.

The four major functional groupings of the Scenarist are: program control, input/output, map processing, and unit processing. Although

all of these activities combine to provide the Scenarist's scenario generation capability, the "unit processing" grouping could be referred to as the "scenario generation" grouping (and was, in the early documentation of the system). The program control is accomplished by means of a main program module that executes certain setup functions and then passes control to two menu control modules. The setup functions include loading the CLIPS system, registering fonts, presentation of an information screen, selection and input of a project file, and reading data from the files specified in the project file.

The input/output functions of the Scenarist occur in two places. First, the various higher-level functions used by the Scenarist to input data from disk files and the keyboard and output data to the printer are grouped together in three modules (dealing with printer, window, and mouse functions). Second, these higher-level functions and a variety of lower-level functions (keyboard entry, file reading and writing, and output to the screen) are scattered throughout most of the other Scenarist modules.

The map processing functions include activities such as reading in a map from a map file, displaying a map on the screen, zooming, and changing map locations.

The unit processing functions include activities such as defining a unit, storing a unit in a data file, retrieving a map from a data file, displaying a unit on the screen, moving

Figure 9. Scenarist Program Functions

1. Project Selection
2. Initial Map Location Point
3. System Description
4. Processing Military Units
 - (1). Define Unit
 - (2). Copy Unit
 - (3). Delete Unit
 - (4). Reposition Unit by User
 - (5). Reposition Subunits by User
 - (6). Reposition Subunits by Rules
 - (7). Reposition FEBA
 - (8). Display Unit
 - (9). Output Unit
5. Rules
6. Processing Maps
 - (1). Draw Terrain Map
 - (2). Draw Elevation Map
 - (3). Draw Road Map
 - (4). Add Vector Map with Labels
 - (5). Add Vector Map without Labels
 - (6). Draw Vector Map without Labels
 - (7). Place Unit on Map
 - (8). Zoom Map
 - (9). Change Map Location
 - (10). Change Map Files
 - (11). Print Map
7. Scenario Generation
8. Hardware Data
9. Exit

¶a unit, and repositioning the subitems of a unit (either manually or using the rules of the knowledge base).

Having described the major functional groupings of the Scenarist modules, we shall now present a description of each of the individual Scenarist functions listed in Figure 3. A large number of C-language functions is required to accomplish the functions listed in Figure 3. Those functions are described in the next chapter.

Project Selection. After the main program completes its initialization steps, it calls a C-language function that allows the user to select a project file. As discussed earlier, the project file indicates what map, unit, and rule files and functions will be used in the current run. The main program then calls another function that reads the data stored in these files into memory.

Initial Map Location Point. As discussed earlier, the Scenarist can process data from three maps -- a terrain-type map, an elevation map, and a road map. It was also mentioned that the program reads at most a 32-cell x 32-cell "window" from each map. Since the maps specified in the project file may cover different areas, the first step is to decide what (common) area will be read from the three maps. The program reads a 32-cell x 32-cell map relative to a "map location point," which is either the upper-left-hand-corner of the map or the center of the map.

The program reads header files of the terrain-type and elevation map files, and determines the location point of the intersection of these two maps. Next, the program asks if the user wishes to change the (display) map location point from the map-intersection location point to some other point. He is then asked if he wishes to define the map location point as the upper-left-hand-corner of the map or the center of the map. If the user requested to change the map location point from the map intersection point, the program then asks the user to input the coordinates of the map location point.

The program then reads in a 32-cell by 32-cell from the road map (if a road map is available).

¶At this point, control returns to the main program. The main program then calls the main menu selection function, where control resides for the remainder of the run. The main menu function presents the user with seven options, numbered 3-9 in Figure 3. Two of those options (system description and hardware data) provide the user with some information about the Scenarist system and some data about the user's computer system (e.g., the type of video monitor). Two other options -- Processing Military Units and Processing Maps -- lead to second-level menus ("submenus") that in turn contain a number of options that correspond to basic Scenarist system functions, such as drawing a map or placing a unit on a map or repositioning the subitems of a unit. Two of the main-menu options (Rules and Scenario

Generation) are simply entry points for areas in which the Scenarist system might be expanded in the future. The final main-menu option exits the user from the Scenarist program.

The paragraphs that follow describe the functional nature of each of the main-menu and submenu options of the Scenarist program. The main-menu options that do not have sub-menu options are described first.

System Description. The system description provides the user with the telephone number and address of Scenarist system points of contact at the US Army Communications-Electronics Command (CECOM) and Vista Research Corporation.

Hardware Data. Selection of this option provides the user with a description of hardware system information provided by the Microsoft "videoconfig" function. The data provided include the number of horizontal and vertical pixels on the user's screen, the number of horizontal and vertical lines, and the number of color indices.

Rules. This entry point has two uses. First, it is the entry point for the user to obtain a listing of the CLIPS rule file. During the processing of the rules to reposition subitems, the Scenarist system presents a limited amount of data about what rule or rules are violated, if a decision is made that an item is not suitably placed. If the user wants to have additional information about the rules, he may obtain a complete listing of the CLIPS rule file currently

being used, by accessing the Rules function. The CLIPS rule file not only shows the rules (in CLIPS syntax), but also contains additional information about the source of the rules.

The second use for the Rules entry point relates to planned future development of the Scenarist. If the Scenarist is to be used in the future in conjunction with the CLIPS expert system, it is our opinion that the user should be provided with a "rule editor" to make the process of specifying rules in the CLIPS syntax easier. The Rules option would be the entry point for this feature.

¶Some additional comments will be presented about the use of CLIPS in the Scenarist system. The CLIPS expert system was selected as the knowledge-based system for use in the Scenarist for a number of reasons, discussed earlier. It is incorporated into the Scenarist system in an "embedded" mode (as opposed to an interactive mode in which the user generates and executes rules interactively over the keyboard). While the CLIPS system performed adequately in the developmental demonstration of the Scenarist, based on our experience it is not particularly easy to use. In fact, it is conceivable that the difficulty in using CLIPS might well deter some potential users from using the Scenarist.

The major problem experienced with CLIPS is that there is not a user-friendly way of generating rules. The user must become quite familiar with the nature of the CLIPS system and, in essence, learn the CLIPS "language" for rule specification and construction of a CLIPS rule

file. Based on our experience, our opinion is that this is not easy to do.

In view of the difficulties in using CLIPS, if future development work on the Scenarist occurs, serious consideration should be given either to replacing CLIPS with an easier-to-use KBS, or to the development of a "rule editor" that would enable the user to specify rules and create a CLIPS rule file in an easy fashion. As mentioned, the entry point for such a rule editor would be the Rules main-menu option.

Scenario Generation. This option is simply a "hook" to an area for possible future development of the Scenarist. The Scenarist is currently set up to enable the user to view (on the screen) the results of applying the rules to reposition the subitems of a unit. This type of interface is appropriate for generating "small" scenarios, i.e., scenarios involving a small number of units, for which only a small number of subitems have been defined. In the future, it is conceivable that the Scenarist could be applied to generate scenarios in a case in which a large number of units and subitems could be involved. In such cases, the user may not wish to view the results of applying rules to each and every unit, but simply wish to process all of the units in a noninteractive mode. The code for such processing has not been developed, but it would be accessed through the Scenario Generation menu option.

Exit. This main-menu option terminates Scenarist processing and exits the Scenarist system.

Processing Military Units. This main-menu option is one of the two principal main-menu options. When selected, it immediately presents the user with a second-level menu that offers the user the option of executing any one of nine unit-processing functions:

- (1). Define Unit
- (2). Copy Unit
- (3). Delete Unit

- ¶(4). Reposition Unit by User
- (5). Reposition Subunits by User
- (6). Reposition Subunits by Rules
- (7). Reposition FEBA
- (8). Display Unit
- (9). Output Units

The functional capabilities of each of these nine options will now be described.

Before describing these options, however, some comments will be made about the level of detail to be used in the description. The level of detail presented below may seem somewhat high for a "top-level" design. The purpose of the top-level design is, however, to identify the functional capabilities of the Scenarist, and the number of functions in the Scenarist is substantial. The top-level design will be described in this report by identifying the variables over which the user has direct control (through data input, either by file or keyboard), and the functions of the system with respect to these variables.

The top-level design is heavily conceptual in nature; it specifies the general sorts of operations to be conducted, and the general level of detail at which those operations will be implemented. The exact procedures for implementing these "top-level" functions is the subject of the detailed design. That is, the detailed system design is concerned with exact definitions of the variables, specification of the level of measurement of the variables (discrete/categorical, discrete/ordinal, continuous), and specification of the system's functions in terms of these variables. From the top-level design specification, it is clear what data elements must be input to the model, and what data elements will be output from the model.

¶It is helpful to distinguish between the detailed system design and the detailed software design. This distinction is easily blurred, because the Scenarist is primarily a software system. The detailed system design is concerned with specification of variable definitions and functions -- essentially, a mathematical description of the system, that identifies the entities of a model and defines their interrelationships. The detailed software design is concerned with implementation of this system as a computer program and associated data files. That is, it is concerned with specification of data types of variables and the combining of related variables into data structures; combining the functions into groups called modules, with specification of the flow of data among the modules and classification of

variables as global or local; specification of the methods for inputting, storing, and retrieving the data; specification of whether data are to be input using the keyboard or a mouse or by reading from a disk file; specification of data formats, file structures and storage types (binary or text); specification of what data are to be output, their format, and the output medium (file, screen, printer); and specification of the particular languages ("standard" C, object-oriented C++) and existing-code programs or packages (windows, mouse control functions, graphics subroutines, printer subroutines) to be used. This final report presents a considerable amount of information about the system detailed design, and somewhat less about the detailed software design. The Scenarist Programmer's Manual and Variable Glossary present information about the detailed software design. The final report includes information about why certain approaches were employed in the software design, but it does not include detailed software design documentation (e.g., data flow diagrams, definitions of all functions and variables).

In spite of the above criteria for distinguishing between the system design and the software design, the classification of a design decision to one or the other design aspects is in some cases arbitrary. For example, the contract calls for the use of an expert system in the Scenarist. In the top-level design process, it was decided to use the CLIPS system for the Scenarist's expert system. The CLIPS system and other expert systems are, however, software packages, and the choice of the expert system

could have been classified either as a top-level system design issue or a top-level software design issue.

In some cases, the modeling approach (which is often essentially a system design issue) and the choice of software are closely related. For example, the Scenarist design is heavily object-oriented, and (as discussed earlier) a decision could have been made to use object-oriented C++ as the programming language instead of the non-object-oriented Microsoft C that was in fact selected. If the decision had been made to use C++, the model specification would have been adjusted to facilitate easy use and take full advantage of the special features of C++'s object-oriented features (encapsulation, polymorphism, inheritance).

Define Unit. This function enables the user to specify the characteristics of a military unit. The function asks the user a large number of questions about the unit, its parent unit (if any), its subordinate units, and its objective and mission (if any). The user answers some of the questions using the mouse, but most of the answers require keyboard input. Once all of the data have been entered, they are stored in a file for future retrieval by the Scenarist system.

¶The information requested by the Define Units function is the following:

1. Whether the unit to be defined is a generic unit or a specific unit (a generic unit is a geographic-independent structural specification of a unit, used to facilitate the

making of specific units, which are actual units located on a map)

2. The unit name

3. The unit code (The unit code is an identifier used by the Scenarist to store and retrieve units; the unit's echelon is derived from the unit code.)

4. The unit type (an identifier used to distinguish units of the same echelon)

5. If the unit being defined is a specific unit:

a. The unit's parent-unit's code, which will be referred to as "parentcode"

b. A user-specified unit identifying number (optional), which will be referred to as "idno"

c. The parent-unit's idno

6. If the unit being defined is a specific unit whose parent unit is not defined, the map coordinates of the unit's corners (if a specific unit's parent unit is defined, the units corner locations are derived from the parent unit's specification)

7. The unit's front/rear boundary

8. The number of on-line (front) subordinate units ("subunits") in the unit (these are subunits of geogtype 1)

9. For each front subunit, the subunit's echelon, number, type, and idno (for generic units, specify the idno as 0)

10. The boundaries between each front subunit

11. The number of reserve (rear) subunits in the unit (subunits of geogtype 2)

12. For each rear subunit, the subunit's echelon, number, type and idno

13. The boundaries between each rear subunit

14. The number of all-area subunits (subunits of geogtype 3)
15. For each all-area subunit, the subunit's echelon, number, type, idno, and symbol location point
16. The number of major subarea subunits (subunits of geogtype 4)
17. For each major subarea subunit, the subunit's echelon, number, type, and four corner locations (in standardized coordinates, relative to the unit's boundaries)
18. The number of large minor subarea subunits (subunits of geogtype 5)
19. For each major subarea subunit, the subunit's echelon, number, idno, location point, and radius (relative to the size of the unit)
20. The number of small minor subarea subunits (subunits of geogtype 6)

- ¶21. For each small minor subarea subunit, the subunit's echelon, number, idno, location point, and radius (in meters)
22. The number of point subitems (subunits of geogtype 7)
23. For each point subitem, the echelon, number, idno, and location point
24. For specific units, the unit's objective, mission and avenue of approach to the objective (if any)

Copy Unit. This function enables the user to define a new unit very quickly, by "copying" an existing unit, i.e., by using much of the data in the record of an existing unit and changing

the unit's idno, parent, location, objective, mission, and avenue of approach. The user may copy either a generic unit or a specific unit to a unit of either type.

The information required as input to the Copy Unit function is the following.

1. Whether the unit to be copied is a generic unit (i.e., a unit stored in the generic unit file) or a specific unit (i.e., a unit stored in the specific unit file)
2. New unit's name
3. Old unit's code
4. New unit's code
5. Parent unit's code
6. idno and parent unit's idno
7. If the unit being defined is a specific unit whose parent unit is not defined, the map coordinates of the unit's corners (if a specific unit's parent unit is defined, the units corner locations are derived from the parent unit's specification)
8. For specific units, the unit's objective, mission and avenue of approach to the objective (if any)

Delete Unit. This function enables the user to delete a unit from the generic unit file or from the specific unit file. This function is would be used if a unit's definition was determined to be flawed, and it was desired to erase the unit (i.e., delete it from the unit file). Also, as mentioned earlier, the Scenarist possesses a very limited unit-editing capability -- the present version can only modify the locations of subitems of geogtypes 6 and 7. If the user

wishes to make any other modifications to a unit, he must erase it and redefine it.

The information required as input to the Delete Unit function is the following.

1. Whether it is desired to delete a unit from the generic unit file or the specific unit file
2. The unit code

Reposition Unit by User. This function enables the user to reposition (move) an entire unit. Since only specific units have map locations (generic units do not), this function operates only on specific units, not on generic units. The function automatically repositions all of the unit's progeny (i.e., the unit's subordinate units, the subordinate units' subordinate units, and so on). Repositioning is allowed only for units that have no defined parent units, i.e., that are at the "top" of the definition "chain." If it is desired to reposition a unit that has a defined parent unit, it is necessary either to reposition the parent unit or, for units of geogtype 7, to apply the function "reposition subunits by user," to the parent unit. (If the parent unit also has a defined parent, this same condition holds.)

If the unit to be repositioned has an objective, the user is given the option of repositioning the objective. Recall that the location of the objective and the avenue of approach to the objective are defined in map coordinates, independent of the unit's location.

The information required as input to the Reposition Unit by User function is the following.

1. Unit code
2. Four corners of the unit boundary, counterclockwise
3. If the unit has an objective:
 - a. Whether to delete the objective
 - b. Whether to retain the current objective without change
4. If the unit has no objective and it is desired to define one, or if the unit currently has an objective and it is desired to change it, the objective, mission type, and the avenue of approach

Reposition Subunits by User. This function enables the user to reposition subordinate items of geogtypes 6 and 7 in a unit. Items of geogtype 6 are small minor subarea subordinate units whose radius is specified in meters. Items of geogtype 7 are nonarea subitems (platforms and equipment). This function is the initial version of a general unit editing function that would enable the user to modify any of the features of a unit. It enables the user to reposition those subitems whose locations do not affect any other subitems.

The information required as input to the Reposition Subunits by User function is the following.

1. Whether the unit whose subunits are to be repositioned is a generic unit or a specific unit
2. The unit code

¶3. If the unit contains subitems of geogtype 6, the user must input NMINORSUBAREAABS sets (i.e., one set for each of the subitems of geogtype 6) of location point coordinates, x,y (0-1.0), radius, r (meters) and a placement code. The placement codes are as follows:

- a. Placement code 1: user-reviewed location for subitem, location unchanged
- b. Placement code 2: user-suggested placement
- c. Placement code 3: user-mandated placement

These codes mean the following. Placement code 1 is used if the user chooses not to relocate a subunit. Placement code 2 is used if the user chooses to relocate a subunit, but will allow the rules, if applied later, to change that position. Placement code 3 is used if the user chooses to relocate a subunit and the rules are not allowed to alter that location.

4. If the unit contains subitems of geogtype 7 (platforms, equipment), the user must input NPOINT sets (i.e., one set for each of the subitems of geogtype 7) of location point and a placement code

Reposition Subunits by Rules. This function is the "heart" of the Scenarist. It is the function that applies the rules of the knowledge base to reposition the subitems of a unit. The rules may be used only to reposition subitems of geogtypes 6 and 7. As discussed earlier, the function operates by applying all local rules and then

applying all global rules. The rules are applied only to units that appear on the screen.

Note that rules are applied to subunits, not to units. The reason for this is that the original concept of the Scenarist was to locate a unit on the battlefield, and then use rules to position its subordinate items, taking into account tactical doctrine, terrain features, friendly mission, and enemy threat. It was never intended to develop a rule-based system for moving independent units around the battlefield --that is the subject of a war-game model, not a scenario generator.

The information required as input to the Reposition Subunits by Rules function is the following.

1. Whether the user wishes to use rules stored in the CLIPS knowledge-based system or specified in C-language subroutines
2. The unit code

Reposition FEBA. This function allows the user to reposition the Forward Edge of the Battle Area (FEBA). The FEBA is an imaginary line separating the friendly and enemy units. Its position is specified by the user independently of the positions of units, and the user would wish to reposition it if substantial changes were made in the positions of the friendly and enemy units.

¶Specification of the FEBA is optional. Its position must be specified to the Scenarist only if the user wishes to refer to its location in the rules (e.g., in a rule requiring that a

particular type of unit must remain a certain distance behind the FEBA).

The FEBA is represented as a sequence of straight-line segments connecting up to five points.

The information required as input to the Reposition FEBA function is the following.

1. The number of points defining the FEBA
2. The map coordinates for each of the FEBA points

Display Unit. This function enables the user to display a specific unit on a map. When this function is used, the map is "zoomed" to show the minimal area required to display all of the unit on the map (not including the unit's objective or avenue of approach). The Display Unit function allows the user to see all of the units in the specific unit file, and to select one of them. The unit may be selected either by index (rank in the file), idno, or code.

Note that idno's may not be unique (i.e., two or more units may possess the same idno). If the user selects a unit by idno, the system selects the first unit in the file having that idno.

The Display Unit function works only for specific units. No similar function exists for displaying generic units.

The information required as input to the Display Unit function is the following.

1. Whether the user wishes to see a listing of all of the units in the specific unit file.
2. By what identifier the user wishes to select a unit: index, idno, or code

Output Units. This function formats the data describing all of the units in either the generic unit file or the specific unit file and outputs this data to the printer or to a file of the user's choosing. The data defining a specific unit or a generic unit are contained in a data structure. To simplify the code, units are stored in and retrieved from the unit files by data structure. That is, an entire unit data structure is written into the file and read from the file (using binary reads and writes). Because the data are stored in binary form, it is not possible to review the file contents by printing the file. Instead, it is first necessary to format it, and write the formatted data to the printer or to a file.

¶Once a user has used the Scenarist to generate a scenario, i.e., to define a number of units and apply the rules to reposition the subitems of those units, he will want to input the locations of the units and subitems into some other model. The Output Units function places this information into a file of the user's choice (or sends it to the printer) in text format. The user may then reformat it in any desired fashion for use in his intended application.

A scenario is defined in terms of specific units (which have map locations), and it is the specific unit file (reformatted) that the user would wish to input to another model. The Output

Units function may also be used to output the generic units file. This information is useful during the process of defining units.

The information required as input to the Output Units function is the following.

1. Whether the user wishes to output the generic unit file or the specific unit file

Processing Maps. This main-menu option is the second of the two principal main-menu options (the other one being the Processing Military Units option just described). When selected, it immediately presents the user with a second-level menu that offers the user the option of executing any one of eleven map-processing functions:

- (1). Draw Terrain Map
- (2). Draw Elevation Map
- (3). Draw Road Map
- (4). Add Vector Map with Labels
- (5). Add Vector Map without Labels
- (6). Draw Vector Map without Labels
- (7). Place Unit on Map
- (8). Zoom Map
- (9). Change Map Location
- (10). Change Map Files
- (11). Print Map

The functional capabilities of each of these eleven options will now be described.

Draw Terrain Map. As discussed earlier, the Scenarist draws two types of maps -- vector maps use for backgrounds, and cellular maps used for analysis. When first placed on the screen, the cellular maps are defined by a 32 x 32 rectangular

grid. The user may then zoom the map in various ways.

In the current version of the Scenarist, the user may define up to three cellular maps -- a terrain-type map, an elevation map, and a road map (optional). The function Draw Terrain Map draws the terrain-type map.

¶There is no data input required from the user for the Draw Terrain Map function. The terrain map that is currently in memory (i.e., that was read into memory at the beginning of the Scenarist run or during the execution of the Change Map Files function, to be described later) is immediately drawn on the screen.

Draw Elevation Map. If the user selects this function, the system immediately draws the elevation map that is currently in memory. No data input is required from the user.

Draw Road Map. If the user selects this function, the system operates in either of two ways. If the user has not accessed the function "Reposition Subunits by Rules" in the TRAILBLAZER application (problem 02), the system immediately draws the road map that is currently in memory. If no road map has been read into memory, this map displays all "no data" cells.

If the user has accessed the function "Reposition Subunits by Rules" in the TRAILBLAZER application, then the user is presented with the option of displaying either the road

availability map or the TRAILBLAZER accessibility map.

Add Vector Map with Labels. If the user selects this function, the system draws a vector map over whatever map is currently on the screen. The vector map contains all of the area objects defined in the area objects map file, all of the linear objects defined in the linear objects map file, and all of the point objects defined in the point objects map file. No data input is required from the user.

This function is intended for use when there is already a map on the screen. It draws only the geographic objects -- no map borders or coordinate labels are drawn (since they are presumed to be already on the screen).

Add Vector Map without Labels. If the number of objects in the area objects, linear objects, or point objects is very large, the screen may become obscured by the object labels, when the user executes the Add Vector Map with Labels function. If this happens, it is desirable to suppress the labels. The function Add Vector Map without Labels operates exactly the same as the function Add Vector Map with Labels (i.e., it draws all of the area, linear and point map objects included in the geographic object map files), except that it prints no labels. No data input is required from the user.

Draw Vector Map without Labels. As has been discussed, while cellular maps are useful for analysis purposes, they are not pleasant to look

at -- they are grainy at best and blocky at worst. In most cases, the user will prefer to place a vector background map on the screen. The vector map provides a pleasing background for the units, and a useful frame of reference for placing or repositioning units. Vector maps are also "prettier" for report purposes than cellular maps.

The Draw Vector Map without Labels function draws a vector map on the screen. This function is the same as the Add Vector Map without Labels function, except that a complete map (borders and coordinates) is drawn.

If the user also wants the geographic object map labels, he should execute the Add Vector Map with Labels function after executing the Draw Vector Map without Labels function.

No data input is required from the user for this function.

Place Unit on Map. This function draws a unit on a map. The unit to be drawn on the map is identified by the user by means of the unit code. If the user has already referred to a unit (e.g., in a call to Display Units or any other function requiring the user to identify a unit by code), the user may request the system to draw that previously identified unit.

The user may place as many units on the map as he wishes.

The information required as input to the Place Unit on Map function is the following.

1. The unit's code

Zoom Map. This function zooms the map. The user has several options. If he has already zoomed the map, he may request that the map be "unzoomed," to return to the original map. Or, he may require that the zoomed map coordinates be the coordinates of the bounding rectangle of a unit. (A bounding rectangle is the smallest-sized rectangle, with sides horizontal and vertical, that includes the unit. "The" coordinates of a bounding rectangle are the coordinates of any two opposite corners of the rectangle). Or, he may specify the coordinates of the top-left corner of the map and the map width, in meters.

The information required as input to the Zoom Map function is the following.

1. Whether the user wishes to return to the original map
2. Whether the user wishes to set the zoom map coordinates as the bounding rectangle of a unit, in which case the user must also input the code or specify that the last-input unit code be used.
3. If neither options 1 nor 2 is used, the user must specify the coordinates of the top-left corner of the map and the map width, in meters.

Change Map Location. This function enables the user to extract a 32 x 32 map from an arbitrary location of any one of the three map files. The procedure for using this function is exactly the

same as the procedure described under the main-menu option, Initial Map Location Point.

Change Map Files. The user has two ways of specifying what map files are to be used by the Scenarist. First, he may specify the names of the map files in the program file. When the user selects a particular program file at the beginning of a Scenarist run, the system reads the contents of the map files specified in the program file in to memory.

The Change Map Files function enables the user to change the map files. This option is selected when the user has completed processing at a particular level of map resolution and wishes to proceed to a higher level of resolution. Alternatively, the user may terminate the Scenarist run, restart it, and select a new project file containing the desired higher-resolution map. The Change Map Files option saves the user the trouble of exiting the Scenarist program (and defining another project file, if one containing the desired selection of maps does not exist).

Note that the Scenarist processing should be done with maps (terrain-type, elevation, and road) that are of comparable resolution. This means that the user will typically change all three map files, rather than just one of them (provided that similar-resolution maps are available for all three map types).

¶VII. Scenarist Development Software Detailed Design

The purpose of this chapter on detailed design is to provide a user or modifier of the system with a clear understanding of how the system software works. It is not intended, however, to present a complete description of all of the data flows between the system modules, or a line-by-line description of the system software, or definitions of all of the functions and variables of the Scenarist programs. That description is presented in the Scenarist Programmer's Manual and Variable Glossary.

The development software of the Scenarist system is concerned primarily with data manipulation that is mathematically simple in concept, although it may be complicated in amount (e.g., retrieving and displaying maps; creating, storing, retrieving, modifying and restoring units). The development software is not concerned with implementation of complex mathematical algorithms, such as optimization or statistical estimation procedures. For this reason, it is not necessary to use a lot of mathematical formulas to describe the system, as would be appropriate for the description of a complex mathematical algorithm, to describe the Scenarist system. Description of the inference-engine (rule processing procedures) of CLIPS might require some mathematical formulas, but description of that off-the-shelf component's design is not within the scope of the contract.

Given the nature of the Scenarist system, description and discussion of the detailed design is facilitated by presenting it in programming terminology alone, rather than by presenting a model description in traditional mathematical symbology and then describing the model implementation in programming terminology. The use of programming terminology alone implies that each variable has but a single symbol (i.e., the C-language variable name used in the program), rather than two symbols (i.e., a traditional mathematical symbol and the C-language name). In view of the advantages of restricting the system description to the use of programming terminology, no traditional mathematical system description will be presented.

¶In other words, the Scenarist development effort was primarily concerned with the application of a well-defined mathematical procedure to solve a specific problem (i.e., the application of an existing KBS to generate scenarios), not with the development of new mathematical theory to generate scenarios. The project was in essence a software engineering effort, not a mathematical research study. Most of the project system design effort centered on development of the software subsystem, not on the development of mathematical algorithms. For this reason, the system can be described quite readily using the terminology and symbology of software engineering, rather than of mathematics.

The Scenarist variables are of various types, e.g., characters; scalars (integers, floating point numbers); vectors and matrices ("arrays"); data structures; pointers to structures; and arrays of data structures. Some comments are in order regarding the definition of the term "data structure," for readers who are unfamiliar with this concept. A "data structure" is an ordered set of related variables, which may be of various types; it corresponds to the concept of a record in a file (in fact, in the Pascal and Modula-2 programming languages, data structures are referred to as records). In C, the programmer can operate on data structures (e.g., move them in and out of files), whereas in FORTRAN he cannot. Major advantages of data structures is that their use makes programs easier to describe, understand, develop and maintain, particularly if the program involves a lot of manipulation of complex data or objects. All of the attributes of an object may be assembled together in a single data structure, just as is done in a file, even though those attributes are of different types. The use of data structures in a program is analogous to storing sample data in a single file by observation, rather than in several files by variable.

Most of the Scenarist program is concerned with functions for processing maps and units. For this reason, most of the Scenarist's data structures (and other variables) fall into two categories -- those relating to maps and those relating to units. It is not the case that all of the variables relating to a map are included in a single data structure. Instead, similar

variables are combined into groups in ways that facilitate efficient handling. In the Scenarist program, a single data structure was defined that contains all of the variables describing a unit. On the other hand, a number of different data structures were used for maps. A map of a given geographic area may be a composite that includes a cellular map and from one to three vector maps. The user may wish to combine these map elements in various ways. For this reason, and since the procedures used to draw cellular maps differ considerably from the procedures used to draw vector maps, it is convenient (and also enhances program understandability) to keep the various map elements in different data structures. Also, the headers of the cellular map files were processed independently of the array of cell data, and whereas the header is used only once in a run (to show the user the map identification) the cell data could be used many times in a run. For this reason, the cellular map header was defined in a structure separate from the cell data.

¶The Scenarist program makes effective use of data structures. There is an intimate relationship between the data structures and the algorithms (functions) used to operate on them. Because of this close relationship, and because one of the reasons for using data structures is to enhance program description, the detailed design description that is presented in this chapter makes considerable reference to the program's data structures. In short, the detailed design description is a description of

the major data structures of the Scenarist and the algorithms (functions) that operate on them.

The dimensions of the vectors and matrices are specified in the Scenarist program code. Many of these variables are initialized (obtain their initial values) from the data files, and others obtain their initial values from the keyboard or during program processing.

A. Overview of the Scenarist Software System Detailed Design

1. Approach to Software System Detailed Design Documentation

The Scenarist software system program is comprised of the Scenarist program (which consists in turn of variables and functions (algorithms) that operate on those variables), and data files (that are accessed by the program). The values of the program variables are initialized (set) at the beginning of a program run either by using initialization statements or by reading the values from files or by requesting keyboard/mouse input from the user. The values may be changed during the course of the program execution by the various program functions. This section identifies the major program variables and all of the program functions and system data files. These items will be described in detail in subsequent sections of the chapter.

There is no attempt, either in this final report or in the program documentation, to define and

describe every single variable in the Scenarist program. Many variables are required to execute the various Scenarist functions, in addition to those that correspond to data input with which the user is familiar. Many of these variables are of no interest to the system user, but are of concern only to a programmer attempting to modify the system software. In the system documentation -- both that presented in this chapter and that presented in the Programmer's Manual and Variable Glossary -- the goal has been to present a description at a level of detail that will enable the user to make effective use of the system and the programmer to modify the system.

¶Relatively little detail is provided on operating system functions and input/output functions, whose formats will vary from compiler to compiler and operating system to operating system. Many of the higher-level printer, screen, and mouse functions were extracted from texts on MS-DOS, C, and AT-BIOS. Those functions will be replaced by other similar functions if the Scenarist is ported to a computer having a different operating system or compiler. This low level of detail for documenting printer and graphics input/output is consistent with the project officer's expressed desire to emphasize development of a "core" scenario generator and expend a minimum of project resources on graphics development.

If the Scenarist is successful in the future, the software will no doubt be modified, at which point a high-detail description of every variable will become obsolete. The production

of high-detail program documentation is expensive, and no point was seen in allocating a large portion of the project's resources to documentation that either would be of no value if the system was unsuccessful, or of limited, short-lived value if the system was successful. That approach would have served to decrease the likelihood that the Scenarist would be successful (since a substantially lower level of resources would have gone into program design). Instead, the allocation of project resources was balanced between allocating most of the resources to system design and development, and reserving a reasonable portion for the production of documentation that would facilitate future program modifications.

2. Major Program Variables

a. Explanation of Scenarist Data Types

The major variables of the Scenarist program are defined below, along with the definitions of the major data structure types used in the program. As mentioned above, a data structure type is a "composite" data type comprised of other data types. The concept of data structures was introduced to enable a programmer to use a single identifier to refer to a collection of variables of mixed types. In FORTRAN, only "noncomposite" data types are allowed, such as character, integer, real, complex, array, and the like. In C, a data type may be a programmer-specified composite of various types, including other data structures.

The major data structures used in the Scenarist are defined below in the notation used in the C language. For readers not familiar with C, a structure type is defined simply by listing all of the variables ("components," "members," or "fields") contained in the structure, along with the variable type (type first, variable name(s) second).

b. Listing of Major Scenarist Variables

¶The major variables of the Scenarist program are listed below.

(1). Map-Related Variables

The following are the major map-related variables used by the Scenarist program.

Variables Related to Cellular Maps

1. mapinf, a data structure array each of whose array components contains over a dozen variables describing the cellular maps (terrain-type, road availability, elevation, and TRAILBLAZER accessibility) stored in memory by the Scenarist (and stored in the headers at the beginning of cellular map files)

2. geodisc, an array containing the cell values of the three 32-cell by 32-cell discrete-variable cell maps (terrain-type, road availability, and TRAILBLAZER accessibility) stored in memory by the Scenarist

3. geocont, an array containing the cell values of the single 32-cell by 32-cell

continuous-variable cell map (elevation) stored in memory by the Scenarist

Variables Related to Vector Maps

1. `areaobject`, a data structure containing over a dozen variables describing a area-type geographic object (such as a body of water)
2. `linearobject`, a data structure containing over a dozen variables describing a linear-type geographic object (such as a road)
3. `ptobject`, a data structure containing variables that describe point-type geographic objects (such as a mountain peak)

(2). Unit-Related Variables

The following is a list of the variables used by the Scenarist that are related to military items (units, platforms, equipment).

1. `genericunit`, `specificunit`, `unitblank`, `unit0`, `unit1`: data structures (of data structure type "unit") containing several dozen variables describing military items.
2. `code`, `parentcode`: vector arrays that contain the 13-digit code identifying a unit
3. `symbols`, an array containing symbol numbers for military items (classified by side, echelon, and type)
4. `labels`, an array containing labels for military items (classified by side, echelon, and type)

¶3. Scenarist Data Files

The following is a listing of the types of data files used by the Scenarist program. The list does not include files used by the CLIPS expert system, other than the two referred to in the Scenarist program. The following list includes the name of the variable in which the MS-DOS system name of the file is stored, the name of the pointer variable containing the address of information defining the file, and an example of an MS-DOS file name for each file type. Additional explanation of these terms and of the Scenarist file-naming convention will be presented later.

```
project      file:      filenamesn,      filenamesp,
proj0101.fil
terrain-type  map:      geodfiln,      geodfilp,
geod0101.fil
elevation map: geocfiln, geocfilp, geoc0101.fil
road map: georfiln, georfilp, geor0101.fil
geographic area-objects map: geoafiln, geofilp,
geoa01.fil
geographic  linear-objects  map:  geofiln,
geofilp, geol01.fil
geographic  point-objects  map:  geopfiln,
geopfilp, geop01.fil
generic unit file: genunitfiln, genunitfilp,
genu01.fil
specific unit file: specunitfiln, specunitfilp,
spec01.fil
scratch      file:      scratchfiln,      scratchfilp,
scra01.fil
symbol file: symbolfiln, symbolfilp, symb01.fil
platform file: platformfiln, platformfilp,
plat01.fil
equipment file: eqptfiln, eqptfilp, eqpt01.fil
```

FEBA file: febafiln, febafilp, feba01.fil
introduction file: introfiln, intorfilp,
intro.fil
summary information file: introfiln, introfilp,
highlits.fil
CLIPS rule file: clipfiln, clipfilp, clip01.fil
CLIPS output file: dribble.txt
output file: outputfiln, outputfilp, out01.fil

4. Scenarist Functions

The following is a list of the more "prominent" of the C-language functions of the Scenarist program. The functions are grouped into categories corresponding generally to the major Scenarist functions discussed in the top-level design (i.e., to the program's menu options). In cases in which a function is used (or "called") by more than one major function, it is associated with the function category in which it was first used.

¶The functions listed below perform many of the higher-level Scenarist system functions, i.e., those that correspond to the higher-level flow diagrams of a data-flow-diagram description of the system. No attempt is made in this final report to describe all of the Scenarist's lower-level functions (such as the variety of graphics, mouse, and printer functions used by the program). These functions are identified and documented in the Programmer's Manual. The emphasis here is in providing a conceptual understanding of what the major system functions do.

Program Setup

Program entry point: Main. When the Scenarist begins execution, a function called "Main" is executed. This function calls a number of other functions required to "set up" the Scenarist system i.e., to establish an interface with CLIPS, open files, read data from files, and transfer control to the `_main_menu_selection` function. The functions called by Main are the following:

- `_init_clips`, establishes the interface of the Scenarist with the CLIPS system

- `_setup_screen_windows`, sets up the initial screen of the Scenarist

- `_display_intro_screens`, presents information about the Scenarist system

- `_project_selection`, prompts the user to select a Scenarist "project file," containing the names of the map, unit, and other files to be used in the run

- `_getfilenames`, reads the project file, stores the names of the files named in the project file in memory, and stores the data in many of these files in memory. The major subordinate functions of `_getfilenames` are:

 - `_readmapheader`, reads the map headers from the cellular map files and stores them in memory

 - `_readmapdata`, extracts 32-cell by 32-cell maps from cellular map files and stores them in memory

 - `_resetmaplocpoint`, resets the map location point

- `_main_menu_selection`, the main menu control function

Menu Control

Program entry point: `_main_menu_selection`,
called by Main

Major subordinate functions:

`_About`, provides information about the
Scenarist system

`_Units`, entry point for functions that operate
on units

`_Rules`, entry point for a CLIPS rule editor
(f
ut
ur
e

de
ve
lo
pm
en
t)

`_Map`, entry point for functions that draw maps
`_Scenario_Generation`, entry point for
(essentially noninteractive) application of
rules to reposition subunits in a large number
of units (future development)

`_Hardware_Setup`, provides information about
the user's hardware system

`_Quit`, which performs certain
end-of-processing functions (e.g., closing
files) and exits (terminates) the Scenarist
program

Project Selection

Program entry point: `_project_selection`,
called by Main

Major subordinate functions:

None

Initial Map Location Point

Program entry point: `_getfilenames`, called by Main

Major subordinate functions:

`_resetmaplocpoint`, called by `_getfilenames` in the program setup steps, resets the map location point

System Description

Program entry point: `_About`, called by Main

Major subordinate functions:

None

Processing Military Units

Program entry point: `_Units`, called by Main

Major subordinate functions:

`_unit_menu_selection`, the menu-generating function that provides options for processing units, by calling the following functions:

`_Define_Unit`, an entry-point function that calls the main unit-definition function, `_defineunit`

`_Copy_Unit`, an entry-point function that calls the main unit-copying function, `_copyunit`

`_Delete_Unit`, an entry-point function that calls the main unit-deletion function, `_deleteunit`

`_Reposition_Unit_by_User`, an entry-point function that calls the main unit repositioning function,

`_repositionunitbyuser`. The main subordinate function called by

`_repositionunitbyuser` (and the functions `_repositionsubunitsbyuser` and `_repositionsubunitsbyrules`) is `_repositionprogeny`. The function `_repositionprogeny` repositions all subordinate units when a specific unit's position is changed or the relative positions of its subordinate units are modified.

`_Reposition_Subunits_by_User`, an entry-point function that calls the main user-controlled subunit repositioning function, `_repositionsubunitsbyuser`

`_Reposition_Subunits_by_Rules`, an entry-point function that calls the main rule-controlled subunit repositioning function, `_repositionsubunitsbyrules`. This function in turn calls a "suitability" function -- either `_suitabilityxxxx` or `_clipssuitabilityxxxx` (where `xxxx` is a project/subproject identifier), an action function `_actionxxxx`, and a variety of other functions that compute factors used in rules.

`_Reposition_FEBA`, an entry-point function that calls the main FEBA-repositioning function, `_repositionfeba`

`_Display_Unit`, an entry-point function that calls the main function used to display units, `_displayunit`. The main subordinate functions called by `_displayunit` are:

_getunitbycode, which accesses a unit in a unit file (either a generic unit file or a specific unit file) by the unit's code
 _getunitbyidno, which accesses a unit by its idno
 _getunitbyindex, which accesses a unit by its index (rank order in the file)
_Output_Units, an entry-point function that calls the main function used to output the units stored in the generic and specific unit files,
_outputunits

Rules

Program entry point: _Rules, called by Main

Major subordinate functions:

None

Processing Maps

Program entry point: _Maps, called by Main

Major subordinate functions:

_map_menu_selection, the menu-generating function that provides options for creating maps, by calling the following functions:

_Draw_Terrain_Map, an entry-point function that calls the main map-drawing function, _drawmap, with the calling parameters (arguments) set to draw the 32-cell by 32-cell terrain-type map that is stored in memory. The function _drawmap draws both cellular maps and vector maps. The main subordinate functions called by _drawmap are:

`_label`, which draws map labels
`_legend`, which draws map legends
`_Draw_Elevation_Map`, an entry-point function that calls the main map-drawing function, `_drawmap`, and instructs it to draw the elevation map that is stored in memory.

`_Draw_Road_Map`, an entry-point function that calls the main map-drawing function, `_drawmap`, and instructs it to draw the road map that is stored in memory. In problem 02 (TRAILBLAZER), this function also provides the user with the option of drawing a TRAILBLAZER accessibility map, after the `Reposition Subunits by Rules` function has been called.

`_Add_Vector_Map_with_Labels`, an entry-point function calls the main map-drawing function, `_drawmap`, with the calling parameters set to add (superimpose) all of the geographic objects stored in the area object, linear objects, and point objects files to the map currently on the screen, with labels printed for all objects.

`¶_Add_Vector_Map_without_Labels`, an entry-point function calls the main map-drawing function, `_drawmap`, with the calling parameters set to add (superimpose) all of the geographic objects stored in the area object, linear objects, and point objects files to the map currently on the screen, with no labels printed for any object.

`_Draw_Vector_Map_without_Labels`, an entry-point function calls the main map-drawing function, `_drawmap`, with the calling parameters set to draw a (vector) map containing all of the geographic objects stored in the area object, linear objects, and point objects files, with no labels printed for any object.

`_Place_Unit_on_Map`, an entry-point function that calls the main unit-drawing function, `_drawunit`, and draws a unit on the map currently on the screen (or on a blank screen if no map is currently on the screen). The function `_drawunit` call the function `_symbol` for drawing item symbols and printing item labels.

`_Zoom_Map`, an entry-point function that calls the function `_setcoordsforzoommap`, that sets the values of certain global variables accessed by the function `_drawmap` so that the maps it draws will be zoomed (i.e., will cover a smaller area)

`_Change_Map_Location`, an entry-point function that calls the functions `_readmapheader`, `_readmapdata`, and `_resetmaplocpoint` to input a 32-cell by 32-cell cellular map from a different part of the maps stored in the current cellular map files.

`_Change_Map_Files`, a function that enables allows the user to specify new map files, and then calls the functions `_readmapheader`, `_readmapdata`, and `_resetmaplocpoint` to input 32-cell by 32-cell maps from the new files.

`_Print_Map`, an entry-point function that calls the function `_printscreen`, which prints the contents of the screen (in graphics mode) on a Hewlett-Packard series II laser printer.

Scenario Generation

Program entry point: `_Scenario_Generation`, called by Main

Major subordinate functions:

None

Hardware Data

Program entry point: `_Hardware_Setup`, called by Main

Major subordinate functions:

None

¶The above list of major Scenarist functions is about a third of all of the functions in the Scenarist program. The rest are lower-level functions that perform various processing or input-output operations; as mentioned, those functions will not be described in this report, since their description would add little to an understanding of how the system works.

Having now identified the major variables, files, and functions of the Scenarist software subsystem, we shall now proceed to describe those components.

B. Detailed Description of Scenarist Program Variables

1. Map-Related Data Structure Types

There is a single map-related data structure type, called "mapinfo," for cellular maps, and several more data structure types for vector maps. The definition of the mapinfo data structure type, in C notation, is as follows. Definitions of the members of the structure (e.g., filename, mapname) will be presented in the description of the map files, later on.

(Note: variables are defined in the Scenarist program in two places -- in "header" files (called "include" files, in C terminology) and at the beginning of functions. "Global variables," whose values may be accessed by any function, are defined in the header files. "Local variables," whose values are known only to a specific function, are defined at the beginning of that function. The variables discussed in this final report are all global variables, and so their definitions may be found in the header files of the Scenarist program. The MS-DOS system names for these header files are of the form xxxxxxxx.h, whereas the names of the C-language functions are of the form xxxxxxxx.c (where the x's refer to a legal file name prefix of up to eight characters).)

```
struct mapinfo{
    char        filename[13];
    char        mapname[80];
    char        datatopic[80];
    char        datasource[80];
    char        creationdate[80];
    char        changedata[80];
    int         ncats;
```

```

int      codevalue[20];
char      codename[20][80];
char      unit[80];
int      utmzonenumber;
char      utmrowletter;
float     xmin,ymax;
float     xmax,ymin;
float     cellwidth;
int      rows,cols;

char      format[80];
float     xminwindow,ymaxwindow;
float     xmaxwindow,yminwindow;
int      rowwindow,colwindow;
};

```

The data structure types related to vector maps are areaobject, linearobject, and pointobject. These structure types are defined in terms of a more elementary data structure type, called coord. These data structures are defined in C notation as:

```

struct coord{
float      x,y;
};

struct areaobject{ /* irregular area
objects */
/* must be drawn counterclockwise */
/* for zoom to find correct interior point
*/
/* objects drawn on map in order in file
*/
char      name[17];
int      printind;

```

```

    int                areaobjtype; /* filltype */
    int                index;
    struct coord       br1,br2; /* bounding
rectangle corners */
    struct coord       intpoint, namepoint; /*
interior point,
                                name point */
    float              area, magnitude, value,
meanel, vare1;
    int                nopts;
    struct coord       vertex[20];
};

    struct linobject{ /* piecewise linear
objects */
    char               name[17];
    int                printind;
    int                linobjtype;
    int                index;
    struct coord       br1,br2;
    struct coord       namepoint;
    char               origobj[5],destobj[5];
    float              length, width, capacity,
flowrate, flowdir,
                                value;
    int                nopts;
    struct coord       vertex[85];
};

    struct pointobject{ /* point objects */
    char               name[17];
    int                printind;
    int                ptobjtype;
    int                index;

    struct coord       namepoint;

```

```
struct coord    location;  
float          magnitude,value;  
};
```

2. Map-Related Variables

The following is a list of the major map-related variables of the Scenarist.

- a. mapinf: data structure (of data structure type mapinfo) used to store the data contained in the header of a cellular-data map file
- b. geodisc: array used to store the values of the cells of a discrete-variable cellular map (terrain-type, roads)
- c. geocont: array used to store the values of the cells of a continuous-variable cellular map (elevation)
- d. areaobject: data structure (of data structure type areaobject) used to store the attributes of a geographic area object (e.g., urban area, body of water). The attributes of a geographic area objects are read from a geographic area objects file.
- e. linobject: data structure (of data structure type linobject) used to store the attributes of a geographic linear object (e.g., road, river)
- f. ptobject: data structure (of data structure type pointobject) used to store the attributes of a geographic point object (e.g., mountain peak)
- g. attributelabels: array used to store labels for categories of cellular maps or the attributes of objects of vector maps. The values for the elements of this array are

initialized at the beginning of the Scenarist program execution (in a C "header" file)

3. Unit-Related Data Structures

In the current version of the Scenarist, there is but one data structure type used for all types of units of every echelon level. Different data structure types could have been used for units of different complexity (specifically, of different echelon levels), but it was decided to use a single data structure type for all of them. This is a little inefficient from the point of view of memory utilization (because not all units possess all attributes of the data structure type), but it significantly simplifies the program complexity.

¶The same data structure type is used both for generic units and for specific units. Consideration was given to using a simpler data structure type for specific units, but little saving in file storage would have been achieved at the expense of a considerable increase in file access. The reasons for this are that it is necessary to specify all of the relative locations of all subordinate units in a specific unit (and this consumes a lot of the unit storage area) and it would be necessary to access the generic unit to determine the values of needed items not stored in the specific unit (of the same echelon and type).

The unit data structure type is denoted as "unit." It is defined in terms of a number of

variables and other data structure types. These other data structure types are not used in the processing (i.e., no variables are declared as being of these types). They are used to simplify the definition and understanding of the unit data structure type, by building it up from simpler structures.

In the definition of the data structure type presented below, a number of descriptive "comments" are made alongside various members of the structure, and at various points in the structure definition. These comments are surrounded by a delimiter pair that begins with the "token" /* and ends with the token */.

The unit data structure type includes the structure types unitid, subid, unitatt, and itematt. The unitatt and itematt data structure types include the data structure type subid. All of these data structure types are defined in the following order: unitid, subid, unitatt, itematt, and unit.

The unit data structure type is intended for use with military items of all echelon levels -- both units (echelons 1-11) and nonunit items (echelons 12-13, i.e., platforms and equipment). During the course of the Scenarist development, all of the items that were defined (in the Beqaa Valley and Spearfish/TRAILBLAZER examples) were in fact units; no platforms or equipments were defined. Note that the term "definition" means that the Scenarist program was used to specify the attributes of a unit and write the specified unit in either the generic unit file or the

specific unit file. Also, note that only units (either generic units or specific units) are stored in the (generic or specific) unit files. Nonunit items (platforms or equipment) are defined generically, and are stored in the platform and equipment files. Since no platforms or equipments were defined in the development test cases, no entries were made in either the platform file or the equipment file.

¶It may happen that when the Scenarist is applied to an application involving a considerable amount of equipment that a need will arise to define a different data structure type for platforms and equipment, in order to save file space (in the platform and equipment files), even though only a single generic version of each type of platform or equipment is stored. Each variable of data type unit requires 4000 bytes of memory, and each stored item of the same format requires 4000 bytes of file space. The amount of memory used by variables of data type unit is not much because there are very few such variables, but the amount of file space used to store units or items of this format could be considerable.

Although defining a different data structure type for units (items of echelons 1-11) would have a detrimental effect on program complexity, defining a different data structure for platforms and equipment (echelons 12-13) would have little effect on program complexity, since these items are defined generically, not specifically. Although positions for them are specified in a defined unit, they are not

themselves "defined." They are not placed as specific units on the map, and no repositioning of their subordinate items is allowed.

In the current version of the Scenarist, no data structure types have been defined for platforms or equipment. The data structure type definitions would be made when the user having an application involving platforms and equipment decides what characteristics he wishes to store in the platform and equipment files on those items.

Note that in the unit data structure, information on the subunit positions is stored there twice -- once in a nonredundant form (for possible inclusion of unit data in a relational data base management system) and once in a form that facilitates access to each subunit's data.

The unit data structure type will now be defined (along with the simpler data structure types in terms of which it is defined).

```
struct unitid{
    short          side,echelon,type;
    /* Unit kind. Used for both generic and
specific units */
    short          number,idno,parentidno;
    /* Used for specific units only (0,0 for
generic units).*/
    short          code[13],parentcode[13];
    /* 13 components correspond to side, army,
corps, div, bde, rgmt, btn, co, plt, sec,
squad, platform, equipment. Used for generic
and specific units. Note: code[0] = side
```

```

    (redundancy for convenience -- use side when
    side is treated as a dimension independent of
    echelon. Use code[0] when side is used as
    highest-level echelon descriptor of a unit in
    the code[] identifier). */
};

struct subid{
    short                echelon,type; /* Subunit
kind. */

    ¶short                number,idno;
    /* Used for specific units only (0 for generic
units). */
};

struct unitatt{
    /* Specific attributes of an area subitem
(units, geogtypes 1-6) */
    /* All of a specific unit's specific attributes
that are needed by the program for analysis
(not for editing or reversion to the standard
form), and are stored in the specific unit
file. (A specific unit's generic attributes
(e.g., symbol, label) are stored in various
matrices in symbolfiln, accessible by all
specific units of the same side, echelon, and
type.) */
    struct subid        id;
    float                corners[4][2];
    /* For specific units, these are map
coordinates. For generic units these are
(0,0), (1,0), (1,1), (0,1). */
    float                loc[2];
    /* location for symbol, or for analysis */
    float                rad;

```

```

short          geogtype;
/* geogtype indicates the geographic type of
a subunit, with respect to the specification
of the unit's geographic layout by the user
when defining a unit.  The geogtype is stored
to determine what parts of a parent unit must
be respecified, if the position of a subunit
is changed.  It is not used in drawing a unit.
A unit is drawn by drawing all of its subitems
independently.  All of the data needed to draw
the subitems are contained in struct unitatt
or struct itematt.*/
/* Definition of geographic type code:
Areas defined by 4 corners (relative
coords):
    1. on-line (front) subunits
    2. reserve (rear) subunits
    3. all-area subunits
    4. major area subunits
Areas defined by location (relative)
    + radius (relative)
    5. minor area subunits (relative)
Areas defined by location (relative)
    + radius (absolute, in meters)
    6. minor area subunits (absolute)
Non-area items (defined by location only)
    7. point subitems

```

¶Notes: Units (echelons 1-9) may be of geographic types 1-6. Only units may have specific subunits. Type 7 is reserved for use by platforms (echelon 10) and equipments (echelon 11) only. "Point subitems" (#7) are generic items (i.e., their attributes are a function of the side, echelon, and type, not the individual item). (The term "point

subitem" is a little misleading: e.g., a platform may possess an area, but it is a generic characteristic.) Platform point subitems may have generic subunits, but not specific subunits. Equipment point items have no subunits (either generic or specific). The generic characteristics of platforms, including a list of up to 9 equipments and other attributes, are stored in the platform file. Note that the location is not included (it is a specific characteristic, stored only in the parent unit.) The generic characteristics of equipments are stored in the equipment file (no location data). */

```
short          placementcode;
/* Placement code. The placement code indicates how the location of a subitem was determined:
```

```
    0: Canonical placement (the placement code is initialized to 0 for all defined
```

```
units)
```

```
    1: User-reviewed, unchanged
```

```
    2: User-suggested placement
```

```
    3: User-mandated placement
```

```
    4: Position reviewed by rules, whether moved or not
```

```
    5: Position reviewed by rules, moved to satisfy local constraints
```

```
    6: Subitem of geogtype 6 was moved in
```

```
a
```

```
search to satisfy global
```

```
constraints
```

```
    7: Subitem of geogtype 7 (platform/equipment) was moved in a search to satisfy global constraints */
```

```

};

struct itematt{
  /* Specific attributes of a point subitem
  (platforms or eqpt, geog type 7)*/
  struct subid      id;
  float             loc[2];
  short             placementcode;
};

```

```

struct unit{
  struct unitid     id;
  char              name[17];
  float             corners[4][2];
  /* For specific units, these are map
  coordinates. For generic units these are
  (0,0), (1,0), (1,1), (0,1). These corner
  coordinates are not needed in the generic unit
  file (since they are the same for all generic
  units); the locations are nevertheless defined
  to permit copying of the complete data
  structure to a specific unit record (which does
  use them) */

```

¶/* What follows next is the editable form of a unit specification. It is normalized (no redundancy) for possible future inclusion of unit data in a relational data base management system. This structure is used to edit a unit and to reset a specific unit to its standard position, after a move. The editable form is designed for retrieval and editing of unit data, not for speed of access during analysis. (The major reason for normalization in the current application is to insure that if the

boundaries of one unit are changed, the boundaries of neighboring units are correspondingly modified.) Note: A potential dependency exists in the data base, viz. the subunit boundaries are derived from the parent unit boundaries, and then stored in the subunit record. There is no dependency if the subunit position is viewed as the position corresponding to the initially specified position of the parent unit. In general, however, a user would wish for the subordinate unit boundaries to be consistent with the current parent unit boundaries. To insure this, a function (`_repositionprogeny`) will be executed every time a unit is repositioned, to insure that all successor unit boundaries are consistent. By automatic (program) control of redundancy (the occurrence of unit boundaries in both the parent and the subunit) and dependencies (the matching of boundaries of neighboring units), the integrity of the data base is assured. */

```

float      loc[2];
/* location for symbol, or for analysis */
float      rad;
float      flankp[2];
short      nfront;
/* geographic type 1: on-line (front)
subordinate units */
struct subid frontid[5];
float      frontp[4][2];
short      nrear; /* 2: reserve (rear)
subunits */
struct subid rearid[5];
float      rearp[4][2];

```

```

    short                nallarea; /* 3: all-area
subunits */
    struct subid        allareaaid[5];
    float                allareasymloc[5][2];
    short                nmajorsubarea; /* 4: major
subarea subunits */
    struct subid        majorsubareaaid[5];
    float
majorsubareacorners[5][4][2];
    short                nminorsubarearel; /* 5:
large minor subarea
subunits (relative radius) */
    struct subid        minorsubarearelid[5];
    float                minorsubarearelloc[5][2];
    float                minorsubarearelrad[5];
    short                nminorsubareaabs; /* 6:
small minor subarea
                ¶ subunits (absolute radius) */
    struct subid        minorsubareaabsid[5];
    float                minorsubareaabsloc[5][2];
    float                minorsubareaabsrad[5];
    short                npoint; /* 7: point (nonarea)
subitems */
    struct subid        pointid[30];
    float                pointloc[30][2];
/* What follows next is the form of the unit
record that is designed for rapid retrieval of
subunit position (location, boundaries)
information during analysis, and simplicity of
drawing of units (since the subunits of a unit
may each be drawn independently). It is not
normalized. Instead, it contains redundancy
(i.e., the boundaries of bordering units are
repeated, so that each subunit is stored as an
independent unit in the parent record) to

```

```

enable fast access. Without this redundancy,
it would be necessary to completely reprocess
the "editable" form of the record, in order to
determine a subordinate unit's boundaries from
the parent unit. Because the following form
is not normalized, it would not be stored in
an RDBMS, but instead would be reconstructed
by the program after retrieving the editable
form from the RDBMS. */
/* Store positions of subunits with parent
unit, to allow for rapid plotting (so don't
have to define or access subordinate unit
record, and don't have to compute boundary
corners from input record). If and when a
subordinate unit is defined, its position is
copied from its parent record, and stored,
redundantly, in the subunit record, to avoid
need to access parent record for location.
User cannot change the position of a subunit
in a subunit record -- the location can be
defined only in its parent unit (and only in
the editable form, not here in the access
form). If user changes the position of a unit
(either of a unit that has no parent, or of a
subunit in a unit), the program automatically
updates the positions here and in all defined
subunit records (and subunits of subunits,
etc.). */
short                nsubunits; /* no of items
of geog type 1-6                (units) */
struct unitatt  subunitatt[30];
short                npointsubitems; /* no of
items of geog type                7
(platforms, equipment) */
struct itematt  subitematt[30];
};

```

4. Unit-Related Variables

The following is a list of the major unit-related variables.

a. genericunit, specificunit, unitblank, unit0, unit1: data structures (of data structure type unit) used to store the attributes of a military unit

b. code, parentcode: arrays (one-dimensional arrays, i.e., vectors) used to store a unit's code

c. symbols: array used to store the symbol number of each type of unit

d. labels: array used to store the labels for each type of unit

5. FEBA-Related Variables

a. feba: an array used to store the locations of the points defining the FEBA

C. Detailed Description of Scenarist Data Files

1. Listing of the Scenarist Data Files

The Scenarist program uses about a dozen files on the computer disk to store map, unit, and other system-related data. Each file has an MS-DOS file name, according to which the MS-DOS system keeps track of it. The Scenarist program identifies files by use of a file name variable and a file pointer variable. The MS-DOS file name is stored in the file name variable, and the

file pointer contains an address of a data structure containing information about the file.

Whenever a user applies the Scenarist to a new application, he will create and modify a new set of data files. During a particular application, he will create and modify additional files, such as map files of varying resolutions. It is important to keep track of the data files both between applications and within applications. In order to assist the user in keeping track of files, a particular file naming system has been proposed for Scenarist files. Each file of a particular type (e.g., a cellular-map file) is given a unique four-letter prefix, which forms the first four letters of the file name. The next two letters specify the application number (e.g., in the Scenarist development the Beqaa Valley example was given the number 01 and the TRAILBLAZER example was given the number 02). The next two letters (optional) specify the sequence of files of a particular type (e.g., if there are two terrain-type maps of different resolutions, the first one would correspond to 01 and the second to 02). These letters are optional because for some file types, there is never more than a single file. To these letters the suffix ".fil" is appended.

The files used by the Scenarist program are identified below. Each file is identified by a descriptor, the Scenarist file name variable, the Scenarist pointer variable, and an example.

¶project	file:	filenamesn,	filenamesp,
proj0101		fil	

terrain-type map: geodfiln, geodfilp,
geod0101.fil
elevation map: geocfiln, geocfilp, geoc0101.fil
road map: georfiln, georfilp, geor0101.fil
geographic area-objects map: geoafiln, geofilp,
geoa01.fil
geographic linear-objects map: geolfiln,
geolfilp, geol01.fil
geographic point-objects map: geopfiln,
geopfilp, geop01.fil
generic unit file: genunitfiln, genunitfilp,
genu01.fil
specific unit file: specunitfiln, specunitfilp,
spec01.fil
scratch file: scratchfiln, scratchfilp,
scra01.fil
symbol file: symbolfiln, symbolfilp, symb01.fil
platform file: platformfiln, platformfilp,
plat01.fil
equipment file: eqptfiln, eqptfilp, eqpt01.fil
FEBA file: febafiln, febafilp, feba01.fil
introduction file: introfiln, intorfilp,
intro.fil
summary information file: introfiln, introfilp,
highlits.fil
CLIPS rule file: clipfiln, clipfilp, clip01.fil
CLIPS output file: dribble.txt
output file: outputfiln, outputfilp,
out0101.fil

Of these files, the user has no occasion to use the scratch file, the introduction file, or the summary information file. The generic unit and specific unit files are binary files that cannot be printed meaningfully. The file structure of these files is very simple -- each file consists

of a number of records whose record layout (format) corresponds exactly to the "unit" data structure type format. The data are written to these files and read from them using the "block" input/output functions fwrite and fread. That is, reading and writing is accomplished in "blocks" that correspond to a single data structure of the "unit" data structure type.

The platform and equipment files have not yet been used, and they may or may not be binary files, depending on how they are implemented in the first application that uses them. The information in the generic unit and specific unit files is formatted and transferred to the output file through use of the function Output Units.

The CLIPS rule file and CLIPS output file are not discussed in this report (their content and format are discussed at length in the CLIPS documentation).

The output file is a text file whose contents are simply concatenated formatted versions of the unit data structures stored in binary form in the generic unit and specific unit files.

¶The remaining files (the project file, the six map files, the symbol file, and the FEBA file) may be generated manually by the user using a line editor, or created by the user using other software. They are in text format, and may be printed for review by the user, if desired. These files are read by the Scenarist, and the data in them must be of a particular content,

order and format. This section describes the content and format for these nine files.

2. Map Data Files

a. Cellular Map Files

The cellular map files used in the Scenarist are of two types -- those that store "discrete," or categorical, data, and those that store "continuous" data. The two types of categorical map data accepted by the Scenarist are terrain-type map data and road map data. The single type of continuous data accepted by the Scenarist is elevation data.

The same format is used to store both categorical-data and continuous-data maps. For both maps, the value in a particular cell is stored as an "integer" data type, using two bytes of storage.

The content and format of the cellular map files is specified below. The description of both content and format is presented in English. Terms not included in brackets should be typed as is, but excluding the number and period at the beginning of each line (e.g., 1., 2.). These terms are read by the program up to the colon (the information in these terms is not used by the program). The brackets contain the content and format description of data that will vary from map to map. Examples of these data files are included in the program listing provided with the Scenarist Programmer's Manual.

The cellular-data map files consist of two parts -- a "header" containing descriptive information about the file, and the map data. The program reads in the headers from the terrain-type and elevation map files before reading the map data from these files, and before reading any data from the road file (if any data in fact are present in the road file). The maximum amount of data read from a cellular map file is a 32-cell by 32-cell square map.

¶The map headers are stored in the variable `mapinf[][]` of data type `mapinfo`. It is declared (defined in the C language) as "struct mapinfo mapinf[2][3]." Symbolically, this data structure is represented as `mapinf[map type][map index]`. The first dimension refers to the type of cellular maps -- discrete-variable maps (dimension value 0) and continuous-variable maps (dimension value 1). The second dimension refers to a particular map of the type specified by the first dimension. The variable `mapinf[0][0]` refers to the terrain-type map, `mapinf[0][1]` to the road availability map, `mapinf[0][2]` to the TRAILBLAZER accessibility map, and `mapinf[1][0]` to the elevation map. (Note: vector dimensions in the C programming language start at zero, and brackets are placed around each dimension.)

The categorical-map data are stored in an integer array (matrix) variable declared as "int geodisc[3][32][32]." The variable `geodisc[0][][]` contains the terrain-type map data, the variable `geodisc[1][][]` contains the road map data (or the no-data code 0 in all

elements if there are no road data), and the variable `geodisc[2][][]` contains the TRAILBLAZER accessibility map data. The continuous-map data are stored in a map declared as `"int geocont[1][32][32]."` The variable `geocont[0][][]` contains the elevation map data.

Having defined the structure type used for the map file header and the array variables for the map data, we shall proceed to define the file content and format. The content and format of the cellular map file header corresponds closely to the content and format of the `mapinfo` data structure defined above. They are not exactly the same, since the file is generated manually using a line editor rather than by reading data structures of data structure type `mapinfo` into the file, and because a number of line descriptors (e.g., File Name) are included in the file but not in the data structure.)

Data from the map files is read in "free format." That is, the user may input spaces, carriage returns, or line feeds at will in the file. The program reads each successive variable separated by these codes and stores them in memory in the storage location of the corresponding variable of the data structure corresponding to the file. (Note: any of the Scenarist files that may be generated manually (i.e., using a line editor) are read by the program in free format.)

1. File Name: [File name, up to 12 characters, no imbedded blanks. Example: `geod0101.fil`]
2. Map Name: [Map name, up to 79 characters, no imbedded blanks. Example: `Beqaa_Valley`]

3. Data Topic: [Name of data type, up to 79 characters, no imbedded blanks. Example: Terrain_type]

4. Data Source(s): [Names of sources of map data, up to 79 characters, no imbedded blanks. Example: Rand-McNally_Atlas]

5. File Initial Creation Date: [Date on which file was created, up to 79 characters, no imbedded blanks. Example: June_15,_1990]

6. Date of Last Change: [Date on which file was last changed, up to 79 characters, no imbedded blanks. Example: January_26,_1991]

7. Legend:

8. Number of categories: [For continuous data enter "0". For categorical data, enter the number of categories, an integer between 1 and 7. Example: 7]

9. [For continuous data, enter:] Unit of measurement: [Name of unit of measurement, up to 79 characters, no imbedded blanks. Example: meters]

[For discrete data, enter, for each category, a one-line description of each numerical code to be used (an integer between 0 and 6), followed by a colon, followed by a description of the category (up to 9 characters, no imbedded blanks). Example (for 7 data categories):

0: No_data

1: Plains

2: Woods

3: Hills

4: Mts

5: Water

6: Urban

Note that the value "0" is always interpreted by the Scenarist to mean "no data." The numerical codes are used as indices in arrays, and must be integers between 0 and 6 (other values will be replaced by the "no data" code value, 0). (Note: if the Scenarist is developed further, consideration should be given to allowing the user to input any seven different code values, and having the Scenarist recode them to the values 0-6. This modification would enable the user to continue to use code values with which he may be more familiar.)

10. UTM Zone: [enter UTM zone number, an integer, or "0"] UTM Band: [enter UTM row letter, a character, or "0"] [If "0"s are entered, the map coordinates are not relative to any UTM zone or band, but to a user-specified coordinate system.]

11. Map top left corner, in UTM coordinates: [no data]

12. East-west (meters): [x coordinate, in meters, of the top left corner of the map, decimal value. Example: 0]

13. North-south (meters): [y coordinate, in meters, of the top left corner of the map, decimal value. Example: 72000]

14. Cell width (meters): [width (and height) of the map cells, in meters, decimal value. This is usually referred to as the "resolution" of the map. Example: 2000]

15. Rows, columns: [The number of rows and columns of the map to be stored in the file. Two integers, greater than zero. A 32-cell x 32-cell map will be extracted by the Scenarist from this map. The number of rows and columns can be less than, equal to, or greater than 32.]

16. Row format: [This is descriptive information for the user; it is not used by the Scenarist. Up to 79 characters, no imbedded blanks. The map data should be stored row by row, as integer values with a space between each value. Example: (68I2)]
17. [The map data, stored row by row, as integer values with a space between each value.]

b. Vector Map Files

There are three types of vector map files used by the Scenarist. These map files contain data to provide background information to provide a frame of reference for the user in working with units on the screen. The three types of vector maps are area-object maps, linear-object maps, and point-object maps.

The vector map files are read whenever a vector map is drawn. The cellular-map data are read from a file and stored in memory until the end of a Scenarist run or until the user reads a new map in from another file. Unlike the cellular-map data, vector-map data are not stored in memory. As the data for each map object is read from a vector-map file, it is used to draw a representation of the object on the screen, but it is not retained. The map-object data are retained in memory (i.e., in a variable location) just so long as the representation is being drawn. As soon as the next object is read from the file, the information in memory about the preceding object is destroyed (replaced).

The vector-map data were not stored in memory because it was not clear how much memory would be required. We had firm control over the amount of cellular-map data stored in memory, because only 32 cell by 32 cell portions of cellular map files were read from cellular map files (even though the cellular map file could be much larger). For vector maps files, however, the all of the objects in the map file were to be drawn on the screen. Since these files were of indeterminate size and it was not desired to place a limit on how many objects from the file would be drawn, the objects were read from the file one by one and drawn, and no attempt was made to store any of the vector objects in memory. (In a possible extension of the Scenarist, the program could be modified to store a small number of vector map objects in memory. This would speed up the process of drawing a vector map with a relatively small number of objects, since no file access would be required after the first such map was drawn (and the objects had been stored in memory).

¶The content and format of the vector map files will now be described. The content (variables) and order of the vector map file records is exactly the same as the content (components) and order of the corresponding data structure into which the vector map file data are read in memory, i.e., areaobject for the area-objects file, linobject for the linear-objects file and ptobject for the point-objects file. The format is not exactly the same, however, since the vector map file was prepared manually using a line editor rather than by writing data

structures into the file. The variable types and order are the same for the file as for the corresponding data structure in the program, but, as noted earlier, the vector map files are read in free format, so that the user may input spaces, carriage returns, and line feeds between the variables.

Area-Object Vector Map Files.

The format and content of the data elements corresponding to each object stored in the geographic area-objects vector map file is as follows:

1. [Object name, 16 characters, no imbedded blanks] [Print indicator variable: 0 = don't print name; 1 = print name]
2. [Area object type: 0 = reserved; 1 = plains; 2 = hills; 3 = woods; 4 = mountains; 5 = urban area; 6 = body of water]
3. [Index: order of object in the file, integer, 1, 2, 3,.....]
4. [Coordinates of bounding rectangle. The map coordinates of two corners of a rectangle that just contains the object. Four decimal numbers, x1, y1, x2, y2, in meters. If no part of the bounding rectangle falls on the map, no attempt is made to draw any part of the object.]
5. [Coordinates of any point in the interior of the object. Two map coordinates, decimal numbers, x, y, in meters.]
6. [Coordinates of the point at which the name is to start. Two map coordinates, decimal numbers, x, y, in meters.]
7. [Object attributes: any five numerical attributes the user wishes to specify for the

object. Suggested attributes: area, magnitude, value, mean elevation, variance of elevation. Five decimal numbers.]

8. [The number of points, npts, to be specified on the boundary of the object. Two map coordinates per point, decimal numbers, integer between 3 and 20.]

9. [The npts pairs of coordinates for the npts points defining the object boundary. npts pairs of map coordinates, decimal numbers, meters, followed by 2(20-npts) zeros (for a total of 40 numbers in all.)

The above format is repeated for as many area objects as are in the area-object file. The file is read until an end-of-file condition occurs.

The area-object map data are read into a variable of data structure type areaobject, declared as "struct areaobject areaobject[20]." (The dimension is 20 since it was originally planned to store as many as 20 area objects in memory; only one is stored in the current version of Scenarist.)

Linear-Object Vector Map Files.

¶The format and content of the data elements corresponding to each object stored in the geographic linear-objects vector map file is as follows:

1. [Object name, 16 characters, no imbedded blanks] [Print indicator variable: 0 = don't print name; 1 = print name.]

2. [Linear object type: 0 = reserved; 1 = roads; 2 = bridges; 3 = rivers; 4 = coast; 5 = border; 6 = reserved.]
3. [Index: order of object in the file, integer, 1, 2, 3,....]
4. [Coordinates of bounding rectangle. The map coordinates of two corners of a rectangle that just contains the object. Four decimal numbers, x_1 , y_1 , x_2 , y_2 , in meters. If no part of the bounding rectangle falls on the map, no attempt is made to draw any part of the object.]
5. [Coordinates of the point at which the name is to start. Two map coordinates, decimal numbers, x , y , in meters.]
6. [A descriptor of an object at the origin of the linear object (the first point in the sequence of points defining the linear object). Up to four characters, no imbedded blanks.]
7. [A descriptor of an object at the destination of the linear object (the last point in the sequence of points defining the linear object). Up to four characters, no imbedded blanks.]
8. [Object attributes: any six numerical attributes the user wishes to specify for the object. Suggested attributes: length, width, capacity, flow rate, flow direction, value. Five decimal numbers.]
9. [The number of points, $npts$, to be specified along the object. Two map coordinates per point, decimal numbers, integer between 2 and 85.]
10. [The $npts$ pairs of coordinates for the $npts$ points defining the object. $npts$ pairs of map coordinates, decimal numbers, meters, followed by $2(85-npts)$ zeros (for a total of 170 numbers in all.)

The above format is repeated for as many linear objects as are in the linear-object file. The file is read until an end-of-file condition occurs.

The linear-object map data are read into a variable of data structure type `linearobject`, declared as `"struct linobject linobject[10]."` (The dimension is 10 since it was originally planned to store as many as 10 linear objects in memory; only one is stored in the current version of Scenarist.)

Point-Object Vector Map Files.

The format and content of the data elements corresponding to each object stored in the geographic point-objects vector map file is as follows:

1. [Object name, 16 characters, no imbedded blanks] [Print indicator variable: 0 = don't print name; 1 = print name.]
2. [Point object type: 0 = reserved; 1 = placename; 2 = crossroads; 3 = mountain peak; 4 = reserved; 5 = reserved; 6 = reserved.]
3. [Index: order of object in the file, integer, 1, 2, 3,....]
4. [Coordinates of the point at which the name is to start. Two map coordinates, decimal numbers, x, y, in meters.]
5. [Coordinates of the object location. Two map coordinates, decimal numbers, x, y, in meters.]
6. [Object attributes: any two numerical attributes the user wishes to specify for the

object. Suggested attributes: magnitude, value. Two decimal numbers.]

The above format is repeated for as many point objects as are in the point-object file. The file is read until an end-of-file condition occurs.

The point-object map data are read into a variable of data structure type pointobject, declared as "struct pointobject pobject[10]." (The dimension is 10 since it was originally planned to store as many as 10 point objects in memory; only one is stored in the current version of Scenarist.)

3. Symbol File

The symbol file contains a three-dimensional array of symbol numbers, followed by a three-dimensional array of labels. Each number in the symbol array is the symbol number of a military item (unit, platform, or equipment) of a particular side (of which there are 3), echelon (of which there are 13) and type (of which there are 10). Each number in the label array is a label (up to 8 characters in length, no imbedded blanks) of a military item of a particular side, echelon, and type.

The first data in the file are read into the array named "symbols," which is declared as "short symbols[3][13][10]." The symbol numbers are integers from 1 to the number of symbols defined in the Scenarist function named s03rsymb.c. The fastest running index is type, then echelon, then

side. In other words, all of the symbol numbers for side 1 (BLUE) are read in, echelon by echelon (for 13 echelons), from the first type to the last type (for 10 types). Then, all of the symbol numbers for side 2 (RED) are read in, in this same order. Finally, all of the symbols for side 3 (GRAY) are read in.

The remaining data in the file are read into the array named "labels," which is declared as "char labels[3][13][10][9], in the same order as described above for the symbol matrix.

An example of the symbol file is provided in program and file listing accompanying the Programmer's Manual. The symbol file is stored in free format. In the symbol file listing submitted with the Programmer's Manual, each line contains each of the ten symbol types or labels corresponding to a particular side and echelon.

4. FEBA File

The FEBA file contains two entries, in free format: the number of points on the FEBA (an integer between 1 and 5), and two coordinates (map coordinates, decimal numbers, meters) for each point. The data are read into the array feba, which is declared as "float feba[5][2]."

D. Detailed Description of Scenarist Functions

1. Description of Unit-Processing Functions

a. Define Unit Function, defineunit

The `_defineunit` function enables the user to define a unit, by providing mouse or keyboard responses to a number of questions. User responses to these questions provide data for each of the variables contained in the unit data structure type defined earlier. As the user enters the data, they are stored in memory in a data structure of type "unit." When the user has finished entering all of the data for a unit, the data structure is appended to (written at the end of) either a generic unit file or a specific unit file, according as the user is defining a generic unit or a specific unit.

As mentioned earlier, the generic unit file and specific unit files are binary files into which data structures of type "unit" are written. The process of reading unit data from these files is straightforward, because the file format is exactly the same as the unit data structure format. Unit data are written to and read from the unit files using C's "block i/o" functions, `fread` and `fwrite` (the term "block" refers to the size, in bytes, of the data structure).

The parametric structure of units was described in general terms in the last chapter. Earlier in this chapter, the "unit" data structure that stores all of the unit parametric data was described. The following paragraphs describe the questions (or menu options) posed by the Scenarist program in the process of defining a unit. Each question corresponds to an aspect of the Scenarist's parametric specification of units. From the questions, the level of detail

that the Scenarist maintains on each unit is made quite clear.

¶In the process of defining a unit, the user is required to identify the number and characteristics of the unit's subunits, by geographic type. The Scenarist screen presents verbal descriptions of each geographic type of subunit. To facilitate the process, it may be helpful to refer to Figures 3-8, which contain illustrations of the various unit geographic types.

In the description that follows, some discussion may be presented after each question/menu option. No discussion is presented if the concept was described sufficiently well above. The test deployment data presented as an appendix to the Test Report included as an appendix to this report contain examples of the input that may be entered in response to the unit-definition questions.

"Menu Choice:

Option 1: Define GENERIC UNIT

Option 2: Define SPECIFIC UNIT."

The first choice to be made in the Define Unit function is whether the user wishes to define a generic unit or a specific unit. The recommended procedure in using the Scenarist is to define a generic version of a unit and then to "copy" it to create a specific unit by adding specific information (such as a parent unit identifier, a map location, and an objective). Generic units are stored in the generic unit file identified in the project file, and specific

units are stored in the specific unit file identified in the project file. Over time, as generic units are added to the generic unit file, it becomes a "library" of units that can be copied to form new units without having to go through the definition process.

"Input unit name (up to 16 characters)."

The unit's name may be any descriptive name of up to 16 characters (no blanks). Since the name will be printed on the screen next to the unit, it is best to use short names. Typically, a name would indicate the echelon level and type of the unit, e.g., Division1.

"Unit Code: side, army, corps, division, brigade, regiment, battalion, company/battery, platoon, section, and squad. Input unit code (11 integers)."

¶The unit code is a series of eleven numbers (i.e., an 11-component vector), one for each of the 11 unit echelons of the Scenarist model. The first code number indicates the side to which the unit belongs (e.g., 1 = BLUE, 2 = RED, 3 = GRAY). The second number indicates the army to which the unit belongs (e.g., 1, 2, 3,...). The third number indicates the corps to which the unit belongs, and so on. If the unit being defined is of echelon i , the i -th code number is the unit number, and all later code numbers are zeros. If the unit does not have a superior unit of a particular echelon, the code number for that echelon is zero. In other words, if the unit being defined is of echelon i , then for the first $i-1$ code numbers, the k -th code number specifies

the unit number for the k-th echelon superior unit to which the unit belongs; the i-th code number specifies the unit number for the unit being defined; and all later code numbers (above i) are zero.

The number of a unit can be any number (i.e., it does not have to be the sequential digits 1, 2, 3, ...).

"Input unit type (integer \geq 0)."

As discussed earlier, the "type" attribute is used to distinguish units of the same echelon. The unit type may be any integer between 0 and 9. Since this index is used as an index in storing symbols and labels, it must be an integer between 0 and 9; no other values will be accepted. (Note: As discussed previously, the value "0" should be used only for generic units, to facilitate a planned development to display a generic unit of type 0 whenever the user begins to define a unit of the same echelon structure.)

If the unit being defined is a specific unit, the user must specify a parent-unit code, a unit idno, and a parent idno.

"Input parent-unit's code (11 integers)."

"Unit ID no. ("idno"): Any unique integral number. Input unit id number and parent-unit's id number."

The idno is used to enable the user to retrieve units by a number with which he is familiar. Its use is optional; if it is desired not to use this feature, simply input a zero. If two or more

units are assigned the same idno and the user attempts to retrieve a unit having this idno, the system will retrieve the first such unit in the file.

If defining a specific unit whose parent unit is not defined, read in the map coordinates of the unit's corners. If defining a specific unit whose parent unit is defined, determine the unit's corner coordinates from the specification of the parent unit.

"Parent unit is not defined. Input 4 corners, counterclockwise (8 numbers, x1,y1,x2,y2,x3,y3,x4,y4, map coordinates)."

"Front/rear boundary points: Two points, one on each side of the unit, that define a straight line separating the front and rear areas of the unit. Input front/rear boundary points, p1, p2 (0-1.0)."

"On-line (front) subordinate unit: A subunit occupying a slice of the parent-unit's front area. Input NFRONT = number of on-line (front) subordinate units (1-5)."

¶"Input NFRONT set(s) of [echelon,number,type,idno] (one for each subunit)."

For each of the NFRONT on-line subunits, the user must input the echelon, number, type, and idno of the subunit. If the user is defining a generic unit, the idno may be zero.

"Front-unit boundary points: Two points, one on the parent-unit's front and one on the front/rear boundary, that define the side of a front unit. Input NFRONT-1 sets of front-unit boundary points, p1, p2 (0-1.0)."

"Reserve (rear) subordinate unit: A subunit occupying a slice of the parent-unit's rear area. Input NREAR = number of reserve (rear) subordinate units (1-5)."

"Input NREAR set(s) of [echelon,number,type,idno] (one for each subunit)."

For each of the NREAR on-line subunits, the user must input the echelon, number, type, and idno of the subunit. If the user is defining a generic unit, the idno may be zero.

"Rear-unit boundary points: Two points, one on the parent-unit's rear and one on the front/rear boundary, that define the side of a rear unit. Input NREAR-1 sets of rear-unit boundary points, p1,p2, (0-1.0)."

"All-area subordinate unit: A subunit that covers the entire parent-unit area. Input NALLAREA = number of all-area subordinate units (1-5)."

"Input NALLAREA set(s) of [echelon,number,type,idno] (one for each subunit)."

"Symbol location point: The point at which the subunit's symbol will be displayed. Input

NALLAREA set(s) of symbol location points, x,y (0-1.0)."

"Major subarea subordinate unit: A subunit whose domain is a quadrilateral of arbitrary location and orientation within the parent-unit area. Input NMAJORSUBAREA = number of major subarea subordinate units (1-5)."

"Input NMAJORSUBAREA set(s) of [echelon,number,type,idno] (one for each subunit)."

"Input NMAJORSUBAREA set(s) of four corners, counterclockwise, i.e., NMAJORSUBAREA sets of x1,y1,x2,y2,x3,y3,x4,y4 (each 0-1.0)."

¶"Large minor subarea subordinate unit: A subunit whose domain is a square whose half-width is specified as a fraction of the parent unit width. Input NMINORSUBAREAREL = number of large minor subarea subordinate units (1-5), half-width specified as fraction of unit width."

"Input NMINORSUBAREAREL set(s) of [echelon,number,type,idno] (one for each subunit)."

"Input NMINORSUBAREAREL set(s) of location points and radius, x,y,r (0-1.0)."

"Small minor subarea subordinate unit: A subunit whose domain is a square whose half-width is specified in meters. Input NMINORSUBAREAABS = number of small minor subarea subordinate units (1-5)."

"Input NMINORSUBAREAABS set(s) of [echelon,number,type,idno] (one for each subunit)."

"Input NMINORSUBAREAABS set(s) of location point coordinates, x,y (0-1.0) and radius, r (meters)."

"Input NPOINT = number of non-area (point) subordinate items (1-30) (e.g., platforms and equipment)."

"Input NPOINT set(s) [echelon,number,type,idno] (one for each subunit)."

"Input NPOINT set(s) of location points x,y (0-1.0)."

If the unit being defined is a specific unit, it may possess an objective, a mission, and an approach to the objective. The objective is represented by an ellipse drawn on the screen. The mission type is an integer between one and five; what actual mission type (e.g., proceed to, monitor, jam, occupy, destroy) corresponds to these five numbers is up to the user. Once he has set up a correspondence between these five numbers and five actual mission types, he may refer to the mission numbers in the rules. The avenue of approach is a line drawn from the center of the unit's front to the center of the ellipse defining the objective.

¶Note that the objective location is specified in map coordinates, not in the unit's

standardized coordinates. An implication of this is that, although the unit may be moved, the objective and area of approach do not. Also, since the objective location and avenue of approach are not linked to any geographic or military object, they never change. In the current version of the Scenarist, the only way to change the objective and avenue of approach is to redefine the unit (e.g., by erasing it and forming a new unit by copying from a generic unit, or by copying it to a new specific unit (and specifying a changed objective and avenue of approach) and erasing it).

"Input 1 to input objective."

"Input mission type (integer, 1-5)."

"Input two corners of objective's bounding rectangle (4 numbers: x1,y1,x2,y2, map coordinates)."

"Input completed. Enter 1 to save defined unit, ANY OTHER KEY TO ABORT." At this point, the user would typically save the defined unit, and it would be recorded at the end of the generic-unit file or specific-unit file.

This completes the description of the function `_defineunit` that enables the user to define a unit and store it for future reference.

b. Copy Unit Function, `_copyunit`

The function `_copyunit` enables the user to define a new unit by "copying" an existing ("old") unit.

By the term "copy" is meant that the new unit possesses all of the same attribute values of the old unit, except for a few attribute values that may be modified by the user. (An "attribute" is any of the components, or variables, of the unit data structure.) The user may copy a unit from either a generic unit file or from a specific unit file, and form a new unit that is either a specific unit or a generic unit.

The `_copyunit` function is intended mainly to assist the definition of specific units. The user defines a generic unit of a particular side, echelon, and type, and then makes as many copies of it as there are specific units of this type. All that is added for each new copy is the new unit's code, parent unit's code, idno, parent unit's idno, map location, objective, mission, and avenue of approach. The new unit's side, echelon, and type are inferred from the code. If the new unit's parent unit already exist, its map location is derived from the location of the parent unit and its relative location in the parent unit.

In the current version of the Scenarist, the occasions in which the user would wish to copy from a generic unit to a generic unit, or from a specific unit to a specific unit are probably rare. These options would be of greater interest if the Scenarist had a general unit editing capability, rather than the limited editing capability it currently possesses (i.e., simply the ability to reposition subitems of geogtypes 6 and 7).

¶The process of copying units is facilitated by the use of data structures. The function `_getunitbycode` is used to find the old unit in its unit, and read it into a data structure of the unit data structure type in memory. All the program has to do is modify selected components of this data structure, and write the modified data structure to the end of a file.

The following paragraphs describe the questions (or menu options) posed by the Scenarist program in the process of copying a unit. Since the data required in the process of copying a unit are generally a subset of the data required to define a unit, much less descriptive material is presented in the discussion that follows than in the description of the Define Unit function presented above.

Menu choice:

- 1: Copy FROM Generic Unit TO Generic Unit
- 2: Copy FROM Generic Unit TO Specific Unit
- 3: Copy FROM Specific Unit TO Specific Unit
- 4: Copy FROM Specific Unit TO Generic Unit"

The old unit (to be copied from) will be read from the appropriate unit file (i.e., either the generic unit file specified in the project file or the specific unit file specified in the project file).

"Input new unit's name (up to 16 characters)."

"Input code for old unit (11 integers)."

If the user is copying to a specific unit file, the idno parentidno, and parentcode must be specified.

"Input code for parent unit (11 integers)."

"Input 'idno,parentidno' for new unit."

If the user is copying to a specific unit file and the parent unit is defined, the map coordinates of the new unit are derived from the parent unit's location and the relative position of the unit in the parent unit. (Whether a parent unit exists is determined by the function `_getunitbycode`.) If the user is copying to a specific unit file and the parent unit is not defined, the user must input the map coordinates of the new unit.

"Parent unit is not defined. Input four corner, counterclockwise (8 numbers, x1,y1,x2,y2,x3,y3,x4,y4, map coordinates."

If the user is copying to a specific unit file, then he may specify an objective, mission, and avenue of approach for the new unit.

"Input 1 to input objective, any other key otherwise."

¶"Input mission type (integer, 1-5)."

"Input two corners of objective's bounding rectangle (4 numbers: x1,y1,x2,y2, map coordinates)."

"Input number of points along avenue of approach to objective."

"Input coordinates for xxx avenue-of-approach points."

"Input completed. Enter 1 to save copied unit, ANY OTHER KEY TO ABORT."

If the copied unit is to be saved, it is then appended either to the generic unit file or the specific unit file specified in the project file.

c. Delete Unit Function, deleteunit

The deleteunit function enables a user to delete a unit from either a generic unit file or a specific unit file. This function operates by writing all of the units before and after the unit to be deleted to a "scratch" file, and then rewriting them (without the unit to be deleted) back into the unit file. Once again, this process is straightforward because the units are stored in block format in the unit files.

The deleteunit function asks the following questions.

"Input 1 to delete a unit from the generic unit file, any other key to delete a unit from the specific unit file."

"Input code for unit to be deleted (11 integers)."

d. Reposition Unit by User Function, repositionunitbyuser

The `_repositionunitbyuser` function enables the user to change the position of (i.e., "move") a unit on the battlefield. Since this operation is accomplished by specifying new coordinates for the four corners of the unit, the user may change not only the general location of a unit, but its size and shape as well.

¶The significant aspect of moving a unit is that the locations of all of the unit's defined subordinate units must be correspondingly changed (and all of the subordinate units' defined subunits, etc.). This relocation of all of the defined subordinate units is accomplished by the function `_repositionprogeny`. The `_repositionprogeny` function operates by placing all of the codes of the defined subunits of a moved unit in a "stack" (i.e., by scanning the specific unit, finding all units that have the moved unit as a parent, and placing all of their codes at the bottom of the stack). It then selects (from the specific unit file) the subunit corresponding to the code at the top of the stack, and modifies its location in accordance with the new location of the parent unit. Every time it processes a defined subunit in this way, it adds all of its subunits' codes to the bottom of the stack. Eventually, all units in the specific unit file whose positions were affected by the original unit move are repositioned.

The `_repositionprogeny` makes use of the C function `ftell`, which determines the file position of a record (i.e., the file position of a unit whose location is being modified), and the C function `fseek`, which can be used to set the

file-writing function to rewrite the modified unit record in the place of this record.

The `_repositionunitbyuser` function asks the following questions.

"Input code for unit to be moved (11 integers)."

The program prints the coordinates of the unit's current location on the screen.

"Input four corners of unit boundary, counterclockwise (8 numbers, `x1,y1,x2,y2,x3,y3,x4,y4`, map coordinates."

Recall that a unit's objective and avenue of approach are specified in absolute (map) coordinates. If the unit has an objective, the user may wish to move it as well. If the unit does not already have an objective, the user is given the opportunity of inputting one. If it already has one, he is given the opportunity of modifying it.

"This unit currently has no objective. Input 1 to define one, any other key otherwise."

If the unit already has an objective:

"This unit currently has a defined objective. Input 1 to delete the objective, any other key otherwise."

"Input 1 to retain current objective without change, any other key otherwise."

"Input objective (new objective or modified existing objective."

"Input mission type (integer, 1-5)."

"Input two corners of bounding rectangle around objective (4 numbers, x1,y1,x2,y2, map coordinates."

"Input no. of points along avenue of approach to objective."

¶"Input coordinates for xxx avenue-of-approach points."

"Processing completed. Enter 1 to save moved unit, ANY OTHER KEY TO ABORT." If the user enters "1" the program proceeds to execute the _repositionprogeny function.

e. Reposition Subunits by User Function, repositionsubunitsbyuser

The _repositionsubunitsbyuser function enables the user to reposition subitems of geogtypes 6 and 7 in a unit. Items of geogtype 6 are small minor subarea subunits of absolute radius (i.e, radius specified in meters); items of geogtype 7 are platforms and equipment. The function operates by reading the unit from a file, modifying the subitem positions (and entering a placement code), and writing the unit back in file over its original place in the file.

This function asks the user the following questions.

"Input 1 to reposition subunits in a generic unit, any other key to reposition subunits in a specific unit."

"Input code for unit to be reconfigured (11 integers)."

The program then requests the user to input new positions for the subitems of geogtypes 6 and 7.

"Input NMINORSUBAREA = xxx sets of location point coordinates, x,y (0-1.0), radius, r (meters) and placement code.

Placement code: 1 = user-reviewed, unchanged
2 = user-suggested placement
3 = user-mandated placement"

"Input NPOINT = xxx sets of location points, x,y (0-1.0) and placement code.

Placement code: 1 = user-reviewed, unchanged
2 = user-suggested placement
3 = user-mandated placement"

"Processing completed. Enter 1 to save reconfigured unit, ANY OTHER KEY TO ABORT."

If the user enters a "1," the reconfigured unit is replaced in the unit file. The function `_repositionprogeny` is then executed to reposition all defined subunits of the reconfigured unit in accordance with the changes.

f. Reposition Subunits by Rules Function, repositionsubunitsbyrules

(1). Function Overview

The function `_repositionsubunitsbyrules` applies the rules of the knowledge base to assess the suitability of the locations of subitems of geogtypes 6 and 7, and repositions them in accordance with the "action" functions. The function operates in two stages. First, it applies all "local" rules (to both types of subitems), and then it applies all "global" rules (to both types of subitems). The distinction between local rules and global rules was discussed earlier.

The function can be operated in a "CLIPS mode" or a "nonCLIPS mode." To operate the function in the nonCLIPS mode, the user must specify the rules in a C-language function, `_suitabilityxxxx` (where the `xxxx` is a project/subproject identifier). To operate the function in the CLIPS mode, the user has to define a CLIPS interface function, `_clipssuitabilityxxxx`, and place the rules in a CLIPS rule file, `clipxxxx.fil`.

If a subitem is deemed (by the rules) to be unsuitably located, an "action" function determines how to relocate the item. For local rules, the action is to initiate, continue, or terminate a "spiral" search for a suitable location. For global rules, the action is specified by an action function, denoted by `_actionxxxx`.

The numerical values of factors to be used in the rules are determined in a number of C-language functions, such as were discussed in the

preceding chapter. These values are derived from data about the relationship of subunit positions to geographic terrain features (e.g., terrain-type, elevation) the positions of other subunits, and the positions of other units. In many cases, these factors may be computed "on-line," as the rules are processed. In other cases, it may be advantageous to compute a matrix of factor values (one for each cell of a grid covering the unit) in advance of any rule processing for the unit. This matrix is then available to permit rapid access of factor values during the rule processing. Whatever factor preprocessing is desired is accomplished in the function `_preprocessingxxxx`.

The only map data available to the Reposition Subunits by Rules function are the cell values of the 32-cell by 32-cell maps stored in memory.

(2). Function Details

¶The essence of the Scenarist is that it is a rule-based system for positioning subunits. Since the `_repositionsubunitsbyrules` function provides the interface between the rules of the knowledge base (whether stored in C-language functions or in the rule files of CLIPS) and the rest of the Scenarist system, this function represents the heart of the Scenarist program. Because of the importance of this function, a higher level of detail will be used to describe its design than is used for the other Scenarist functions. The following paragraphs provide additional discussion about the Reposition Subunits by Rules function. This information

has been extracted from the Scenarist program listing that accompanies the Programmer's Manual (specifically, the program listing for the file s03kpsr.c, which contains the function _repositionsubunitsbyrules and several other functions that it calls).

In order to use the Scenarist to generate scenarios in a particular application, the user must develop a C-language "module," or file, containing functions that evaluate the factors he wishes to include in rules and a CLIPS interface function. The Scenarist listing contains two examples of these modules -- s03kpsr.c and s03dprs2.c. The file s03kpsr.c contains all of the functions that were used in the "Beqaa Valley" example used in the development of the Scenarist, as well as several functions (_repositionunitsbyrule and _spiralsearch) that are used by all applications. The file s03kpsr2.c contains all of the functions used in the TRAILBLAZER test. It is suggested that the user name his corresponding file s03kpsr3.c for the next application, s03kpsr4.c for the next, and so on.

To assist the user in developing his first application, the following description provides explanation of some of the important features of s03kpsr.c and s03kpsr2.c and the functions they contain. The description about s03kpsr.c and s03kpsr2.c reiterates some of the discussion that was presented earlier in this report in a general context or at a higher level of detail.

The function `_repositionsubunitsbyrules` examines the positions of the subitems of a unit, and determines, using the rules stored in a knowledge base, whether to reposition them. The present version processes nonarea subitems -- subitems of geographic type 6 (minor area subunits of absolute radius) and subitems of geographic type 7 (non-area (point) subordinate items (platform or equipment)).

The file `s03kpsr.c` contains two kinds of functions -- general functions that are used (or may be used) in all problems, and functions that are specific to problem 01 (a test case involving the placement of field artillery and air defense radars). The general functions are:

- `_repositionsubunitsbyrule`
- `_spiralsearch`
- `_terrain`

- `¶_elevation`
- `_distancetofeba`
- `_horizonangle`
- `_mapcellsizestdcoords`

The specific functions are:

- `_preprocessing0101`
- `_suitability0101`
- `_clipssuitability0101`
- `_action0101`

The file `s03kpsr2.c` contains functions that are specific to problem 02, a test case involving the placement of TRAILBLAZER units. The functions included in `s03kpsr2.c` are:

- `_preprocessing0201`
- `_createaccessibilitymap`

```
_accessibility
_losttotarget
_LOS
_road
_slopetorearcell
_losttootherunits
_lostoheadquarters
_disttootherunits
_disttofront
_inforwardarea
_action0201
_suitability0201
_clipssuitability0201
```

The functions `_preprocessing0201`, `_action0201`, `_suitability0201`, and `_clipssuitability0201` were all given the suffix 0201 because it is unlikely that they would be used in another application. The other functions might be reused, and for this reason they have not been given a problem-specific suffix. An explanation of what each function does is given at the beginning of the program listing for each function.

¶The function `_preprocessingxxxx` is a function that computes attributes for each cell of a grid defined over a unit. These attributes are available for use by the rules for placement of items. The rules are accessed in the function `_suitabilityxxxx` or the function `_clipssuitabilityxxxx`. `_suitabilityxxxx` is a C-language function that determines whether a location is suitable `_clipssuitabilityxxxx` is a function that calls on the CLIPS knowledge-based system to determine location suitability. The function `_clipssuitabilityxxxx` and the CLIPS

rule file clipsxxxx.fil must be matched. The function `_actionxxxx` specifies the processing that is to be done as the result of the firing of a rule related to a "global" constraint (defined below). The action that is done as a result of the firing of a rule related to a "local" constraint (defined below) is either to initiate or continue a spiral search for a location that satisfies all of the local constraints. The specific functions used in a problem are identified at the end of the projectfile, `projxxxx.fil`, along with a CLIPS rule file.

Additional description of the procedure for rule-based repositioning of subitems follows. The algorithm examines the suitability of the location of an individual subitem or the configuration (locations) of more than one subitem, and, if it is not suitable (according to the rules), rules are executed in an attempt to reposition the subitem or subitems in an attempt to find a more suitable location/configuration. The rules are applied to each new location/configuration considered. If a more suitable location/configuration is found, the subitem(s) is/are relocated to that location/configuration.

The search for more suitable locations for subitems is conducted over a grid over the unit's area. At most `maxcells` (now = 8) of the cells of this grid are examined in each search. For this process to be reasonable, it is desirable that the cellsize of the terrain, elevation, and road maps be comparable.

The functions `_suitability0101` and `_clipssuitability0101` (and the rule set `clips0101.fil`) are designed to handle only "local" constraints, i.e., constraints that can be checked from a knowledge of the local terrain. They are not designed to handle "global" constraints, such as constraints on line of sight to other subunits or constraints on the total number of subunits having line of sight to a headquarters subunits. In these functions, the search for a more suitable location is executed and completed for each subunit in sequence. In summary, these functions:

- Determine suitability of current location (according to rules).

- If current location unsuitable and less than `maxcells` cells have been examined, then continue the spiral search. If current location is suitable, terminate the spiral search and reposition the subitem to that location.

¶The functions `_suitability0201` and `_clipssuitability0201` (and the rule set `clips0201.fil`) are designed to handle both local constraints (e.g., terrain type) and global constraints (e.g., line-of-sight (LOS) among units). In the handling of global constraints, it is not possible (i.e., reasonable) to complete processing for one subunit before continuing to another; instead, the positions of the subunits must be adjusted simultaneously, or small adjustments must be made seriatim. (Reason: LOS constraints may be satisfied for one subunit at

one time, but those constraints may become unsatisfied when another subunit is moved at a later time.) The algorithm implemented by these functions conducts a two-phase search for an improved (more suitable) configuration.

First, for each subitem, it conducts a local search for a suitable location, taking into "local" constraints (i.e., taking into account factors that can be computed from facts about the subitem and terrain, but not from the positions of other subitems). It does this for all subitems seriatim (when the variable localglobal is set equal to 0). If the functions `_suitability0201` or `_clipsuitability0201` initiate a search for an improved location, that search is implemented without further calls to these functions (unlike the procedure with `_suitability0101` and `_clipssuitability0101`).

The algorithm then checks (when localglobal = 1) whether both local and "global" constraints are satisfied ("global" constraints are constraints that require knowledge of the positions of other subitems), makes a minor adjustment in the location of each subunit and then goes on to process the next subunit. This process is repeated a number of times, either until a suitable reconfiguration is found or a maximum number of iterations occurs.

In summary, the algorithm conducts (1) a search for a "locally feasible location" for each subitem (i.e., a location that satisfies all local constraints); and (2) a search for a globally feasible configuration for all subitems

in the unit (i.e., a configuration that satisfies all constraints, both local and global).

Note that in using a rule-based expert system to reposition subitems, there is no attempt to determine a configuration that is optimal with respect to a specified optimality criterion (objective function) -- only a configuration that is specified by the actions expressed by the rules. If a suitable solution cannot be found by the rules (e.g., a unit has been placed mostly over water or mountains), then the user must reposition the unit or one or more of the subunits, or add more rules to the rule base.

In a later version of the Scenarist, a global optimization algorithm might be developed, as an alternative to the rule-based algorithm. This is a nontrivial optimization problem, however, since the objective function is not "separable" (i.e., the suitability of the location of a subitem depends on the locations of the other subitems). (The term "feasibility" is used in the optimization sense, to refer to the satisfaction of a set of constraints -- i.e., a feasible point is one that satisfies the constraints, but may not be optimal.)

¶The flow of logic in this function is (1) do any preprocessing (using `_preprocessingxxxx`) to compute variables that are needed in the search for a configuration that satisfies global constraints; (2) execute the rules (using either `_suitabilityxxxx` or `_clipssuitabilityxxxx`); (3) execute repositioning in an attempt to satisfy

local constraints; (4) execute the action `_actionxxxx` in an attempt to satisfy global constraints.

In the current version of the Scenarist, `suitability` is an indicator variable (0=unsuitable location, 1=suitable location). In a later version, consideration could be given to making suitable a continuous variable (0-1) that indicates the degree of suitability.

The functions `_suitability0101` and `_clipssuitability0101` (and data file `clip0101.fil`) contain rules for repositioning field artillery and air defense radars. The functions `_suitability0201` and `_clipssuitability0201` (and data file `clip0201.fil`) contain rules for repositioning TRAILBLAZER sections. There does not have to be a separate set of `suitabilityfunctions/rulefiles/actionfunctions` for each unit or equipment type; there could be one set. Separate sets were used in the development of the Scenarist to facilitate testing.

The variable `processitemstatus` (formerly `CLIPS_rule`) is a variable used by the function `_clipssuitability0101` to indicate whether an item/location pair is (1) initial location of a subunit; (2) initial location of a platform/equipment; or (3) a spiral-search location of either a subunit or a platform/equipment. This information is needed to present a better description in the `CLIPS dribble.txt` file as to what

item/location-type is being processed, or what is the stage of processing (i.e., initiate or continue search). The function `_clipssuitabilityxxxx` determines suitability (i.e., the value of `CLIPS_suitable`) irrespective of the value of `processitemstatus`. The variable `processitemstatus` is needed only by the subroutine `_clipssuitability0101`, in the CLIPS version of the Scenarist.

The values of `processitemstatus` are as follows:

- 1 when determining the suitability of the initial location of a subitem of geogtype 6 (i.e., a subunit)
- 2 when determining the suitability of the initial location of a subitem of geogtype 7 (i.e., platform/equipment)
- 3 when determining the suitability of a subitem in the spiral search algorithm (`_spiralsearch`)

¶The variable `localglobal` is used to enable the search for locations that satisfy local constraints. When `localglobal=0`, the functions `_suitabilityxxxx` and `_clipssuitabilityxxxx` determine suitability using only local constraints. When `localglobal=1`, suitability is determined with respect to all constraints (global and local). When `localglobal=0`, a spiral search is used to find a location that satisfies the local constraints. When `localglobal=1`, an action function `_actionxxxx()` is executed to move the subitem to a new location (that satisfies the

local constraints and, hopefully, will eventually lead to a location that also satisfies the global constraints).

g. Reposition FEBA

The function `_repositionfeba` enables the user to change the specification of the FEBA, i.e., the number and location of the points that define it. When this function is called, it prints out a description of the current FEBA specification. The FEBA specification is stored in two variables, `nfebapts` (the number of FEBA points) and an array, `feba[5][2]`, that includes the coordinates of the FEBA point locations.

The function `Reposition FEBA` asks the following questions.

"Input new no. of FEBA points."

"Input map coordinates for xxx FEBA points."

The new specification is stored in the variables `nfebapts` and `feba[][]`, and written into the file `febaxxxx.fil` over the old values.

h. Display Unit Function, `displayunit`

The function `_displayunit` enables the user to see a listing of all of the files in the specific unit file, and to select one of them (at a time) to be displayed on the current map. The map is zoomed so that the unit just fits on the map.

The `_displayunit` function operates by calling on the functions `_setcoordsforzoommap`, `_drawmap`, and `_drawunit`. These functions are described later (together with other functions dealing with the drawing of maps and units).

The questions asked by the `_displayunit` function are the following.

"Input 1 to display file contents, any other key to continue."

"Select unit for display. Select by index (input 1), idno (input 2), or code (input 3). Hit any other key to skip display.."

Depending on the option selected, the program asks one of the three following questions.

"Input index."

"Input idno."

"Input code (11 integers)."

"Input 1 to DISPLAY another unit, any other key to continue."

i. Output Units Function, `outputunits`

The function `_outputunits` formats the unit specifications stored in binary form in the generic unit and specific unit files and outputs them to the printer or to a file of the user's choosing. The output file format is text format, so that the user can readily reformat it to any other desired format (to input the

scenario data -- unit identifications and locations -- into another model.

This function asks the following questions.

"Input 1 to output generic unit file, any other key to output specific unit file."

2. Description of Map-Processing Functions

a. Draw Map Function, `_drawmap`

The function `_drawmap` is used to accomplish the Draw Terrain Map, Draw Elevation Map, Draw Road Map, Add Vector Map with Labels, Add Vector Map without Labels, and Draw Vector Map without Labels of the map processing submenu. This single function accomplishes these different functions by changing the values of its calling parameters (arguments).

The function `_drawmap` has a number of parameters, or arguments, that specify what type of map processing it is to do. The four parameters are `cellmaptype`, `mapindex`, `vectormap`, and `printlabels`. The following list specifies how the values of these four parameters control the processing performed by `_drawmap`.

```
if cellmaptype = 1, draw discrete-variable map
    if mapindex = 0, draw terrain-type map
    if mapindex = 1, draw road map
if cellmaptype = 2, draw continuous-variable
map
    if mapindex = 0, draw elevation map
```

```

    if vectormap = 0, don't draw or superimpose
vector map
    if vectormap = 1, superimpose ("add") vector
map

¶if vectormap = 2, draw vector map (i.e., erase
                                wi
                                nd
                                ow

                                fi
                                rs
                                t)
    if printlabels = 0, don't print labels on
                                ve
                                ct
                                or
                                ma
                                p

                                ob
                                je
                                ct
                                s
    if printlabels = 1 print labels on vector map
objects

```

The particular maps drawn are the 32-cell by 32-cell terrain-type, elevation, and road cellular maps stored in memory, and the vector maps specified in the vector map files identified in the project file.

The operation of `_drawmap` is relatively straightforward. When drawing a cellular map, the program draws a rectangle for each cell of

the map, and "fills" it with a fill pattern and color specified in a program header file. When drawing a vector map, the program draws a series of straight line segments that define the boundary of an area object or the segments of a linear object. The rectangles, lines, and fill patterns (and other figures, such as points) are drawn by means of calls to Microsoft graphics functions.

The `_drawmap` function accomplishes zooms through the use of Microsoft "windows" functions, such as `_setwindow` (which places the map in a certain rectangular area on the screen, called a "window"), `_setviewport` (which defines a real (map) coordinate system for the window, and `_settextwindow` (which defines a text coordinate system for the window). The procedures for setting the windows parameters are described in the Microsoft C documentation.

After drawing the maps, `_drawmap` calls the `_label` and `_legend` functions to draw the map labels and legends

b. Place Unit on Map Function (Draw Unit Function), `drawunit`

The function `_drawunit` draws specific units on maps. The function operates as follows. A specific unit is defined in terms of the map locations of its four corners and the relative locations of subunits of various geographic types (geogtypes) within the four boundaries defined by the four corners. The subunits are represented as quadrilaterals, circles, or

points inside the unit boundaries. The locations and sizes of these objects on the map are determined by transforming each subitem's relative coordinates to map coordinates, by means of the function `_transfstdtoreal`. The transformed objects are then drawn using the Microsoft line or circle drawing functions.

¶The function `_transfstdtoreal` converts the standardized relative coordinates of each subitem of a unit to map coordinates by means of a linear transformation from the unit square located at the origin of the first quadrant to the quadrilateral located on the map at the coordinates specified by the user when defining the unit. A mathematical description of the transformation executed by the function `_transfstdtoreal` is presented at the end of this section.

For subunits of geogtypes 1-6, the `_drawmap` function draws a quadrilateral representing the subunit, and places a symbol and label on the map. The symbol and label are drawn by calling the function `_symbol`. For subunits of geogtype 7, the function draws just the symbol and the label. The symbols and labels used for each type of unit are those specified in the symbol file.

Through the use of relative coordinates to define the positions of subunits in units, a tremendous simplification was realized in the amount of data processing associated with the repositioning of units. With this approach, all that is necessary to modify each time a unit is moved are the locations of the four corners of defined

subunits. It is not necessary to recompute any subunit locations stored in the unit, because these are specified in relative coordinates and are unaffected by the move.

The only disadvantage associated with the use of relative subunit coordinates is that it is necessary to transform these relative coordinates to map coordinates each time a unit is drawn. That is a small price to pay for the simplification described above.

We shall now present a mathematical description of the transformation effected by the function `_transfstdtoreal`. This function transforms coordinates from the unit coordinate system to the map coordinate system.

Figures 10 and 11 present illustrations of a unit represented in unit (normal, standard, relative) coordinates and the same unit represented in map (real) coordinates. In unit coordinates, the four corners of the unit are $p1:(0,0)$, $p2:(1,0)$, $p3:(1,1)$, and $p4(0,1)$. In map coordinates, the four corners are $p1t:(x1p,y1p)$, $p2t(x2p,y2p)$, $p3t:(x3p,y3p)$ and $p4t:(x4p,y4p)$. (Note: The notation $x1p,y1p,\dots$ for the transformed points might have more descriptively been $x1t,y1t,\dots$ ("t" for transformed), but p was originally used, and that notation now occurs throughout the Scenarist code.)

¶We shall now derive the formulas used to convert the point $p:(x,y)$ in the unit coordinate system to the point $pt:(c,d)$ in the map coordinate system. Let px denote the proportion of the

distance along the x-axis that the point p lies, and p_y denote the proportion of the distance along the y-axis that the point p lies. Since the unit is defined on the unit square, it is clearly the case that $p_x=x$ and $p_y=y$. Denote also $q_x=1-p_x$ and $q_y=1-p_y$.

Let $p_5:(x,0)$ denote the projection of the point p on the x-axis, and $p_6:(x,1)$ the projection of the point p on the upper side of the unit square. Denote the transformed points corresponding to these points as $p_{5t}:(c_1,d_1)$ and $p_{6t}:(c_2,d_2)$.

Let the transformation $T(x,y)=(c,d)$ be defined such that:

1. The four points p_1, p_2, p_3, p_4 in unit coordinates transform (map) to the points $p_{1t}, p_{2t}, p_{3t}, p_{4t}$ in map coordinates.
2. The transformation is linear.

The point $p_5:(x,0)$ maps to the point $p_{5t}:(c_1,d_1)$. Since the transformation is linear, we have:

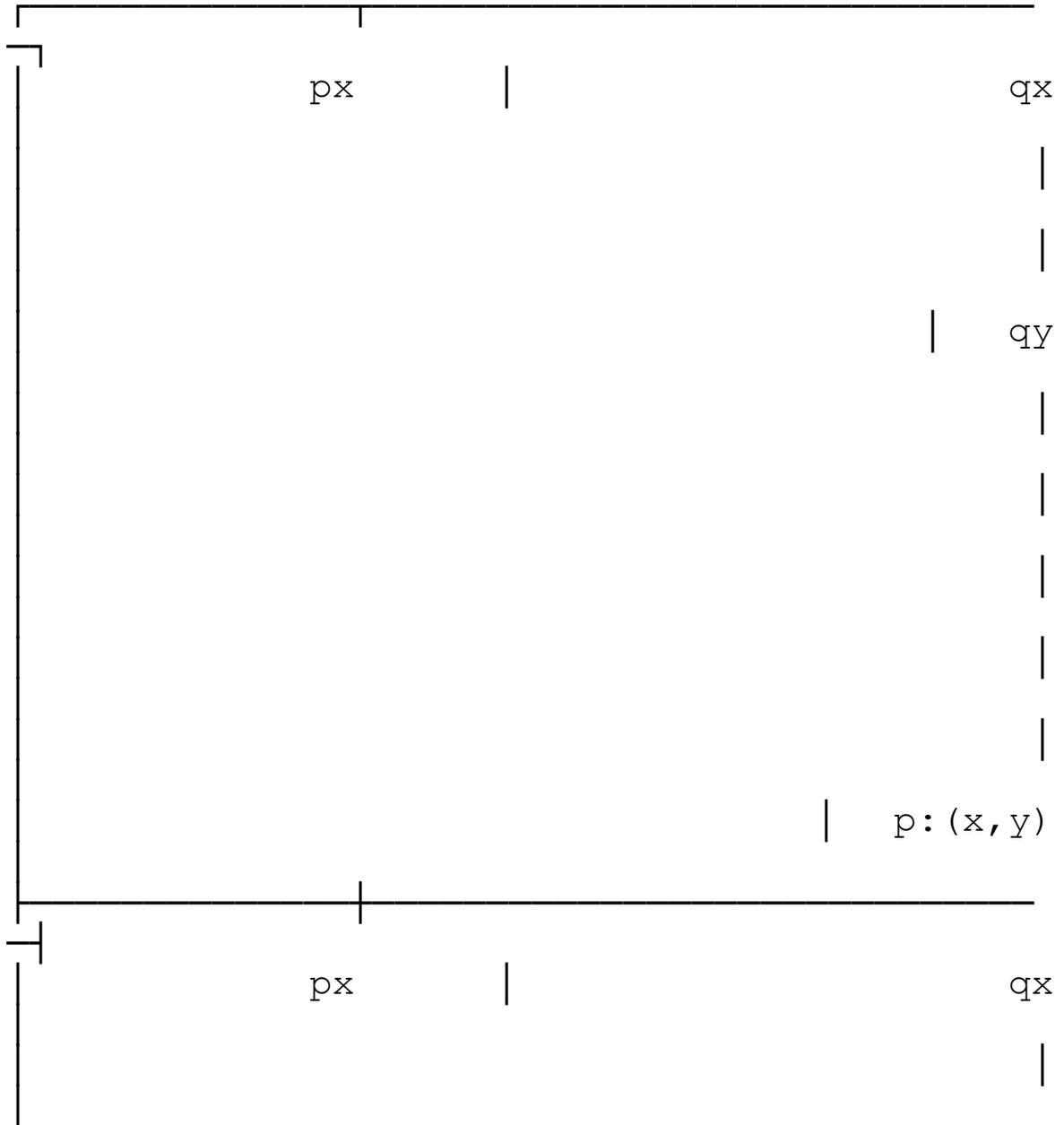
$$\begin{aligned}
 p_{5t}:(c_1,d_1) &= T(p_5:(x,0)) \\
 &= T(q_x p_1 + p_x p_2) \quad (\text{since the point } p_5 \\
 \text{is} & \\
 &\quad \text{located a proportion } p_x \text{ of the distance} \\
 &\quad \text{along the line from } p_1 \text{ to } p_2) \\
 &= T(q_x p_1) + T(p_x p_2) \quad (\text{since } T \text{ is a} \\
 \underline{\text{linear}} &
 \end{aligned}$$

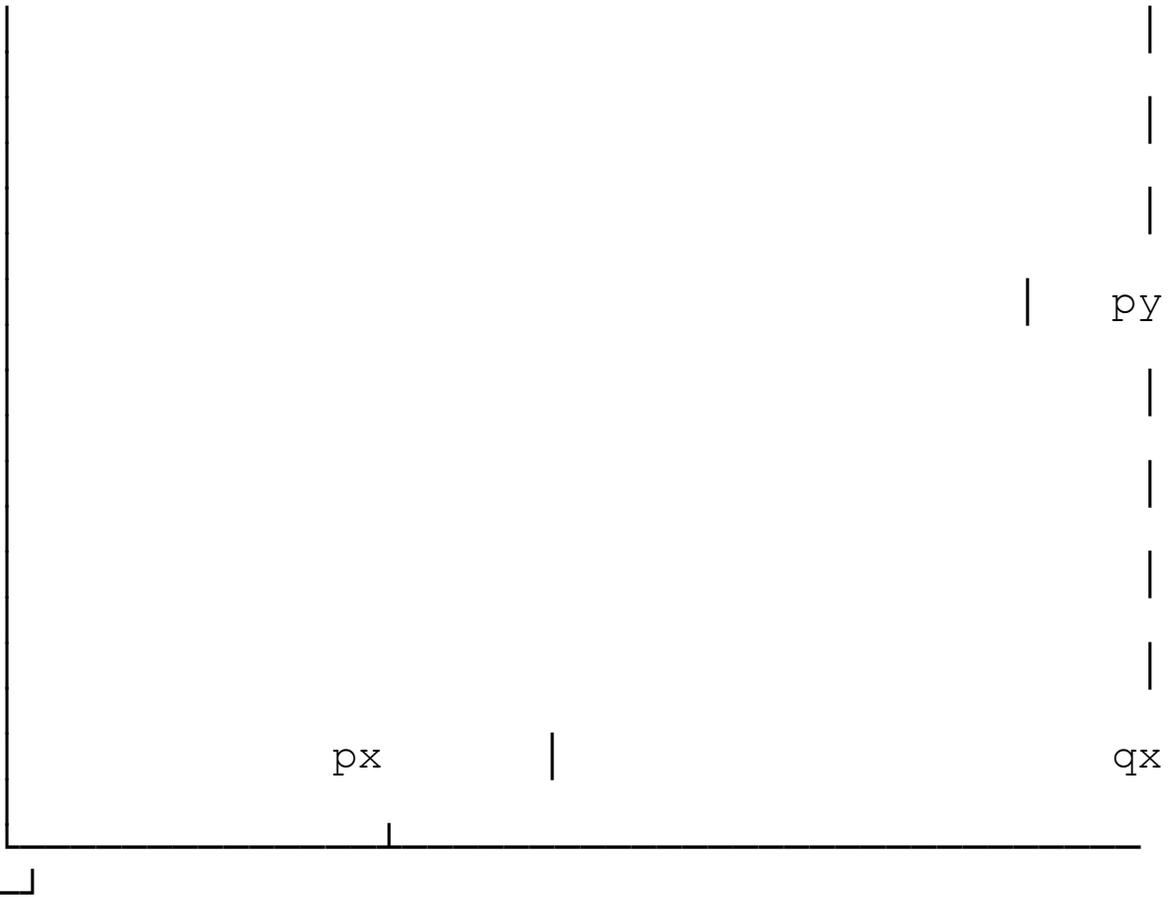
¶

Figure 10. Representation of a Military Unit in the Unit Coordinate System.

p4: (0,1)
p3: (1,1)

p6: (x,1)





p1: (0, 0)
 p2: (1, 0)

p5: (x, 0)

Unit Front

¶ Figure 11. Representation of a Military Unit in the Map Coordinate System.

p3t: (x3p, y3p)

p6t: (c2, d2)

p4t: (x4p, y4p)

qx

px

qy
 qx
 px
 $pt: (c, d)$
 py
 qx
 $p2t: (x2p, y2p)$
 px
 $p5t: (c1, d1)$
 $p1t: (x1p, y1p)$
 Unit Front
 \mathbb{T}
 transformation)
 $= qx T(p1) + px T(p2)$ (since T is a
linear
 transformation)
 $= qx p1t: (x1p, y1p) + px p2t: (x2p, y2p)$.

Hence

$$c1 = qx x1p + px x2p \quad (1)$$

and

$$d1 = qx y1p + px y2p \quad (2) .$$

Similarly, the point $p_6:(x,1)$ maps to the point $p_{6t}:(c_2,d_2)$. Since the transformation is linear, we have:

$$\begin{aligned}
 p_{6t}:(c_2,d_2) &= T(p_6:(x,1)) \\
 &= T(qx p_4 + px p_3) \quad (\text{since the point } p_6 \\
 \text{is} & \\
 & \quad \text{located a proportion } px \text{ of the distance} \\
 & \quad \text{along the line from } p_4 \text{ to } p_3) \\
 &= T(qx p_4) + T(px p_3) \quad (\text{since } T \text{ is a} \\
 \text{linear} & \\
 & \quad \text{transformation)} \\
 &= qx T(p_4) + px T(p_3) \quad (\text{since } T \text{ is a} \\
 \text{linear} & \\
 & \quad \text{transformation)} \\
 &= qx p_{4t}:(x_{4p},y_{4p}) + px p_{3t}:(x_{3p},y_{3p}) \quad .
 \end{aligned}$$

Hence

$$c_2 = qx x_{4p} + px x_{3p} \quad (3)$$

and

$$d_2 = qx y_{4p} + px y_{3p} \quad (4) \quad .$$

Finally, the point $p:(x,y)$ maps to the point $p_t:(c,d)$, so

$$p_t:(c,d) = T(p:(x,y))$$

$$\begin{aligned}
&= T(qy \ p5 + py \ p6) \\
&= qy \ T(p5) + py \ T(p6) \quad \text{since } T \text{ is linear} \\
&= qy \ p5t:(c1,d1) + py \ p6t:(c2,d2)
\end{aligned}$$

Also

$$c = qy \ c1 + py \ c2 \quad (5)$$

and

$$d = qy \ d1 + py \ d2 \quad (6) .$$

Along with the definitions $px=x$, $qx=1-x$, $py=y$, and $qy=1-y$, the formulas (1) - (6) are those used in the function `_transfstdtoreal` to transform unit coordinates to map coordinates.

c. Zoom Map Function, setcoordsforzoommap

The function `_setcoordsforzoommap` modifies the values of the global variables that control the map window specifications that in turn control the function `_drawmap`. It modifies these values in ways that accomplish the type of zooming specified by the user in response to the following questions.

There are three options for specifying a zoom. The user may "unzoom" the map to the original, full size it had prior to any zooming (or calls to `_displayunit`, which automatically zooms). Or, he may set the zoom map coordinates as the bounding rectangle of a unit (as `_displayunit` does). Or, he may specify the location point (top-left corner) and width of the map. Selection and specification of these options is implemented through the following questions.

"Input 1 to return to original map, any other key otherwise."

"Input 1 to set zoom map coordinates as bounding rectangle of a unit, any other key otherwise."

"Input unit code (11 integers)."

"Input location point (x,y) for zoomed map (location point is coordinates of top left corner)."

"Input width of zoomed map (in meters)."

d. Change Map Files Function

The Change Map Files menu option is implemented by means of the `_Change_Map_Location` function, an entry-point function that calls the functions `_readmapheader`, `_readmapdata`, and `_resetmaplocpoint`. These functions allow the user to input a 32-cell by 32-cell cellular map from a different part of the maps stored in the current cellular map files.

¶The approach to handling several types of maps of varying resolutions was described in the preceding chapter. This approach consists of reading the headers from the terrain-type and elevation map files, and suggesting a map location point to the user. The suggested map location point is the top left corner of the intersections of the terrain-type and elevation maps. The user may accept this suggested point or input another one. In either case, all three

cellular maps files (terrain-type, elevation, and road) are then accessed and a 32-cell by 32-cell map read from each one from the specified location.

The questions posed to the user by the function `_Change_Map_Location` are the following.

"Enter 1 to change output (display) map location point from map-intersection location point, any other key otherwise."

"Enter 1 to use upper-left-hand-corner as map location point, any other key otherwise."

"Enter coordinates of map upper-left-hand corner (x,y)."

"Enter coordinates of map center (x,y)."

e. Change Map Files Function, Change Map Files

The function `_Change_Map_Files` enables the user to specify new map files. It informs the user of the MS-DOS names of the current map files (terrain-type, elevation, and road), elicits certain responses from the user, and then calls the functions `_readmapheader`, `_readmapdata`, and `_resetmaplocpoint` to input 32-cell by 32-cell maps from the new files.

The questions posed to the user by `_Change_Map_Files` are the following.

"Input 1 to change terrain-type file, any other key otherwise."

"Input name of new terrain file (up to 12 characters)."

"Input 1 to change elevation file, any other key otherwise."

"Input name of new elevation file (up to 12 characters)."

"Input 1 to change road file, any other key otherwise."

"Input name of new road file (up to 12 characters)."

f. Print Map Function, _printscreen

¶The function _Print_Map is an entry-point function that calls the function _printscreen, which prints the contents of the screen (in graphics mode) on a Hewlett-Packard series II laser printer. This function was extracted from published sources.

The striking difference between the printed output and the monitor screen is, of course, that the screen is in color whereas the printed copy of the screen is in black and white. The maps and units have been drawn in the screen using a "colorblind" approach, to maximize the readability of the printed output. That is, although a variety of colors were used on the screen, color was not used as a code. Also, different densities of dots were used to represent different elevations. Higher

elevations correspond to higher dot densities (in cells), so that high areas (hills and mountains) show up darker than neighboring low areas (valleys).

References 13 and 14 (Scenarist Summary and Sample Demonstration Demo 1) present a demonstration of the Scenarist that includes a number of hard-copy printout of several screens generated by the Scenarist program. These items are included in Appendix B to this report.

¶VIII. Scenarist Software Documentation

The Scenarist software, consisting of the Scenarist program and data files, is documented primarily in three volumes, entitled the Programmer's Manual (a two-volume set, one volume of which is the program listing), the Variable Glossary, and the User's Manual. The purpose of the Programmer's Manual is to provide assistance to a programmer in making modifications to the system. The purpose of the User's Manual is to provide assistance to a new system user.

The Programmer's Manual includes a description of all of the Scenarist functions and their interrelationships. It presents a number of data flow diagrams that show the flow of data into and out of the major system modules. It also includes a line-by-line listing of the program. The code is heavily commented, and has been divided into many small modules (C-language functions) to enhance understandability and

assist program maintenance. Related functions have been grouped into files.

The Variable Glossary contains definitions of all functions and variables used in the Scenarist programs. (Ordinarily, such definitions would have been included within the programs, but it was a contract requirement to produce a separate variable glossary.)

The User's Manual provides information on installing the Scenarist system. It identifies the hardware requirements and installation steps. The system installation has been simplified thorough the use of an installation program, INSTALL. The User's Manual identifies and describes all of the major menu choices available in the Scenarist system, and the steps involved in using the system.

A major aspect of using the Scenarist is the preparation of data (e.g., map files, unit files, rule files) for use by the system. The procedures for accomplishing the data preparation will vary from application to application. The Test Report included as an appendix to this report provides a good step-by-step description of the procedures for setting the Scenarist up and running it.

¶During the course of the software system design, the software system development was documented in accordance with the DOD-STD-2167A software development standard. The System/Segment Design Document, the Software Requirements Specifications, and the Software Design Document

documents of the references (9 documents in all) were prepared in accordance with this standard. As discussed earlier, the burden of updating these documents in accordance with the 2167A standard became too great, and so (in agreement with the CECOM project officer) they were not updated. Because of the fairly substantial design changes that occurred during the rapid prototyping development phase of the project, these documents are now out of date, and of little value in assisting someone desiring to modify the system. The Programmer's Manual, Variable Glossary, User's Manual, and this final report contain the up-to-date descriptions of the top-level design and detailed design.

¶IX. Validation and Verification of the Scenarist System

The Scenarist system is a mathematical model that is implemented in the form of a computer program. As a model, it needs to be validated. As a computer program, it needs to be verified. The validity of a model is the degree to which it is an adequate representation of reality for the purposes to which it is to be applied. With respect to the Scenarist, the validity issue to address is whether the model positions equipment on a battlefield in a fashion that is in some sense similar to what would occur in real warfare. Model validation refers to procedures for assessing the validity of a model. The verity of a computer program is its correctness and completeness. Verification refers to tests

that demonstrate that it correctly performs all of the functions that it is supposed to.

During the course of the Scenarist development program, a considerable amount of effort was expended in testing (verifying) the Scenarist program. In addition to the many undocumented tests of individual modules ("unit tests"), a formal, detailed test of the complete Scenarist software system was conducted. As discussed earlier, this test involved an application of the Scenarist to place TRAILBLAZER equipment on an area near Spearfish, SD. The test involved preparation of a Test Plan that described the goals, objectives, and procedures for the test, and a Test Report that described the activities and results of the test.

A copy of the Test Report is included in Appendix A. This copy is included in this report not so much as to provide a detailed description of the test, but to provide a detailed example of how the Scenarist is used and to illustrate the high level of detail that can be accommodated by the system.

In addition to the formal test, a system demonstration was prepared and sent to a number of individuals who had expressed interest in the Scenarist development. The demo enables the user to step through examples of major system functions. The hardcopy output of the demo is presented in Appendix B.

¶While the Scenarist has been subjected to a considerable amount of testing, with respect to

validation of the Scenarist, the project's accomplishments were quite limited. Validation is a key issue to address relative to future use of the Scenarist. In order for the system to be accepted and used, its validity must be established. There is a substantial literature on the topic of model validation; validation of a model such as the Scenarist can be accomplished in numerous ways. One way is for experts to examine the rules in the knowledge base, and agree to their reasonableness. Another way is for experts to examine Scenarist-generated scenarios and assess their reasonableness. To a large degree, the Scenarist is "self-validating," since it indicates on the screen when an item is not suitably located. The Scenarist possesses a high degree of "face validity." That is, there is a high degree of correspondence between the entities and processes of the model and those of the real world. The heavy object-orientation of the model (i.e., the representation of units as objects, implemented in the form of data structures in the Scenarist program) enhances its face validity. The fact that the composition of units and their organizational structure can be represented at a high level of detail, in close correspondence to actual unit composition and organization, further enhances its face validity.

Perhaps the greatest argument in support of Scenarist validity is the fact that it is a rule-based system. Tactical rules are used in real warfare to guide the positioning of military items, and the Scenarist system is a means for

quantifying those rules and implementing them in an automated fashion.

Earlier in the project, it was hoped that significant progress could be made toward validation of the Scenarist, including reviews by experts as mentioned in the preceding paragraphs. In particular, it was hoped to conduct an in-depth review of the TRAILBLAZER application. The individual at the US Army Intelligence Center and School who had been so helpful in providing us with information on the system had offered to review the TRAILBLAZER test results. Unfortunately, time and resources were so short near the end of the project that this detailed review was not possible.

During the course of the project, a number of Scenarist demonstrations were sent to interested individuals in a number of US military organizations. It is planned to distribute this report to them, for review. If future work is done on the system, their comments could be very useful in guiding the future development.

In summary, the Scenarist has been tested but not validated. In certain ways, its validity has been established. By its very nature (a high-detail, object-oriented, rule-based system), it possesses a high degree of face validity. Much more review needs to be done, however, in the area of running test applications and having them reviewed by experts, before the Scenarist's validity as a system for generating scenarios for use in test and evaluation is well established.

¶X. Summary of Scenarist Capabilities and Limitations

This chapter presents a summary description of the Scenarist system capabilities and limitations.

The Scenarist project was successful in accomplishing its objectives, i.e., the development of an automated scenario generation system based on artificial intelligence technology and that could take into account tactical doctrine, terrain features, friendly mission, and enemy threat. A larger issue to address, however, is whether the Scenarist is, in its present form or with some modification, a practical scenario generation tool. This chapter will discuss what the Scenarist can and cannot do.

The major features of the Scenarist are the following:

1. Ability to accomplish rule-based placement of military items.
2. Ability to use digital terrain mapping data.
3. Ability to operate on a 386-based microcomputer of a standard configuration (1 megabyte memory, modest-sized hard disk, VGA monitor, high-density 3-1/2" diskette drive, mouse, MS-DOS 4.2 operating system).
4. A self-validating ability, viz., the ability to indicate when an item's location violates a rule of the knowledge base.

5. Ability to incorporate vector-map backgrounds.
6. Ability to characterize friendly mission and enemy threat.
7. Ability to handle maps of varying resolutions.
8. Ability to operate over a wide range of map scales, from large-scale maps covering division- or corps-sized areas down to very-small-scale high-resolution maps showing individual items of equipment.
9. Ability to automatically reposition subordinate units when a parent unit is moved.
10. Ability to allow the user to make manual adjustments to unit positions.
11. An automated capability to define units for input to the system. An ability to represent unit composition at a high level of detail. An ability to represent the military organizational structure, i.e., the hierarchical relationships among the units.
12. An ability to include a very high level of map detail.
13. A parametric representation of units that is well-suited to the generation of factors to be used in rules and to the rapid relocation of units on the battlefield.
14. An object-oriented representation of maps and units that lends itself well to expert-system technology and understandable model.
- ¶15. A system developed using modern software engineering methodology (top-down, structured design), and including features to enhance maintainability, such as small modules.
16. A system programmed in C, to enhance portability.

17. A low-cost system, that contains no expensive components (compilers, expert systems, geographic information systems).
18. A system that contains the CLIPS expert system in a fully integrated embedded mode.
19. A system that has a user-friendly mouse/menu/windows graphics interface.
20. An ability to produce high-quality hard-copy scenario map output.
21. An ability to output a generated scenario either to the printer or to a text file.
22. Heavy use of data structures, to accomplish efficient handling of map and unit data and a high level of model understandability.
23. A demonstrated ability to generate laydowns of TRAILBLAZER units on digital-terrain data.
24. The Scenarist is a tool that enables a military analyst to see, on a color graphics screen, the placement of military units on a map. He can move units and subunits around on the map. He can apply rules to reposition subunits, and see the effect of the rules. In short, the Scenarist provides the user with an ability to construct and view a scenario at a wide range of detail.

In spite of the preceding capabilities and desirable features, the Scenarist has a number of shortcomings that limit its ability to be used for its intended purpose. These include the following.

1. In order to use the system, a considerable amount of time is required to review tactical doctrine, define units, and specify rules. Additional effort is required to prepare maps.

If the system is used to generate a single scenario, it is doubtful that the cost of preparing that scenario would be much less than the cost of manual preparation. Once the system is set up, however, many scenarios may be generated quickly.

2. Although the system possesses a high degree of face validity, it has not undergone serious scrutiny by experts. The system needs more validation. Experience is needed with applications that are critically examined by experts for reasonableness. Until the system undergoes substantially more validation, it cannot be used to generate scenarios in a routine "production" manner.

3. In its current form, the Scenarist is in essence still a research tool, not an operational one. The system design is sound, and the system possesses many powerful features. Because of the high level of detail that can be represented in the model, however, it is not possible for a user to simply sit down at the terminal and generate scenarios right away. A considerable amount of data collection and thought is required in preparing map and unit data for the system. To enhance the acceptance of the System, training materials are needed. The system is complex, and it is expected that most users would not be prepared to invest sufficient time in reading the system documentation (this final report, the Test Report, and the User's Manual) to properly use the system. In terms of complexity, the system is similar in nature to a war-game model.

4. Based on our experience with CLIPS, that system is not easy to use. What is needed to make

the system easy to use is some sort of rule editor that would enable the user to formulate rules without learning how to master the CLIPS rule syntax. The difficulty in using CLIPS would be perceived as a serious obstacle to using the Scenarist, for a potential user who had never used CLIPS.

5. Although the Scenarist design is quite general in nature, the current Scenarist program is for use on an MS-DOS microcomputer. This system can handle smaller problems, but would have difficulties with large applications (storage space, processing time). During the course of the Scenarist development, a potential user evidently lost interest in the system because it was "PC-based," and was not available on a powerful graphics workstation (such as a SPARCstation). Acceptance of the system would be significantly enhanced if a version were available for a more powerful graphics workstation.

¶XI. Areas for Future Scenarist System Development

During the course of the Scenarist system design, a large number of design options were considered. Further, during the course of the system implementation, many additional system features were identified that would enhance the user-friendliness of the system or expand the scope of the model. In its review of a demonstration version of the Scenarist, CECOM personnel identified additional improvements that would enhance the value of the system to

potential users. During the "rapid prototyping" development of the Scenarist system, a number of design improvements were identified and implemented. In order for the project to stay on schedule and accomplish its objectives, however, it was not possible to implement many additional desirable features. If additional development work is done on the Scenarist, many of these system improvements could be considered further, and some of them implemented. This chapter identifies various extensions that were identified during the course of the project, but not implemented because of time and resource constraints.

The potential extensions are divided into two categories -- extensions that make the system easier to use without changing its basic scope, and extensions that expand the system scope.

Extensions that Do Not Expand the System Scope

1. Port the Scenarist to a More Powerful Graphics Workstation. As discussed, this extension is desirable for two reasons: (1) to enable the Scenarist to handle large problems; (2) to eliminate the negative image that the Scenarist is a "PC-based" tool.

2. Create Interface to Use DMA Digital Mapping Data. Obtain DMA DTED and DFAD data on CD-ROM disk, and develop software to permit the Scenarist to access it directly.

3. Create Interface to Use DMA Diskette-Based Background Maps. During the course of the

project, we observed a new DMA map product -- map images stored on 3-1/2" high-density diskettes. Software could be developed to use these map images as the background maps, replacing the Scenarist's vector background maps. Alternatively, investigate the availability of software to permit the rapid development of vector background maps, or develop software to enable the user to easily develop Scenarist vector map files interactively, instead of by using a line editor.

4. Create Interface to BEEO Unit Data. Develop a system that would enable the extraction of unit data from the TO&E files maintained by BEEO, for use in the Scenarist.

¶5. Develop a CLIPS Rule Editor. Develop an interactive program that would enable the user to build a CLIPS rule file without having to master the CLIPS rule syntax. This program would be inserted in the Rules entry point of the main menu.

6. Develop a Scenarist Training Course. Develop training materials, such as PC-based tutors or video presentations, to instruct new users on how to use the system.

7. Develop a Rule Dictionary. In the current version of the Scenarist, the user develops a new CLIPS rule file for every application. Design a system for storing all rules in a data base, allowing the user to "pick and choose" the rules for a new application.

8. Develop an Expert System for Determining Geogtypes. Perhaps the most difficult aspect of defining a unit is deciding what geographic type should be used for each subordinate unit. Develop an expert system that would guide the user in doing this.

9. Change Method of Handling Cellular-Map Cell Codes. Currently, the definitions and labels of the codes of the terrain-type, elevation, and road maps are stored in a Scenarist header file. Modify the program so that the definitions and labels are derived from the map file header.

10. Reorganize Symbol File. The ordering of the labels and symbol numbers in the _symbol function is "helter-skelter." Symbol numbers were defined in the order in which symbols were coded. The ordering should be changed to correspond to side and echelon, and the symbol file rearranged to match this order.

11. Enable Mouse Control of Item Repositioning. In the current version, the user effects all repositioning by inputting coordinates over a keyboard. Develop a mouse interface so that the user may "click" on an object and "drag" it to reposition it.

12. Enable Mouse Control of Unit Definition. The process of defining a unit involves preparing manual sketches of units and entering data over the keyboard in response to English questions. Develop a graphics/mouse interface so that the user can see the various geogtypes displayed on

the screen, and see the generic unit displayed as he defines it.

13. Develop Automatic System for Changing Map Resolutions. The current method for generating maps of different resolutions is to generate them "off-line," using the s03xcomp program. Develop a system for changing map resolutions on-line.

¶14. Enable User Control of Stepsize in Rule/Action Processing. During the process of implementing the actions resulting from rules (i.e., in looking for a suitable location), the system scans over a 16 x 16 grid imposed on the unit. Provide an option for the user to change the grid size (e.g., use a 10 x 10 grid or a 20 x 20 grid).

15. Develop Software for "Batch Processing" of Units. Currently, the function Scenario Generation is simply an entry point. Develop software to process (i.e., apply rules to) a large number of units, with minimal user intervention.

16. Develop Rules to Guide Unit Positioning. The rules of the Scenarist reposition subunits of defined units. The user is unconstrained in his positioning of units that are not subordinate units of other units. Develop rules that guide the user in unit positioning. For example, there are presently no constraints on how large or how small a division may be. The system should warn the user if he specifies the unit

corners in such a way that the unit is absurdly small or large.

17. Consider CLIPS Replacement. The decision to embed CLIPS in the Scenarist was made two years ago. Reexamine the alternatives that are now available for expert systems, and determine whether a better choice could be made for the expert system component of the Scenarist.

18. Develop Guidance for Using CLIPS vs. C-Language Rules. The test applications that were developed during the Scenarist development project were small, in the sense that only a small number of rules was involved. There was no real need for the inference engine of an expert system; the rules could be readily coded in C, and there was no real problem in deciding the order in which they should be processed. Develop guidelines for determining for an application whether the rules should be implemented in CLIPS or in C-language functions.

19. Develop Output Formatter. Develop interactive software to enable the user to reformat the output file automatically.

20. Develop Lists of Exemplar Missions and Objectives. Currently, the definition of missions and objectives is left entirely to the user. Develop a list of sample missions and objectives.

21. Develop Software to Draw Line-of-Sight Maps. Develop software to show the user all of the line of sight rays emanating from a specified point,

or the length of sight ray in a particular direction.

¶22. Develop Software to Determine Suitability of a Selected Point. The system determines the suitability of the location of a particular item, according to the rules. Develop a capability by which the system would indicate the suitability of an arbitrary user-specified location (preferably selected using a mouse). That is, tell which rules are violated for the specified position. This information (together with the line-of-sight fan mentioned above) would assist the user in making his own placements.

23. Enable User Specification of Number of Iterations Used in Processing of Global Rules. Currently, at most three iterations are allowed in the processing of global rules. Allow the user to determine how many iterations are done.

24. Develop a General Unit Editor. Currently, the user is very limited in the extent to which he may edit an already defined unit (i.e., he may reposition only subunits of geogtypes 6 and 7). Modify the program to enable respecification of any aspects of a unit.

25. Develop Interactive Capability to Draw Symbols. Symbols are currently specified in C-language code. Identify or develop software to enable a user to draw a symbol interactively and store it.

26. Develop a Capability to Display Generic Units. The program can display specific units

on a map, but it cannot display the internal structure of a generic unit. Develop software to enable this to be done.

27. Develop Software to Extract Rectangular Maps from GRASS Sample Map Files. If additional work is done using the GRASS sample map files, modify the WIN7.FOR program so that it can extract rectangular cellular map files, not just square ones.

28. Develop Software to Store a Number of Vector Objects in Memory. If software is not developed to use DMA map images for map backgrounds, develop software to store a number of vector map objects in memory, to reduce the amount of file access (each time a vector map is redrawn).

29. Develop a Capability for Different Rules to Apply to Different Levels of Resolution. Currently, the same rules are applied, no matter what the level of map resolution. Develop software that would enable rules to be specific to map resolution.

¶30. Develop Guidelines for Data Structure Type for Platforms and Units. The test applications of the Scenarist involved only units, not platforms or equipment. When the first application is conducted involving individual items of equipment, determine a data structure type for platforms and equipment (the unit data structure type is much larger than is needed for platforms or equipment).

31. Enlarge Map. A CECOM review of a Scenarist recommended that the map window be made as large as possible. Pull-down menus could be used instead of the current fixed menu boxes.

32. Rescale Map Axes. The horizontal and vertical axes of the map are not of exactly the same scale. Modify the window size (for both VGA and EGA monitors) so that the scales are exactly the same.

33. Incorporate More Edit-Checking of User Input. While the program tests the validity of certain values input by the user (particularly, any variables that will be used as indices in arrays), it currently performs a minimal amount of edit-checking. For example, if the user inputs "nonsense" values in the process of defining a unit, the result will be a nonsensical unit. Develop edit checks for all variables input to the Scenarist.

34. Adopt Standard Map Symbology. The CECOM review strongly recommended that the system adopt standard topographic symbols and map references. Other recommendations included: the use of clearly marked grid north and magnetic north arrows; specification of the annual magnetic north drift angle; default of the map scale to standard topographic ratios (1:5000, 1:10000, 1:25000, and 1:50000); default of the map grid squares to 1, 2.5, 5, 10, or 50 kilometers; display of grid squares as a background utility, not highlighted, or interfering with military item outlines; use of contour lines (to represent hilly or mountainous

terrain); reference to surrounding maps; specification of direction and range of radar equipment; specification of radio or fiber-optic cable links as an available overlay; symbology for communication links.

Extensions that Expand System Scope

1. Develop a System for Scenario Sampling. Develop a methodological framework for generating a population of scenarios and for selecting scenario samples from that population, and implement such a system.

¶2. Develop a Probabilistic Scheme for Repositioning Subunits. The current Scenarist design is deterministic, and develops a single "best" repositioning of the subunits of a unit. Develop a methodological framework for making probabilistic repositionings, and implement such a system. Two options to be considered are allowing for alternative canonical units (which may be sampled probabilistically) or probabilistic selection of a suitable location from several alternatives near an item, in a local search.

3. Develop a Scheme that Allows for Suitability Values Other Than 0 and 1. Currently, a location is either suitable or unsuitable. Investigate the desirability of adopting a suitability concept that allows for a range of suitability values (e.g., from 0 to 1), and a scheme for selecting a suitable location within this framework.

4. Consider Optimization. Consider alternatives for optimizing the selection of item locations, according to some optimality criterion.

5. Implement the System in C++. Develop a version of the system in object-oriented C (e.g., C++).

6. Implement the System in X-Windows. If the system is ported to a "high-end" graphics workstation, consider implementing a new version in the X-Windows environment.

7. Implement a Raster or Vector-Based Version. Pixel-based computer monitors are slow for displaying graphic information, compared to raster or vector-based graphics systems. Consider conversion to a non-pixel-based system.

8. Develop Rules to "Flesh Out" the Parts of a Unit that Are of No Immediate Concern to the Scenario Application. In any particular application, only certain types of subordinate units will be of interest, and the rules need address only those types of units. Examine the feasibility of developing rules that would enable the complete specification of a unit, using all of the data included in the BEE0 TO&Es.

¶

References

1. Caldwell, J. George, Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems, Final Report, Contract No. DAAB07-87-C-P057 for the US Army Communications-Electronics Command, Vista Research Corporation, Sierra Vista, Arizona, March 28, 1988.

2. Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems, Phase II Small Business Innovation Research (SBIR) Proposal, Vista Research Corporation, Sierra Vista, Arizona, May 9, 1988.

3. Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems, Phase II Small Business Innovation Research (SBIR) Proposal, Cost Proposal Addendum, May 26, 1989, Vista Research Corporation, Sierra Vista, Arizona.

4. Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems, Monthly Progress Reports, beginning with September, 1989, and Quarterly Progress Reports, beginning with November, 1989, Vista Research Corporation, Tucson, Arizona.

5. Software Development Plan for the Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems, Vista Research Corporation, Tucson, Arizona, November, 1989.

6. System Design (System/Segment Design Document) for the Scenarist Automated Scenario

Generation System, Vista Research Corporation, Tucson, Arizona, May 31, 1990.

7. Software Requirements Specifications of the Scenarist Automated Scenario Generation System, Vista Research Corporation, Tucson, Arizona, May 31, 1990.

8. Software Design Document of the Scenarist Automated Scenario Generation System, Vista Research Corporation, Tucson, Arizona, May 31, 1990.

9. Scenarist Automated Scenario Generation System: Summary and Sample Output, Vista Research Corporation, Tucson, Arizona, February 20, 1991.

10. Scenarist Automated Scenario Generation System: Briefing Slides, Vista Research Corporation, Tucson, Arizona, February 20, 1991.

¶11. Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems: Scenarist Program Listing, Vista Research Corporation, Tucson, Arizona, Produced monthly after August, 1990. Final listing included (separately bound) with Programmer's Manual.

12. Test Plan for the Scenarist Automated Scenario Generation System, Vista Research Corporation, Tucson, Arizona, February 28, 1991.

13. Scenarist Automated Scenario Generation System: Sample Demonstration Demo 1, Vista

Research Corporation, Tucson, Arizona, April 1, 1991.

14. Scenarist Automated Scenario Generation System: Summary, Vista Research Corporation, Tucson, Arizona, April 5, 1991.

15. Test Report for the Scenarist Automated Scenario Generation System, Vista Research Corporation, Tucson, Arizona, April 30, 1991.

16. User's Manual for the Scenarist Automated Scenario Generation System, Vista Research Corporation, Sierra Vista, Arizona, August 31, 1991.

17. Programmer's Manual for the Scenarist Automated Scenario Generation System (including separately bound program and file listing), Vista Research Corporation, Sierra Vista, Arizona, August 31, 1991.

18. Variable Glossary for the Scenarist Automated Scenario Generation System, Vista Research Corporation, Sierra Vista, Arizona, August 31, 1991.

19. Scenarist Automated Scenario Generation System: Final Report, Final Report for Project, "Research in Artificial Intelligence for Noncommunications Electronic Warfare Systems (Contract No. DAAB07-89-C-P017)," Vista Research Corporation, Sierra Vista, Arizona, August 31, 1991.

Glossary

ADEM	Air Defense Effectiveness Model
ADP	Automatic Data Processing
C	A high level programming language
CD-ROM memory (compact laser disk for storage of digital data)	Compact disk -- read only
CECOM	US Army Communications-Electronics Command
CEW/RSTA	Center for EW/RSTA
CLIPS	A knowledge base system supported by NASA
COMINT	Communications Intelligence
CSCI	Computer Software Configuration Item
CSC	Computer Software Component
CSG	Command Systems Group
dBASE	A commercial data base management system
DGTS	Dynamic Ground Target Simulator
DMA	Defense Mapping Agency
DMWS	Digital Mapping Workstation
ELINT	Electronic Intelligence
ESM	Electronic Support Measures
EW/RSTA Reconnaissance, Surveillance and Target Acquisition	Electronic Warfare /
GRASS	Geographic Resources Analysis Support System: A Geographic Information System developed by the US Army Corps of Engineers

TRAC TRADOC Analysis Command
TRADOC US Army Training and Doctrine
Command
TAADS The Army Authorization Documents
 System

USAEPG US Army Electronic Proving
Ground
USAICS US Army Intelligence Center
and School

VTAADS Vertical TAADS

WORM Write-once, read-many

¶ **Distribution List**

1. AMSEL-RD-EW-C (Dr. Frank Elmer) (3 copies)
Fort Monmouth, NJ 07703-5000
2. AMSEL-RD-EW-CE (1 copy)
Fort Monmouth, NJ 07703-5000
3. AMSEL-RD-EW-SE (1 copy)
Fort Monmouth, NJ 07703-5000
4. AMSEL-RD-EW-C (Mr. Al Slutsky)
Fort Monmouth, NJ 07703-5000
5. Mr. Earl Anthony
Command Systems Group, Inc.
23430 Hawthorne Boulevard, Suite 150
Torrance, CA 90505
Tel: (213) 373-9619

6. Mr. James L. Cole, Chief
Electro Environmental Effects (E3) Division
Commander, US Army Electronic
Proving Ground
ATTN: STEEP-CP-E (Mr. Jim Cole)
Fort Huachuca, AZ 85613-7110
Tel: (602) 538-4860

7. Mr. Dan Searls, Chief
Simulation and Modeling Branch
Electro Environmental Effects (E3) Division
Commander, US Army Electronic
Proving Ground
ATTN: STEEP-CP-E (Mr. Jim Cole)
Fort Huachuca, AZ 85613-7110
Tel: (602) 538-4953

8. Mr. Robert Reiner, Chief
Modernization and Test Technology Office
ATTN: STEEP-MO
US Army Electronic Proving Ground
Fort Huachuca, AZ 85613
Tel: (602) 538-7618

9. Mr. Robert J. Harder, Head
Artificial Intelligence Office
Software and Automation Branch
ATTN: STEEP-ET-S
US Army Electronic Proving Ground
Fort Huachuca, AZ 85613-7110
Tel: (602) 533-8187

¶10. Col. Jay Vaughn, TRADOC Project Officer
Commander, US Army Intelligence Center and
School
ATTN: ATSI-TPO-AT (Col. Jay Vaughn)

Fort Huachuca, AZ 85613-7000
Tel: (602)533-1143

11. Capt. Ivery
JEWIC/SE
San Antonio, TX 78243-5000
12. Ms. Barbara Sherrell
HQ TRADOC
Office of the Deputy Chief of Staff for
Concepts,
Doctrine, and Development
Ft. Monroe, VA 23651-5000
13. Mr. Bill Dunn
Director of US Army Model Improvement and
Study
Management Agency
1900 Half Street SW
Washington, DC 20324
14. Mr. Mike Davis, Director
US Army CECOM Center for Signals Warfare
Vint Hill Farms Station
Warrenton, VA 22186-51000
15. Mr. Oliver Cathey
SPARTA, Inc.

Laguna Hills, CA
16. Mr. Ron Williams
Commander, US Army Intelligence Center and
School
ATTN: ATSI-TPO-AT
Fort Huachuca, AZ 85613-7000

17. Mr. Steve Cooper
US Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-TFS (Mr. Steve Cooper,
Bldg 63850)
Fort Huachuca, AZ 85613-7000

18. Technical Library
ATTN: ASQH-PCA-CRL
Fort Huachuca, AZ 85613-6000

19. Mr. Dick Porter
Head, Simulation Support Division
TRAC/WSMR
ATTN: ATRC-WEA (Mr. Dick Porter)
White Sands Missile Range, NM 88002

Tel: (505) 678-1901

20. Mr. Rich Maruyama
Commander
TRADOC Analysis Command
ATRC-RPD (Mr. Rich Maruyama)
Fort Monroe, VA 23651
Tel: (804) 727-3004

21. LtC. Tony Anconetani
Army Decision Systems Management Agency
US Artificial Intelligence Center
The Pentagon, Room 1E659
Washington, DC 20310
Tel: (202) 694-6904

22. Cpt. Pat Vye
US Army Combat Analysis Center
ATTN: ATZL-CG (Cpt. Pat Vye)

Fort Leavenworth, KS 66027-5000
Tel: (913)684-5623

23. Col. Donald E. Appler, Director
Scenarios and Doctrine Directorate
TRADOC Analysis Command
Fort Leavenworth, KS 66027-5200
Tel: (913)684-4011

24. Maj. Shuey Wolfe
Joint Warfare Center
Hurlburt Field, FL 32544
(904)884-5870

25. Mr. Don Blumenthal
Conflict Simulation Center
Lawrence Livermore National Laboratory
Box 888, L-315
Lawrence Livermore National Laboratory
Livermore, CA 94550
Tel: (415)423-7164

26. Dr. S. Christian Simonson
Lawrence Livermore National Laboratory
Box 888, L-315
Lawrence Livermore National Laboratory
Livermore, CA 94550
Tel: (415)423-7164

27. Dr. Ralph M. Toms
Conflict Simulation Center
Lawrence Livermore National Laboratory

¶Box 808, L-315
Livermore, CA 94550
Tel: (415)423-7164

28. Ms. Kathi Meyer, RAID Administrator
Computer Sciences Corporation
4045 Hancock Street
San Diego, CA 92110
Tel: (619) 225-8401

29. Dr. Daryl Morganson
Los Alamos National Laboratory
Los Alamos, New Mexico 87544
Tel: (505) 667-6553

30. Ms. Sara Tisdell
USA TRADOC Analysis Command
ATTN: ATRC-FFB (Ms. Sara Tisdell)
Fort Leavenworth, KS 66027

31. Maj. Rick Halek
Director, Army Model Management Office (AMMO)
ATTN: ATZL-CAN-DO (Maj. Rick Halek)
Fort Leavenworth, KS 66027

32. Dr. Carl Builder
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406-2138

33. Maj. Joe Trien
Army Model Improvement Project Management
Office
(AMMO)
ATTN: ATZL-CAN-DO (Maj. Joe Trien)
Fort Leavenworth, KS 66027

34. Mr. Doug Sizelove
ATTN: SAUS-OR (Mr. Doug Sizelove)

The Pentagon, Room 1E643
Washington, DC 20310-0102
Tel: (202) 695-0384

35. LtC. Jim Rungay
Scenarios and Doctrine Directorate
TRADOC Analysis Command
Fort Leavenworth, KS 66027-5200
Tel: (913) 684-4011

36. Mr. Kent Pickett, Director
Model Support Directorate
TRADOC Analysis Command

¶Fort Leavenworth, KS 66027-5200
Tel: (913) 684-4011

37. Mr. E. B. Vandiver III, Director
US Army Concepts Analysis Agency
8120 Woodmont Avenue
Bethesda, MD 20814-2797
Tel: (301) 295-1633

38. Lt. Tom Wong
Contracting Officer's Technical
Representative
Area B, Bldg 11a, Room 205
USAF Air Force Systems Command
Aeronautical Systems Division, ASD/XRX
Wright-Patterson AFB, OH 45433-6503
Tel: (513) 255-5035

39. Lt. Glenn Fye, Program Manager
Distributed Situation Development
Rome Air Development Center
Griffiss AFB, NY 13441-5700

Tel: (315) 330-3175

40. Dr. R. T. Hayes, Program Manager
Joint Theater-Level Simulation (JTLS)
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
Tel: (818) 354-4321

41. Mr. Alfred Silliman, Task Manager
Joint Exercise Support System (JESS)
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
Tel: (818) 397-9328

42. Dr. Joe Fearey
War Gaming Technologist and Scenario Planner
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
Tel: (818) 397-9322

43. Dr. Paul K. Davis, Director
Strategy Assessment Center
The RAND Corporation
1700 Main Street
Santa Monica, CA 90406
Tel: (213) 393-6912

¶
¶

Appendix A

Test Report for the Scenarist Automated Scenario Generation System

¶

Appendix B

Scenarist Automated Scenario Generation System Summary and Sample Demonstration Demo 1

¶

GENU01.FIL

¶

SPEC02.FIL

FndID(188)
FndTitle(SCENARIST™ AUTOMATED SCENARIO
GENERATION SYSTEM, FINAL REPORT)
FndDescription(SCENARIST™ AUTOMATED SCENARIO
GENERATION SYSTEM, FINAL REPORT)
FndKeywords(Scenarist; automated scenario
generaton; artificial intelligence; test and
evaluation; military scenarios)