SUPERVISORY METHODOLOGY AND NOTATION (SUPERMAN) FOR HUMAN-COMPUTER SYSTEM DEVELOPMENT

by

Tamer Yunten

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

H. Rex Hartson, Chairman

Robert C. Williges

Osman Balci

Oner Yucel

James D. Arthur

September, 1985
Blacksburg, Virginia

SUPERVISORY METHODOLOGY AND NOTATION (SUPERMAN) FOR HUMAN-COMPUTER SYSTEM DEVELOPMENT

by

Tamer Yunten

Committee Chairman: H. Rex Hartson
Computer Science

(ABSTRACT)

The underlying goal of SUPERvisory Methodology And Notation (SUPERMAN) is to enhance productive operation of human-computer system developers by providing easy-to-use concepts and automated tools for developing high-quality (e.g., human-engineered, cost-effective, easy-to-maintain) target systems. The supervisory concept of the methodology integrates functions of many modeling techniques, and allows complete representation of the designer's conceptualization of a system's operation. The methodology views humans as functional elements of a system in addition to computer elements. Parts of software which implement human-computer interaction are separated from the rest of software. A single, unified system representation is used throughout a system lifecycle. The concepts of the methodology are notationally built into a graphical programming language. The use of this language in developing a system leads to a natural and orderly application of the methodology.

2/12/86 MCR

I dedicate this work to my wife who encouraged me for years with an angel's patience.

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. H. Rex Hartson, for his invaluable guidance and constant encouragement during the course of this research and my writing of this dissertation. His continual and tireless help was instrumental in bringing this study to a successful completion.

I express my sincerest gratitude to Dr. R. C. Williges for invaluable advice and help in evaluating SUPERMAN.

I am grateful to: Dr. T. Lindquist for his suggestions which helped me create the supervisory concept of SUPERMAN, Dr. B. Shackel, and Dr. J. Greenstein for contributing to human-computer system concept, and Dr. R. W. Ehrich and the programmers under his management for providing the software tools which helped me in rapidly constructing a prototype graphical programming language and which consequently helped in communicating my ideas.

My special thanks are to Dr. A. I. Wasserman, and Dr. R. Jacob for reviewing a manuscript of SUPERMAN prepared for publication in 1983, and to Dr. D. Johnson for her editorial contributions. Also, I thank Dr. D. Johnson, J. Brandenburg, J. Callan, R. Larson, E. Smith, V. Bagrodia, and A. Siochi for diligently participating in a survey for evaluating SUPERMAN.

TABLE OF CONTENTS

SUPERVISORY METHODOLOGY AND NOTATION (SUPERMAN) FOR HUMAN-COMPUTER SYSTEM DEVELOPMENT ii

ACKNOWLEDGEMENTS iv

Chapter page

I. INTRODUCTION AND PROBLEM STATEMENT 1

The Need To Represent Both Human and Computer Components of a System 4
The Need for Cooperation Between the Human Factors Engineer and the Software Engineer 7
The Need for a Carefully Chosen Representational Technique 8
Representability of Pertinent Information . . 9
Spatial Arrangement and Symbology 10
The Software Development Process Needs to be Simplified 12
Software Lifecycle 12
Requirements Specification 13
Design 14
Coding 14
Testing 15
Maintenance 15
Verification and Validation 16
Shortcomings of the Lifecycle 17
Prototyping 19
Outline of Research Goals and Summary of SUPERMAN 21

II. RELATED WORK 26

Methodological Approaches Based on Data Flow . . 27
A Methodological Approach for Interactive Information System Development 30
Methodological Approaches Based on Control Flow 31
Holistic Approaches to Representing Human-Computer Systems 34
Attempts to Ease Phase-Product Transformations . 35
Detailed Design Techniques 36
Graphical Programming 38
Conclusions on Related Work 39

III. FUNDAMENTALS OF THE SUPERVISORY METHODOLOGY AND NOTATION . . . 41

Supervised Flow Diagrams . . . 42
Symbols for Functional Representation of Interactive Software . . . 44
Supervisory Structure . . . 46
A Graphical Programming Language . . . 48
Control Flow . . . 49
Sequence . . . 49
Decision . . . 51
Compound Decision . . . 52
Iteration . . . 53
Recursion . . . 55
Data Flow . . . 57
Data Flow Between a Function and Its Supervisor . . . 57
Data Flow Between a Function and a Storage Area . . . 60
Internal Work of a Function (Internal Code Block) . . . 61
Data to be Declared for a Function . . . 62
Supervision of Concurrent Functions . . . 63
An Example . . . 66
Comparison with Other Representation Techniques 72
The Relationship of the SFD to Operational Sequence Diagrams . . . 74

IV. INTERACTIVE SOFTWARE SYSTEM DEVELOPMENT . . . 77

Requirements Specification . . . 77
Design . . . 83
Coding . . . 84
Testing . . . 85
The Behavioral Demonstrator . . . 85
Human-Computer System Maintenance . . . 86

V. HOLISTIC APPROACH TO HUMAN-COMPUTER SYSTEM DEVELOPMENT . . . 88

The Operational Procedure Structure . . . 92
The Human-Computer System Lifecycle in the Holistic Approach . . . 97
Requirements Specification . . . 98
Role of the Operational Procedure Structure in System Maintenance . . . 99

VI. EVALUATION OF SUPERMAN . . . 100

Evaluation Method . . . 101

A Survey-Based Evaluation of SUPERMAN 105
Comments on Strengths of SUPERMAN 109
The Stated Needs of the Users 112
The Author's Comments on the Evaluation Activity 115

VII. CONCLUSION AND FUTURE DIRECTIONS 117

REFERENCES 119

Appendix page

A. A QUALITATIVE EVALUATION SURVEY FOR SUPERMAN . . 129

B. A DETAILED VIEW OF THE SURVEY RESULTS 205

Ease of Learning 205
Project Management 207
Miscellaneous Comments on Project Management 214
Generality of Application 216
Miscellaneous Comments on Generality of Application 218
SUPERMAN User's Efficiency 222
Miscellaneous Comments on SUPERMAN User's Efficiency 229
Quality of Target System 230

Chapter I

INTRODUCTION AND PROBLEM STATEMENT

The SUPERvisory Methodology And Notation (SUPERMAN) has been developed, as part of the Dialogue Management System (DMS), in the environment of a multidisciplinary project involving the Human Factors Laboratory and the Computer Science Department. DMS is an automated system which aids in the development of other large scale interactive systems with high quality interfaces. SUPERMAN focuses on the process of human-computer system development, and provides conceptual and automated methodological support to DMS. A characteristic feature of DMS is the separation of human-computer dialogue from computational code [HARTH82] in the target system being developed. The dialogue part of a system is developed by a human factors specialist called a dialogue author [HARTH82]. To reflect the separation of software into computational and dialogue components, the roles of system designers are divided into two classes. All human engineering activities, including dialogue authoring [JOHND82, HARTH84], are performed by the class which will be denoted generically here as HFE (human factors engineer). Similarly, the class called SWE (software engineer) represents roles which typically include systems analysts and programmers.

The goal of any system development methodology is to provide for low development cost and high product quality. While numerous methodologies have been developed to facilitate the complex task of creating application systems, few of these methodologies are as cost-effective as desired. Further, although consideration of human factors is important for a high-quality product, only a few methodologies seem to support human-engineering activities.

To facilitate cost-effective construction of human engineered systems, several problems must be addressed. These include:

1. Current methodologies do not provide a notation for representing both human and software components of a system (i.e., a human-computer system). However, for proper human factors analysis, a representation technique which models the cooperation of the human and the computer (e.g., interactive functions, manual functions, control allocation, management functions) is essential.
2. A disciplined approach to human-computer system development requires cooperation of the HFE and the SWE roles. To support this cooperation, a methodology must embody the concepts used and the activities performed by both of these roles.

3. Great care is required in the choice of a technique for representing system designs. It must support the effective cooperation of the designer roles by unifying the shared concerns of these roles, it must separate the discipline-specific concerns, and it must be easy to use by designers in both roles.
4. The software development process needs to be simplified. The lifecycle approach based on phase-product transformations has many shortcomings, and the methodologies which practice this approach are difficult to learn and labor intensive. There is a need for a new approach which eliminates the transformations but preserves the lifecycle concept.
5. Prototyping activities need to be integrated into the disciplined approach. To provide for continuous prototyping activity, system specifications must be executable and must directly evolve into design and implementation.

These problems are discussed in the five sections that follow.

1.1 THE NEED TO REPRESENT BOTH HUMAN AND COMPUTER COMPONENTS OF A SYSTEM

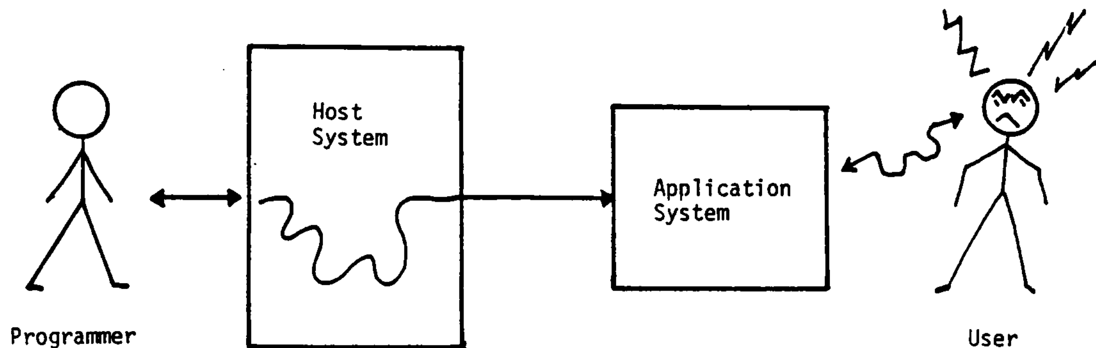

Figure 1. Ad-hoc system production.

The discipline of software engineering was established to replace the ad-hoc system production illustrated in Figure 1. During its evolution, the major concerns of the discipline have been developmental problems such as the high cost of software, incomplete and/or incorrect systems, and late projects. During the past decade, software engineering has largely succeeded in improving the system development process. On the other hand, as Figure 2 illustrates, such improvement has not distinguished with regard to the development of human-engineered systems. In the context of interactive computer systems, Thomas and Hamlin [THOMJ83] cri-

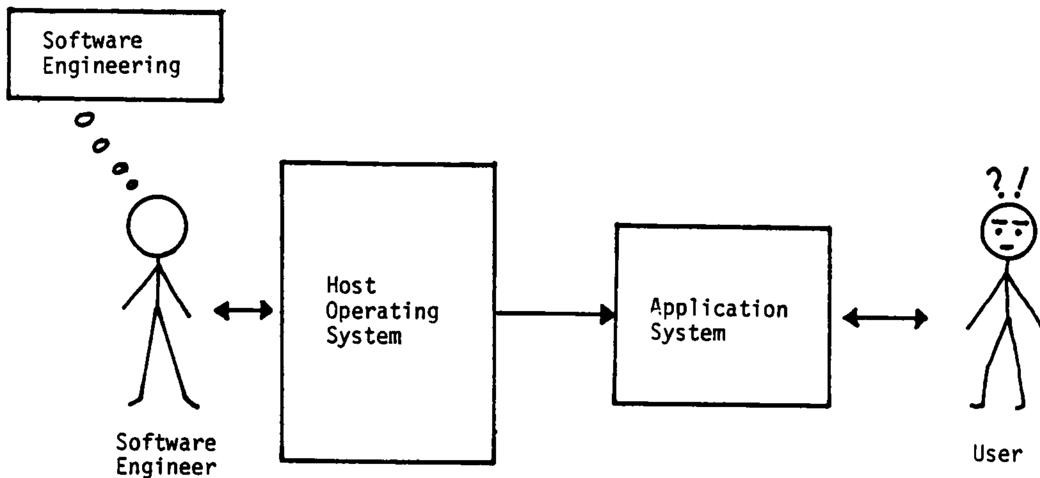

Figure 2. Use of inadequate methodologies in human-computer system development.

ticize most software engineering methodologies because they " . . . focus on description of algorithms in computer terms rather than user-oriented systems." They express the need for a different view of software construction from the one supported by traditional software engineering approaches. In their view, a methodology user must have techniques to detail the individual functions that an end-user will see, the methodology must provide some structure to the overall set of functions provided, and the methodology must

provide techniques for specifying and coding the interaction sequences in an effective way. Likewise, in his classification of software programs (with regard to validation), Lehman [LEHMM81] defines a class of programs which " . . . are embedded and executed in an environment and in some sense control activities and/or events in this environment . . . one must therefore predict its impact on its operational environment and on the people who are to work with it." Kling [KLINR81] illustrates the indirect impact of such programs on their environment with a survey of six banking systems. While some of these systems have increased the variety and scope of their users' work, some others have led to more constrained and repetitive jobs. Lehman's suggestion [LEHMM81] for predicting such impact and dealing with it is that the designer must perceive how the users will be using the system once it is put into operation. Draper and Norman, in a recent publication [DRAPS85], support the above view as follows: "When a programmer implements a system with a user interface, he or she is not only defining what the machine will do but also defining what the user can do . . . the user should be treated as part of the system being designed."

1.2 THE NEED FOR COOPERATION BETWEEN THE HUMAN FACTORS ENGINEER AND THE SOFTWARE ENGINEER

> "The starting point for our consideration for human factors discipline is obviously system development. Conceivably we could confine our interests merely to use of the equipment after the equipment had been designed and produced. In that event, however, we would merely be describing how people use the equipment, which is interesting but not crucial." [MEIST71]

As the above quote from Meister indicates, the HFE is involved in all phases of a system life. The HFE activities include:

- evaluating the function and control allocation configurations in view of performance requirements (e.g., in the design of time critical embedded systems);
- matching software characteristics to user capabilities (e.g., adapting dialogue to the user's expertise level);
- evaluating the interaction sequencing (e.g., removing time consuming repetitions; and
- arranging controls and displays (i.e., form of the dialogue).

To perform these activities, the HFE needs to cooperate with the SWE. However, as Figure 3 illustrates, the absence of a formal tool for inter-role communication complicates the development process. Hence, an easy-to-use technique that integrates the representational needs of both discip-

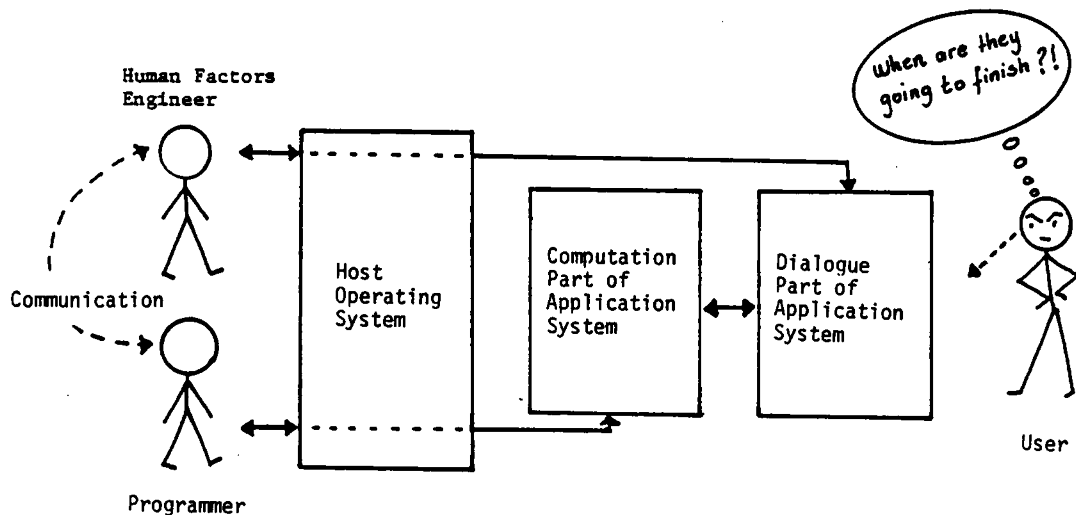

Figure 3. Human factors engineer and the software engineer without a human-computer system methodology.

lines is an essential component of a support methodology. This representational technique is needed whether these roles are played by different persons or by the same person.

1.3 THE NEED FOR A CAREFULLY CHOSEN REPRESENTATIONAL TECHNIQUE

A representation (e.g., design representation) is a physical supplement to human memory; it is part of a human's "external memory" [ANDEB75]. The ability of a technique for representing system designs to convey pertinent information,

and the style of the representation, may directly affect the productivity of system developers, both in problem solving and communication.

1.3.1 Representability of Pertinent Information

Things which cannot be (or are not) represented external to human memory, if they need to be remembered (or communicated), have to be stored in human memory. Limitations of human memory are well known [ANDEB75, BOURL81]. Tracz [TRACW81] describes how representations (also called external memory) support human memory as follows: "External memory, a very important component of human information processing, compensates for the slow fixation times associated with the long term memory and frees the limited short term memory resources for use in problem solving (creativity, concentration, etc.)." Obviously, the more pertinent things a technique can represent, the more a human can utilize the advantages of external memory. Some pertinent things which need to be represented while developing human-computer systems are: functions at abstract levels, who (e.g., a human, a computer, a team of humans and/or computers) performs what functions, non-functional requirements, data flow and/or control flow at an abstract level and thorough levels, communication and synchronization of concurrently operating ob-

jects, external behavior of a system's operation, development-related information, etc.

1.3.2 Spatial Arrangement and Symbology

In addition to the capacity for representation, a technique must provide for fast access to represented information. Due to the limited time that short term memory can hold information, delayed access poses a potential loss of information in short term memory [TRACW81]. Also, because of the limited number of things short term memory can hold, if attention is devoted to a lower level process (e.g., searching for the boundaries of interleaved code blocks), fewer resources are available for higher-semantic processing [ANDEB75]. As quoted by Iverson in his 1979 ACM Turing Award Lecture [IVERK80], A.N. Whitehead states: "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race."

Spatial organization of objects seems to increase the speed of access to information. Anderson, based on cognitive psychology experiments, states that it is difficult to retrieve a perceptually embedded object (e.g., a word or phrase buried in a large amount of text) from external memory [ANDEB75]. However, some spatial organizations are bet-

ter than others. In experiments concerning forward and backward traceability, Sheppard, Kruesi, and Curtis [SHEPS81] have shown that sequentially arranged and indented representation of code is less efficient (in terms of processing time) to a graphically oriented spatial branching arrangement which provides a clear guide to the decision structure of the program. Further, good spatial organization enables the user to handle information more easily, for spatially organized visual images are "especially easy to recall . . . the spatial organization serves as a kind of 'glue' to hold features together" [ANDEB75].

While one representational dimension for increasing the ability to identify system concepts, components, and structure is spatial organization, another dimension is symbology [SHEPS81]. Paivio [PAIVA71] states that perceptual objects and pictures activate the human imagery system more directly than linguistic stimuli. Shneiderman agrees[SHNEB83], pointing out that spatial relationships are more quickly understood with visual than with linguistic representations. The importance of symbology was recognized by the designers of the Xerox STAR interface [SMITD82] who chose iconic menus over textual ones. In human-computer systems, some physical and/or conceptual objects which are candidates for symbolic representation include: various types of functions (e.g.,

human, computer, human-computer, dialogue, etc.), storage areas, devices, objects in the environment, etc.

The above discussion leads to the following conclusion: in a system's representation, a structured and spatially arranged textual and/or graphical information, enriched by a notation that aids differentiation among the system components, can enhance the performance of the system developers.

1.4 THE SOFTWARE DEVELOPMENT PROCESS NEEDS TO BE SIMPLIFIED

The software lifecycle is a central concept of the current system development technology [WASSA80]. The following section briefly presents the conventional lifecycle and the associated verification and validation activities. The reader who is familiar with the subject may wish to continue with Section 1.4.2 which discusses some shortcomings of the current lifecycle concept.

1.4.1 Software Lifecycle

The software development lifecycle typically consists of the following, or similar, phases: requirement specification, design, coding, testing, and maintenance (evolution). Software verification and validation are also important lifecycle activities.

1.4.1.1 Requirements Specification

Requirements specification is the process of understanding and recording what is needed or what must be done [FREEP83a, HEITC83, LUNDM83]. Yeh [YEHRT82] divides system requirements into two general categories: functional and non-functional. Functional requirements specification is a representation of the functions to be performed by the system under development. Non-functional requirements are the constraints that a system must satisfy (e.g., human and/or computer response time.) Desirable characteristics of a specification are: conceptual cleanliness, lack of ambiguity, correctness, completeness, consistency, testability, traceability, and executability [ROMAG85]. Currently, the kind of specification presented to the user or customer to convey the requirements is in one of the following forms: textual list of requirements, a model (e.g., a flow diagram) interpreted manually by designers of the system, or a working model (e.g., a prototype) of the proposed system [MASOR83].

Roman [ROMAG85] argues, as a major weakness of these kinds of requirements specification methodologies, that a broad integration of functional and non-functional requirements has not been accomplished.

1.4.1.2 Design

While the requirements specification activity defines what is to be done in a software system, the design activity determines how to do it [WASSA77]. The design phase of the software lifecycle defines the physical and logical structures that satisfy the requirements [ROMAG85]. Freeman's definition of the design process is that it is a process of building a representation of the object to be created [FREEP83b]. In general, design is divided into two distinct phases [WASSA80]: architectural design and detailed design. During architectural design, the issues of concern are the construction of modular structure, the communication interfaces among modules, and communication paths between modules and databases. During detailed design, the architectural design representation is refined by specifying the internal logic of the modules, the detailed structure of databases, etc. [WASSA80].

1.4.1.3 Coding

Coding, once the major activity of software development, now occupies only about 20% of a system lifecycle [WASSA80]. Freeman [FREEP83b] makes a clear distinction between design and coding with regard to the output of these activities. The output of coding is information in machine

executable form. The output of the design is an abstract representation of this machine executable form.

1.4.1.4 Testing

Testing is the process of experimentally demonstrating the consistency between a program's behavior and its specification. According to Adrion, Barnstad, and Charniavsky [ADRIW82], testing activities are: " . . . obtaining a valid value from the functional domain (or an invalid value from outside the functional domain, if we are testing for robustness), determining the expected behavior, executing the program and observing its behavior, and finally comparing that behavior with the expected behavior." Testing requires an executable object (e.g., a program) and is conventionally performed during and after the coding phase.

1.4.1.5 Maintenance

Maintenance (now called evolution) is the activity of adapting a system to new or changing requirements. Over 50% of the lifecycle cost of a software system is due to maintenance [ADRIW82]. Maintainability is built into a system during design phase [MYERJ78] and largely depends on the management of complexity [STEVW74]. Some means of reducing system complexity during design and maintenance are: repre-

senting a system as a hierarchy, maximizing the independence among the parts of a system [MYERJ78], and modularizing a system. Some criteria for modularization are: module size [HOSIJ78], the information hiding capability of a module [PARND72], and module strength and module coupling [MYERJ78, STEVW81]. Module strength and module coupling are inversely related, and serve as criteria for evaluating the maintainability of a system. Single function modules which are coupled only by data communication are the desirable modules. Modules which hide (e.g., encapsulate) information that is likely to change (e.g., design decisions, data structures) increase a system's maintainability, and support team work.

1.4.1.6 Verification and Validation

Verification and validation activities have evolved to provide for the correctness of the pre-implementation phase products. Lehman [LEHMM81] defines verification as the process of ensuring " . . . the consistency and completeness of each phase model both in itself and in relation to its parent model; that is, the correctness of the transformation process and the detail that has been added, and therefore the model itself." Since an initial specification might not be a correct definition of user needs, verifying that a program meets its specification is not same as demonstrating

that the program is a correct solution for the problem. Ensuring that the specification and/or the program meets the user's needs is termed validation. As Boehm states, validation activity includes " . . . determining the fitness or worth of a software product for its operational mission" [BOEHB84a]. Validation is " . . . a _judgmental_ activity based on measurement and human assessment of _expected effectiveness_, _usability_, _value_ and _cost effectiveness_ in the _operational_ environment" [LEHMM81]. Boehm [BOEHB83], among others, states that, to avoid costly changes, validation should be performed continuously throughout the lifecycle. Obviously, continuous validation begins in the requirements engineering environment, and the success of validation activity largely depends on the success of communication among the developers and the users and the suitability of the communication tool. Rzepka [RZEPW85] describes such a tool as follows: "A simple, easy-to-use pictorial scheme would be an ideal notation and presentation medium for the requirements engineer."

1.4.2 Shortcomings of the Lifecycle

One problem with the conventional lifecycle is that development often begins with data flow and the specification of control flow is postponed until the detailed design

phase. However, in human-computer system development, control sequences are often important much earlier (e.g., in the requirements specification phase). For example, to communicate with an interactive system user, and to evaluate interaction sequencing, one needs to specify the operational sequences. Also, for certain types of software systems, such as embedded systems, representation of "control is of paramount importance" [ZAVEP82b].

Most lifecycle problems are due to redundant representations of the target system. Because different modeling techniques are often employed at each pre-implementation phase (e.g., requirements specification, design representation, code), large amounts of time are spent transforming the representation of one phase to that of another [ZAVEP84, RAMAC84, HAMIM83]. Errors are introduced in this way, and it is expensive to re-examine several representations for changes and to verify the correctness of transformations. No existing method can show the correctness of a verification activity. Also, the lack of a formal (e.g., executable) specification delays the major part of testing and validation until very late in the lifecycle (e.g., after the coding phase). A major problem with testing is that it is frequently most difficult to obtain the description of the expected behavior [ADRIW82]. Prototyping has been used as a

way of overcoming these difficulties and is discussed in the next section.

1.5 PROTOTYPING

The major motivation behind prototyping is the need to communicate with the user and to be sensitive to changes in user requirements over a long period of development. Some experience has shown that the use of a prototype is an excellent method of accomplishing this [GOMAH82]. Users find it easier to evaluate a working system than a paper document [GOMAH82], and can relate what they see directly to their needs [MASOR83, RAMAC84]. During prototyping, the user can find answers to questions about a system feature: "Is there a need for it?", "Does it have a positive contribution?", "What does it contain and what does it do?" [LUNDM83]. Prototying is also an effective way of evaluating human-computer dialogue [WILLR84, THOMJ83]. After a multiproject experiment, Boehm, Terence, and Seewaldt report that protoyping, compared to specifying, yielded products with about 40% less code and 45% less effort [BOEHB84b].

But there are some disadvantages to prototyping, compared to specifying [MASOR83, ALAVM84, BOEHB84b]. Prototyping can result in a higher initial cost for the requirements specification phase, and, due to difficulties in managing

and controlling the design process, the products have a lower functionality and robustness. Roman [ROMAG85] outlines the trade-offs as: "Rapid prototyping seems to lead to less code, less effort, and ease of use, while the traditional approach is characterized by better coherence, more functionality, higher robustness, and ease of integration." Alavi [ALAVM84], after an experimental comparison of prototyping and specifying, discusses "who should use prototyping", "where to use prototyping", and "how to prototype".

Nevertheless, a compromise is possible. Roman [ROMAG85] suggests that prototyping should be mixed with the lifecycle approach in order to utilize the advantages of both. This theme which is a basic concept of SUPERMAN was also emphasized in a recent software engineering workshop: "Design and implementation should be a continuous process of evolution from executable requirements specifications" [BABBR85]. In SUPERMAN, the requirements specification is executable and therefore serves also as a prototype. This specification is the front-end of a system structure, the construction of which continues in the design and coding phases.

1.6 OUTLINE OF RESEARCH GOALS AND SUMMARY OF SUPERMAN

The problems discussed in the previous sections have been the driving force of this work. However, the author did not address these problems one at a time, but rather, in search of a unified methodological approach which encompasses many solutions, addressed the problems as a whole. This approach led to the methodological view of the Dialogue Management System illustrated in Figure 4. As observed in the figure, the major goal of this work is to provide for effective cooperation of the software engineering and the human factors engineering roles. This goal leads to the following subgoals:

1. to develop a comprehensive representation technique which notationally and structurally identifies target system parts that are of interest to above roles, as well as to system users;
2. to provide the means for enhancing productive operation of people in the design roles (e.g., to provide for: ease of learning and use of the methodology, ease of communication, effort minimization, automated support, and support for managing the developmental complexity); and
3. to provide for high-quality (e.g., human-engineered, correct, maintainable) target systems.

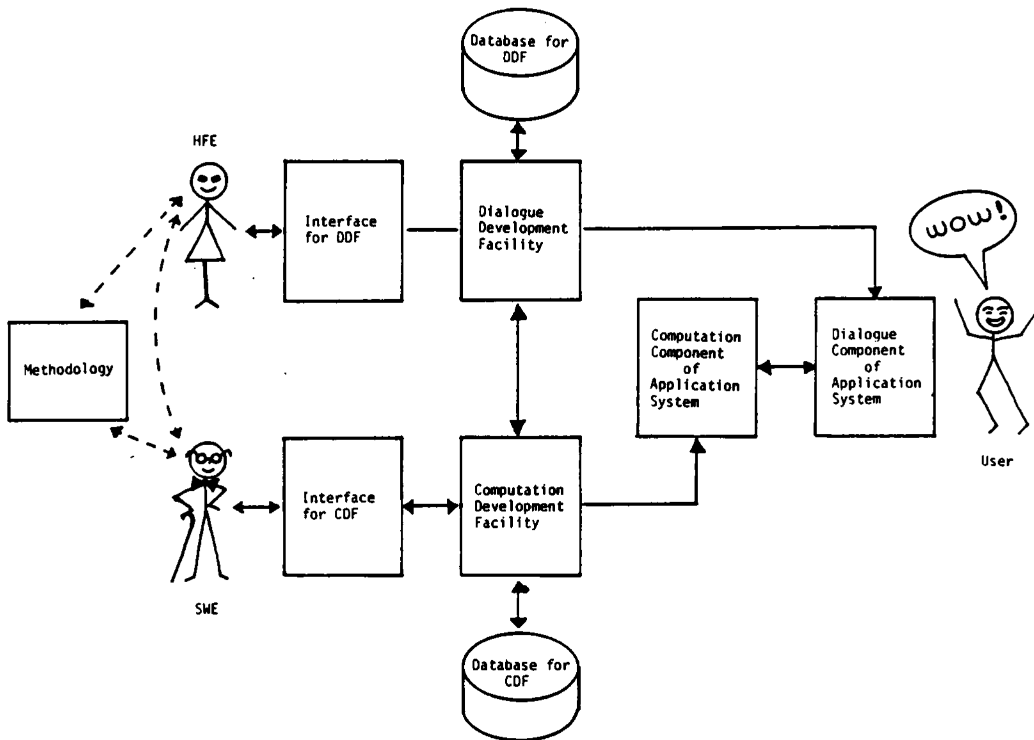

Figure 4. The Dialogue Management System.

The above goals subsume solutions to the stated problems. The author believes that SUPERMAN and the Dialogue Management System provide the solutions to these problems in a unified framework. SUPERMAN is summarized below.

The construction of a software system includes the requirements specification, design, and coding activities. Most methodologies produce different representations for each lifecycle phase. SUPERMAN produces a unified target

system representation which is used in all phases of the system's life. This graphical structure is a functional representation of a system operation, and includes the information structures, data flow, and control flow. The functions may be performed by humans, by computers, or by cooperation between humans and computers. The non-functional attributes of the operations (e.g., user and/or computer response time), and the development related information (e.g., design decisions, personnel allocation) are formally associated with, and organized around, the functions of this structure.

With SUPERMAN, the requirements part of a target system representation is the specification of the required operation. This requirements structure, when developed for interactive software, represents cooperation of the human and the computer and includes both dialogue and computational functions. The requirements structure executes on the computer and, in addition to realizing the specified sequencing, serves as a communicator among the dialogue and computational functions. The requirements structure can be constructed with regard to the user's functional needs (e.g., user's view) only, or it can include the designer's technical view. If constructed from the user's view, this structure may be further enriched with the designer's view

(e.g., details of a logical data structure, control logic for adapting the dialogue to user's expertise level). In the combined view, the designer may choose to hide or to include the various technical design issues. The combined view which hides the technical issues represents a logical design (i.e., logical view) of a system's operation. The design structure is an extension of the requirements structure and implements the logical view. Technical issues such as physical architecture, modularization, mutual exclusion, and data encapsulation are considered during construction of the design structure (although they may also be considered during construction of the requirements structure). Most of the data structures (e.g., global structures) which appear on the requirements structure are implemented in the design phase, and may also be encapsulated. In the coding phase, this graphical structure is further refined by a high level textual language, and the whole structure is compiled into executable code.

SUPERMAN supports any combination of top-down, bottom-up, user-oriented, or design-oriented approaches to system development. The executable requirements structure represents the behavior of a system (e.g., interactive behavior), and can be directly used for empirical interface testing. For validating the user requirements and evaluating the dia-

logue, the designers may choose the prototyping approach. The designers may also begin the system modeling and analysis at a high level which does not differentiate between the human and computer functions. This model is notationally refined to allocate the tasks (e.g., manual functions, interactive functions, control flow management, data flow management, management of other functions) between the human and the computer.

Chapter 2 discusses the related work. SUPERMAN is presented in chapters 3, 4, and 5. Chapter 3 introduces the fundamental concepts of the methodology, and an executable specification language. The use of SUPERMAN for developing interactive software systems is presented in Chapter 4. Chapter 5 extends SUPERMAN to include the human's non-interactive operation with embedded interactive software tools. Chapter 6 is on the evaluation of SUPERMAN. The conclusions and future directions are in Chapter 7.

Chapter II

RELATED WORK

Numerous methodologies fully or partially cover the lifecycle activities with varying levels of automation. A questionnaire-based evaluation of twenty-four methodologies is presented by Porcella, Freeman, and Wasserman in [PORCM82]. The ease of use of these methodologies is largely determined by the modeling techniques they use. Yet, none of the current modeling techniques provide for all the representational needs (data flow, control flow, human behavior, information hiding) discussed in the first chapter. Nor are many of the current techniques widely applicable to a large class of systems. Either most methodologies use techniques which assume a data flow model, or begin with a control flow model. Most conventional methodologies delay the specification of control flow until the detailed design phase. Although some techniques represent both data and control flow (e.g., finite state state machines), they are difficult to use and/or incomplete. Only a few methodologies address human-computer dialogue, or represent the user as part of a system. Many of the modeling techniques are not suitable for all phases of the system lifecycle, and, consequently, the models produced are transformed into other

models. Graphical modeling techniques (e.g., HIPO, data flow diagrams, structure charts, transition diagrams, Petri nets, stimulus response paths, activity diagrams, operational sequence diagrams) are widely used, and some researchers are working towards graphical programming.

2.1 METHODOLOGICAL APPROACHES BASED ON DATA FLOW

Some methodologies model a system in terms of functions (e.g., modules) and data flow among the functions and/or data structures. These system models do not represent control flow.

The Structured Analysis and Design Technique (SADT) [ROSSD77, ROSSD85], an example of a methodology based on data flow, can be used to specify the requirements for a large class of systems. The SADT model is a structure of activity (e.g., function) diagrams which represent the logical architecture of a system. Although the SADT model allows for function allocation activity, the activity diagrams hide the operational sequences (human-only and human-computer), and do not represent the human-computer dialogue. As Wasserman [WASSA80] states, "The modules built in SADT often do not lend themselves to immediate transformations into design or implementations. It takes considerable experience and training to become a skilled SADT user." For other

phases of the lifecycle, SADT interfaces with other methodologies (e.g., PSL/PSA, Structured Design, Jackson Method, HIPO, Nassi-Shneiderman, Petri nets)[ROSSD85].

Structured System Analysis (SSA) [WEINV80] and Structured Design (SD) [STEVW74, YOURE79, MYERJ75, MYERJ78, WEINV80, STEVW81] use the concept of the "process" as the basis of design to build a system around the functions that the final system will provide. For system modeling, SSA uses the data flow diagrams (DFDs) [WEINV80] as front end for the Structured Design methodology. Data flow diagrams are transformed into the structure chart (that indicate which modules call which other nodules) of the Structured Design methodology, and both of these models hide the operational sequences.

PSL/PSA [TEICD76] is also based on data flow, and automates the requirements specification process with a machine-processable specification language, producing complete documentation to be used in the design phase. PSL/PSA is detailed and difficult to learn and is sometimes used "after the fact" to produce documentation, rather than to derive the design.

Some methodologies use data as the basis of design and derive the program structure from data definitions. Examples of such data-oriented approaches include: the Warnier-

Orr [WARNJ81] methodology, DAta oriented DESign (DADES) [OLIVA82], Logical Construction of Software (LCS) [CHAND80], and the Jackson System Development methodology [JACKM75, JACKM83]. While the first three are program design methodologies, the Jackson System Development methodology models a system as a set of communicating concurrent processes.

Another methodology emphasizing concurrent systems is one called Design Realization, Evaluation, And Modeling (DREAM) [RIDDW81], which provides support tools oriented toward architectural design. DREAM, for requirements specification and coding activities, relies on other methodologies. While design notation of DREAM defines what the inter-module interactions are, it does not represent how the interactions are performed.

While methodologies based on data flow have been used successfully to develop functional software, they do not provide enough help to the designer who is concerned with the human's role during the operation of a system. For although these methodologies begin system modeling using data flow based techniques, such modeling must eventually include control flow. The data flow models, to include control flow, are transformed into other models - an expensive and inaccurate process. Also, data flow models are not executable, and cannot be used for prototyping.

2.2 A METHODOLOGICAL APPROACH FOR INTERACTIVE INFORMATION SYSTEM DEVELOPMENT

The User Software Engineering (USE) [WASSA82a, WASSA82b], one of the few methodologies that specifically addresses interactive information system development, is based on state transition diagrams. USE is a skillful incorporation of techniques drawn from established methodologies. For example, it makes use of data flow diagrams from SSA for analysis, structure charts from Structured Design for architectural design, a PDL for detailed design, and a methodology-specific textual language, PLAIN, tailored for interactive information system implementation. In requirements analysis, state transition diagrams are used for modeling the human-computer interaction. This model is automatically encoded in a machine-processable form and interpreted by a transition diagram interpreter for rapid prototyping of the interface.

The transition diagram in USE is a directed graph, the nodes of which represent dialogue and the arcs of which represent machine actions. A major shortcoming of this model is that it can represent only the user-stimulus dependent behavior of the machine. For example, using this model, one cannot automatically adapt the dialogue to the user's expertise level. Consequently, because these diagrams are constructed with the user's view, the designer's

view must be realized separately (e.g., in a structure chart). Also, the transition diagrams implicitly derive the functional structure as a structure of dialogue, and, although used for information systems, this model hides the information structures behind the implicit computational functions. Finally, a stated by Wasserman [WASSA82b], these diagrams are problematic because they are primarily suitable for command-oriented systems (provided that the operations are known in advance) and are less suitable for other systems such as data base systems, and because they do not embody the performance requirements.

2.3 METHODOLOGICAL APPROACHES BASED ON CONTROL FLOW

The methodologies which are based on control flow mostly address embedded (e.g., real time) systems. As stated by Zave, "control is all-important in embedded systems and must be represented in any intelligible model" [ZAVEP82a]. Typical examples of methodologies based on control flow are PAISLEY, SREM, and SYSREM. These methodologies, in the requirements specification phase, differ from the conventional methodologies in that, rather than only specifying what is to be done, they combine this specification with how it is to be done (i.e., they specify the operation of a system). Some current literature states that these methodologies dif-

fer from those based on data flow because of their operational approach. However, most methodologies which are based on data flow also model the system operation. The difference is that, while a control flow model makes an operation more visible, a data flow model makes it less so.

One of the few specification techniques which addresses embedded systems is PAISLEY [ZAVEP82a, ZAVEP84], a textual language for simulating operation of an embedded computer system and the objects in its environment. The specifications produced in PAISLEY hide the external behavior of a system, but represent the internal mechanisms which generate the external behavior. With PAISLEY, the environment of a system (e.g., humans, machines) can be simulated as digital processes. As stated by Zave, PAISLEY is not easily understood by a user; it is "too technical" and, for validating the user requirements, the external behavior is demonstrated by executing a program which interprets the specifications written in PAISLEY. Hence, except through prototyping, a PAISLEY model does not help in requirements related communication among the users and designers. For such communication, Zave suggests the use of diagramming techniques.

The Software Requirements Engineering Methodology (SREM) specifically addresses real-time systems [ALFOM85, SCHEP85]. SREM is based on the finite state machine model,

which has been criticized for its redundancy, incompleteness, and inconsistency [ROMAG85], and because it permits no explicit parallelism nor modeling of the environment [ZAVEP82b]. R-nets of SREM enrich the finite state machine model with stimulus-response paths, by mean of a Petri-net-like notation. R-nets are manually translated into the textual Requirements Specification Language (RSL). Alford's reasoning [ALFOM85] for the choice of the finite state machine model (rather than functional decomposition) is that " . . . the hierarchy of functions model does not explicitly represent the conditions or sequences of processing; requirements generated under it are ambiguous with respect to them" [ALFOM85]. As a consequence of the augmentation of R-nets to overcome the limitations of the finite state machine model, SREM is "labor intensive and hard to learn" [SCHEP85]. SREM has been recently extended to include functional decomposition, and this new methodology is called System Requirements Engineering Methodology (SYSREM). Alford states that one of the "unique" features of SYSREM is the use of supervisory modules which monitor the concurrent functions [ALFOM85]. In both SYSREM and SREM, data flow is separately represented using a PSL/PSA like notation.

2.4 HOLISTIC APPROACHES TO REPRESENTING HUMAN-COMPUTER SYSTEMS

A holistic approach to human-computer system analysis takes a broad view which considers both humans and computers as system components. Some types of human-computer systems (e.g., embedded systems) are characterized by high-performance requirements and, consequently, consideration of human capabilities is essential in automation.

A popular industrial engineering tool for modeling human-machine systems is operational sequence diagrams (OSDs) [MEISD71]. OSDs represent the activities of elements of a system, including humans, machines, and objects in the environment. In verifying fulfillment of performance requirements, time scales on OSDs are used to compare alternative function allocation configurations. Another representational scheme for the evaluation of human requirements with regard to operational feasibility is the Design Analysis System (DAS), which uses information flow diagrams in conjunction with operational flow diagrams [ENOSJ78]. The powers and limitations of other methodologies (e.g., SADT, SSA & SD, PAISLEY, SYSREM) which can be used with the holistic approach were discussed earlier.

2.5 ATTEMPTS TO EASE PHASE-PRODUCT TRANSFORMATIONS

Several methodologies attempt to eliminate the problems of inter-phase transformations. Information Systems work and Analysis of Change (ISAC) [LUNDM79] eases the transition to the coding phase by the attachment of executable code to diagrams representing the activities, data, and procedures of target systems. The System ARchitects Apprentice (SARA) [PENEM81], a design environment for the construction of concurrent systems, provides for automatic generation of code "skeletons" from module specifications. Systems such as ECL, JOSS, and MUMPS attempt a language-based unified software approach to system development, but only partially cover the lifecycle phases [WASSA81a]. Emphasis on a unified approach is also observed in the Higher Order Software (HOS) methodology [HAMIM76, MARTJ82]. In HOS, the system model is a tree of input-function-output nodes. Key-word operations embedded in this structure specify control flow, and each operation is associated with a set of data flow rules. Although this model can be compiled to a target language, the data flow rules make it difficult to use, and it is mostly suitable for a structure of procedure calls. The need for a unified design representation containing both data flow and control flow is also observed by Iwamoto and Shigo [IWAMK81]. However, by choosing either a control flow or a

data flow model at each refinement step, they provide an inconsistent solution.

As discussed above, some software methodologies attempt to ease the difficulties of inter-phase transformations. ISAC and SARA, by facilitating linkage between the development phases, partially succeed in their attempts. The others seek a unified approach to system development but provide incomplete solutions. A complete solution to the problem requires a unified representation technique which covers the lifecycle phases with continuity. As mentioned in Chapter 1, SUPERMAN provides this technique.

2.6 DETAILED DESIGN TECHNIQUES

A major activity of detailed design is to define the logic of software functions. Commonly used tools for this are PDLs, flowcharts, Nassi-Shneiderman charts, and programming languages. Flowcharts and Nassi-Shneiderman charts [NASSI73] specify control flow, and show a clear separation between control statements and non-control statements, but do not indicate data flow. PDLs provide textual, block structured, natural-language-like logic specifications, but they mix control and non-control statements together, and do not provide for the notational representation of semantics.

The above techniques are also directly used for representing programs. Several experimental studies have been performed to compare the effectiveness of these techniques in composing, comprehending, debugging, and verifying computer programs. The results of these studies (performed by different researchers) are mostly conflicting. Shneiderman and Mayer [SHNEB81] state that flowcharts may be an aid in some situations and a hindrance in others. Ramsey et al. [RAMSH83], after an experimental study, report that programmers who used PDLs produced higher quality programs than the programmers who used flowcharts. On the other hand, Sheppard, Kruesi, and Curtis [SHEPS81] state that complex procedural information is more effectively presented with flowcharts than with textual representations (e.g., PDLs).

Obviously, neither of these techniques is satisfactory by itself. A more satisfactory technique must at least integrate the functions of the above techniques. As stated by Shneiderman and Mayers [SHNEB81], the resolution of the "flowchart question" depends on the larger question of what types of supplementary representation help programmers build the internal semantics. For program construction, a minimal set of such information includes: functions (e.g., high-level functions, low-level operations), control flow, data flow, data structures, concurrent operations, and a continu-

ous conceptual structure which integrates these and provides for levels of abstraction and spatial organization of information. In SUPERMAN, such information is integrated into a graphical programming language.

2.7 GRAPHICAL PROGRAMMING

Throughout the history of computer programming, programming languages have largely affected the ease of developing software. To make the programming task easier, machine languages have been abstracted by assembly languages which, in turn, have been abstracted by high level languages. Before each of these transitions, developers had manually played the role of language processors. In the current state of the art there is a similar situation.

Most existing methodologies start with a manually produced graphical system representation which, at implementation time, is manually converted to code. Also, several diverse projects exist in which a corresponding graphical representation is produced after the source text is coded. This graphical notation is used for verification and validation [FUJIM78], for reduction of complexity and highlighting of system composition [PAIGM77], for representing program language semantics [PRATT71], and for ease of analysis and modification [YAUSS81]. Fitter and Green [FITTM79] state

that diagrams make good computer languages when they provide a perceptually clear encoding of information, are used to restrict the designers to "good structures", are easily revisable, and are a powerful communication medium for computer-naive people. Very little research has been done on graphical programming. Glinert and Tanimoto [GLINE84] attempted to construct a procedural, top-down graphical programming language that seemed to hold a great promise [GLINE84]. Unfortunately, they have concluded that the implementation of such a language is impractical given available hardware.

2.8 CONCLUSIONS ON RELATED WORK

The source of difficulties for most current methodologies is the inadequacy of their modeling techniques. Generally, methodologies in the pre-implementation phases use graphical specification techniques and thus produce incomplete models of a system's operation. Part of the operation is represented externally but the other part must be stored in human memory. For example, many of these techniques assume either a control flow, or a data flow model. The user of a control flow model manipulates the data flow in his mind. Likewise, the user of a data flow model manipulates the control flow in his mind. Consequently, the final sys-

tem is created by manual compilation of these parts. A formal technique which unified these parts (i.e., what is represented, and what is left in human memory) would reduce the load on human memory, would ease inter-role communication, and would allow for automatic compilation of the software parts of a human-computer system representation.

In SUPERMAN, a graphical representation includes system functions, data flow, control flow, information structures, and concurrent operations, as well as identifying who performs each of the functions. Hence, SUPERMAN integrates many of the fundamental approaches and functions of other methodologies. Also, SUPERMAN's modeling technique integrates many of the software engineering concepts (e.g., non-functional requirements, representation of a system's behavior, human-computer dialogue, design concepts, structured programming concepts, etc.) and provides for the automatic compilation of a system representation to executable code which realizes the semantics of the notation.

Chapter III

FUNDAMENTALS OF THE SUPERVISORY METHODOLOGY AND NOTATION

In this and the following section, the discussion of SUPERMAN will be restricted to a subset of human-computer systems, namely interactive software systems. The more general form of human-computer systems (i.e., management procedures with computerized tools) will be discussed in Chapter 5.

SUPERMAN integrates into a single representation the functions of data flow diagrams [WEINV80], structure charts of the Structured Design methodology [YOURE79], operational sequence diagrams [MEISD71], Petri nets [PETEJ77, NELSR83], PDLs [CAINS75], conventional flowcharts, and textual programming language representation of function logic. Here, a software function and a software module are equivalent in meaning and accomplish a task described by a single phrase or a sentence (e.g., perform airline reservation). The discussion begins with the fundamental concepts of SUPERMAN: functional structure, control flow, and data flow.

3.1 SUPERVISED FLOW DIAGRAMS

Figure 3.1 illustrates the abstract form of the functional structure of a system developed by SUPERMAN. Each trapezoid in Figure 3.1 is called a *supervisory cell*, and the whole hierarchy of supervisory cells is called the *supervisory structure*. A more detailed form of a supervisory cell is illustrated in Figure 3.2. It consists of a supervisory function and a *supervised flow diagram* (*SFD*). While the supervisory function defines "what" is to be done, the

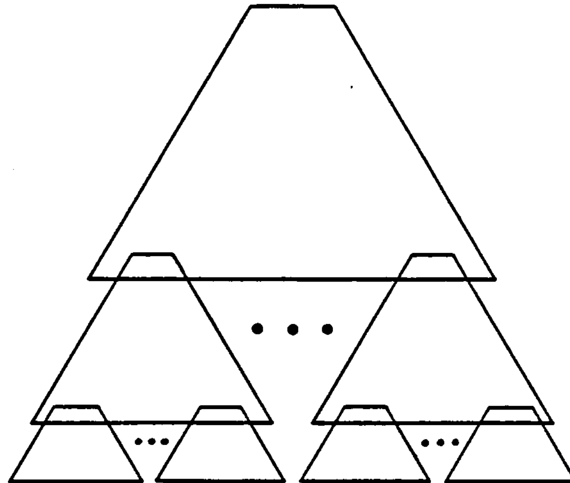

Figure 3.1. General system structure composed of supervisory cells.

SFD defines "how" it will be done. The supervisory function

definition can also state "who" will perform the function. That is, the supervisory function can be tagged with a supervisor (e.g., a human, a computer, etc.) who performs the function. The SFD contains the subfunctions of the supervisory function and the SFD representation specifies the data flow and the control flow through these subfunctions. The key concept is that <u>an SFD's supervisory function administers data flow and control flow through its subfunctions</u> by

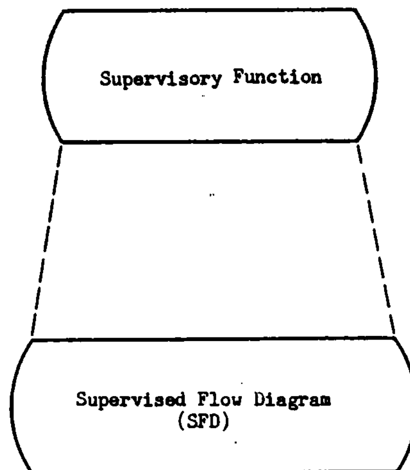

Figure 3.2. Supervisory cell of a supervisory structure.

making decisions and calling the proper subfunctions. Details of this will follow in the next sections. The subfunctions in an SFD can be of two types: another supervisory function or a worker function. If a subfunction is a

supervisory function, it supervises its own SFD at a lower level. A worker function does not supervise other functions, but performs some work (e.g., dialogue or computation) and provides data to, or gets data from, the supervisory functions.

3.1.1 Symbols for Functional Representation of Interactive Software

The graphical function symbols for an interactive software system are shown in Figure 3.3. Their semantics are as follows:

a. Dialogue function - a software function which provides the communication between the user and the computer. A dialogue function, constructed by the HFE, can be a supervisory function (a circle drawn with dashed lines) or a worker function (a circle drawn with solid lines). A supervisory dialogue function can have only dialogue functions in the SFD that it supervises. In the environment of the Dialogue Management System, the dialogue author generates the contents of the dialogue functions called dialogue transactions, using the automated tools provided in the Author's Interactive Dialogue Environment [JOHND82]. Only a dialogue function can communicate with the user.

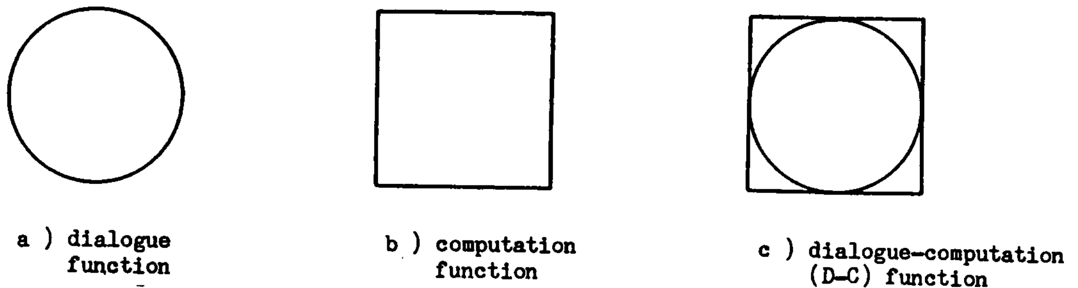

Figure 3.3. Function symbols of an interactive system.

b. Computational function - a software function which will perform only a computation. A computational function, constructed by the SWE, can be a supervisory function (a box drawn with dashed lines) or a worker function (a box drawn with solid lines). A computational supervisory function can have only computational functions in the SFD that it supervises.

c. Dialogue-Computation (D-C) function - a high-level software function composed of both dialogue and computational functions. A D-C function is always a supervisory function, the SFD of which may contain other D-C functions, dialogue functions, and computational functions. D-C functions are used in specifying functional user require-

ments through abstract levels of decomposition. The human-computer interaction sequences are defined within the SFDs of the D-C functions. Therefore, both the HFE and the SWE are involved in constructing a structure of D-C functions.

3.1.2 Supervisory Structure

Figure 3.4 shows part of the hierarchy of supervisory cells for sample supervisory structure. Every system has one top-most supervisory cell, and its supervisory function is the system's overall functional description (e.g., perform airline reservation). In Figure 3.4, this top-most function is labeled A. The SFD for function A is developed to satisfy the requirements for A. In the figure, the SFD for A has a dialogue worker function B, and two supervisory functions C and D. All these functions are supervised by A. The supervisory functions C and D have their own SFDs. Supervisory cells are added to the structure by decomposing the supervisory functions until all the leaves of the tree are worker functions.

An interactive application system, in the design environment of the Dialogue Management System, is decomposed into three components: the dialogue component, the computational component, and the global control component. The di-

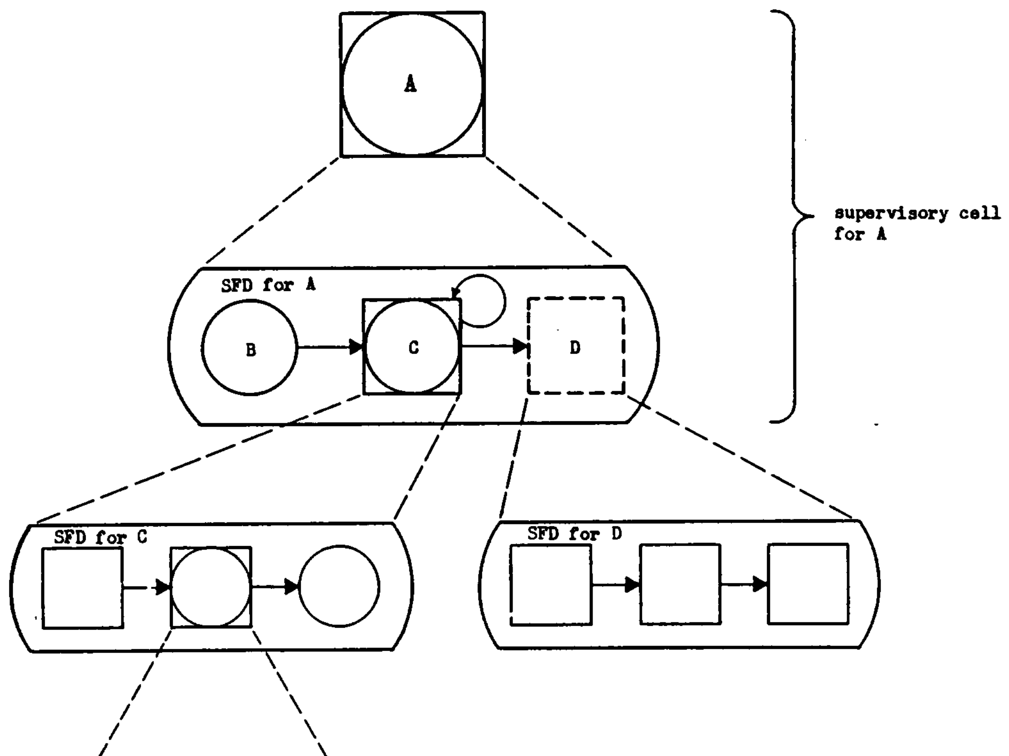

Figure 3.4. The supervisory structure.

alogue component, which is the collection of all dialogue transactions, provides all interaction with the end-user. The computational component, which is the collection of all computational functions, contains the semantic functionality of the target system. The global control component, which is the collection of all D-C functions and the highest level dialogue and computational functions, provides the logical sequencing among the dialogue and computational events. This global control component specifies the functional be-

havior of a system and is called the behavioral structure. The behavioral structure is a high-level specification of the required operation and represents the logical design of a system. The SFDs of the behavioral structure show the dialogue and computational functions as "black boxes". The details of the dialogue and the computational functions are visible at the next lower level of abstraction.

3.2 A GRAPHICAL PROGRAMMING LANGUAGE

Together, the functional symbols of Figure 3.3 represent the semantics of a system design, including data flow and control flow, and are part of the "Notation" of SUPERMAN. As the supervisory structure develops from an early representation of the basic functional system requirements to a complete behavioral structure, the SFD elements are considered to be symbols in a Graphical Programming Language. The methodology is built into the language, and the use of the Graphical Programming Language in developing a system design perforce leads to a natural and orderly application of the methodology.

The Dialogue Management System provides an automated tool as an aid to generating and maintaining SFDs. This tool, in addition to being an editor for the Graphical Programming Language, helps to manage the design by providing

reminders to the system developers about incomplete components and by providing storage and retrieval of those components in a database. The compilable behavioral structure is preserved as a part of the final application system. Then, a dialogue author and an application programmer work in parallel, but rather independently, to implement the dialogue modules and computational modules, respectively. An ordinary textual programming language is often most appropriate for the detailed program code of the worker modules.

The behavioral structure is a graphical, executable representation of the application system control structure. As a result, even before the worker modules are coded, all application system sequences can be demonstrated (See the Behavioral Demonstrator in section 4.4.1).

3.2.1 Control Flow

The control flow in an SFD is indicated by solid lines connecting the functions. The control flow in an SFD consists of three fundamental constructs: sequence, decision, and iteration.

3.2.1.1 Sequence

A sequence of functions in an SFD is illustrated in Figure 3.5. Control flow in an SFD is administered by the

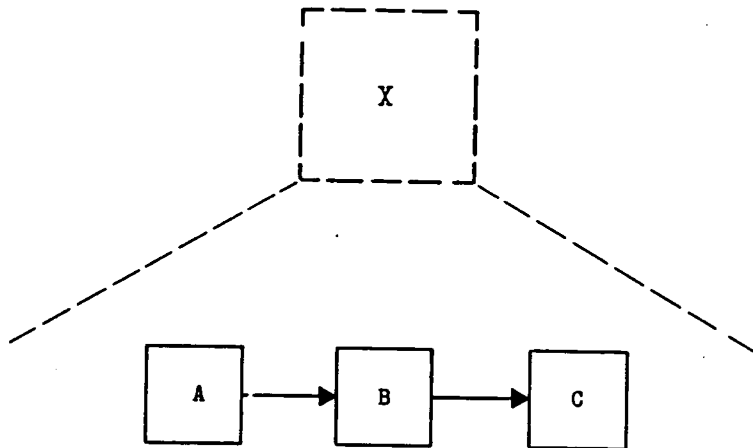

Figure 3.5. A sequence in an SFD.

supervisory function and there is a direct mapping of SFD representation to the code of the supervisory function. The code, in module X, corresponding to the supervisory function of Figure 3.5 is:

```
call A
call B
call C
```

It is important to note that, in the code, A does not call B and B does not call C. Rather each is called by, and returns to, the supervisory function X.

3.2.1.2 Decision

An SFD with a decision construct is illustrated in Figure 3.6. The predicates upon which the run-time path selection is based are enclosed in brackets. In the figure, the triangular symbol indicates the <u>return</u> of control from X to a higher-level supervisory function. The supervisory function, X, first calls function A. Then, depending on the condition, it either returns to its own supervisor, calls B,

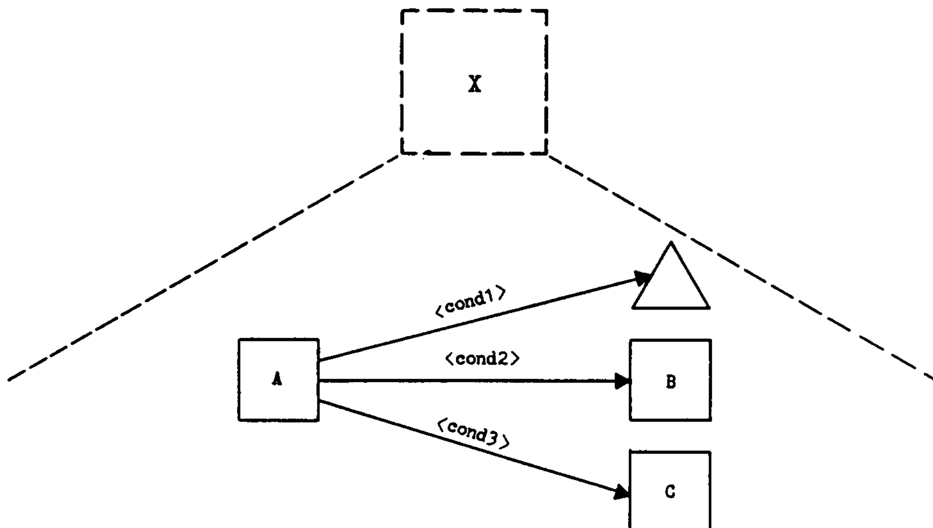

Figure 3.6. An SFD consisting of a decision construct.

or calls C. The code corresponding to X, the supervisory function, is:

```
call A
case cond1: return
     cond2: call B
     cond3: call C
endcase
```

3.2.1.3 Compound Decision

Compound decisions are those which produce nested case statements in the code. A compound decision construct is shown in Figure 3.7. The decision symbol of a conventional flowchart (i.e., the diamond) is used to denote an isolated decision point in an SFD.

The code for the supervisory function of the compound decision construct shown in Figure 3.7 is:

```
case cond1:  call B
     cond2:  case    cond3:  call C
                     cond4:  call D
             endcase
endcase
```

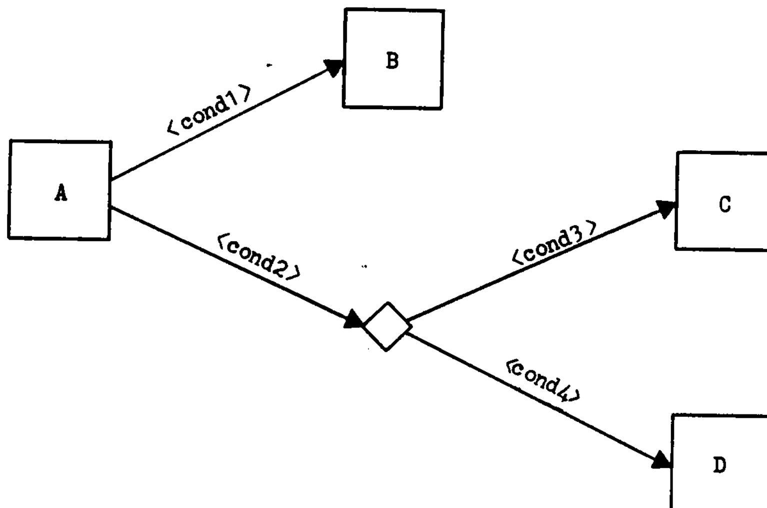

Figure 3.7. A compound decision construct.

3.2.1.4 Iteration

There are two types of iteration constructs: explicit and implicit. Explicit iteration is accomplished through the use of a decision construct and a sequencing construct

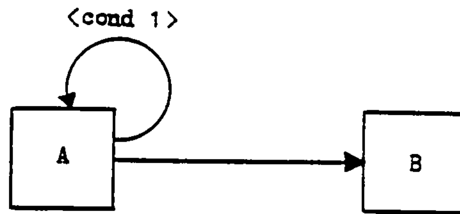

Figure 3.8. An SFD with an explicit iteration construct.

that loops back as in Figure 3.8. The code for the explicit iteration in the supervisory function of Figure 3.8 is:

```
label call A
      case cond1: go to label
      call B
```

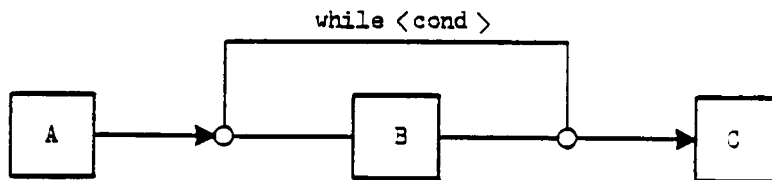

Figure 3.9. An SFD with an implicit iteration construct.

In implicit iteration, the flow of control is implied. Programming language constructs such as DO WHILE, REPEAT

UNTIL, and DO J=K,L are in this class. An SFD with an implicit iteration construct, corresponding to a DO WHILE, is shown in Figure 3.9. The code for the supervisory function is:

```
call A
do while cond
    call B
enddo
call C
```

The direct implementation of the DO WHILE construct in terms

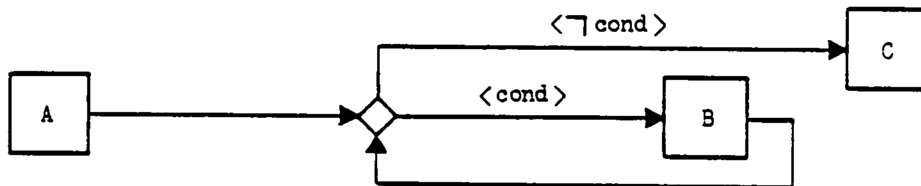

Figure 3.10. An SFD showing an implementation of DO WHILE in terms of other constructs.

of the other constructs is shown in Figure 3.10. The UNTIL <cond> or DO J=K,L constructs can be defined similarly.

3.2.1.5 Recursion

Recursion occurs when a function invokes either itself or a supervisory function which lies on the path from the

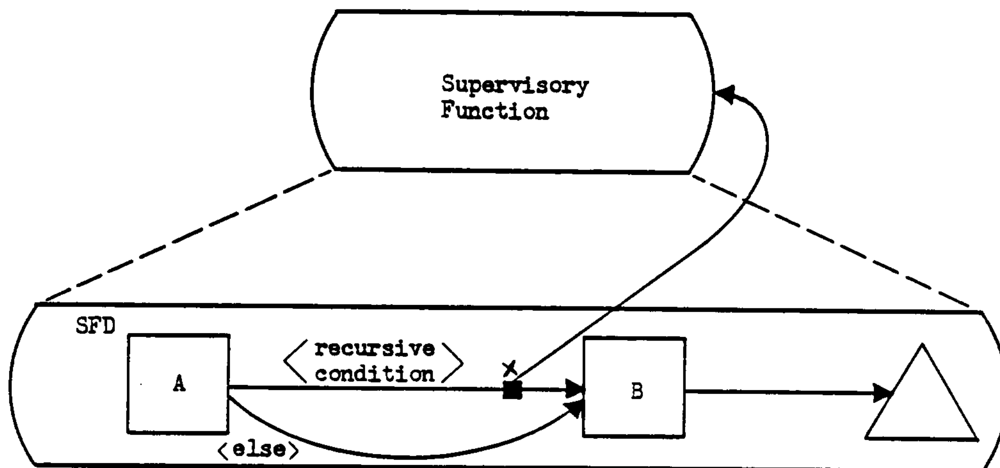

Figure 3.11. Recursion in a supervisory cell.

calling function to the root of the supervisory structure. The small box at point X of Figure 3.11 indicates the point of recursive call. There are two control lines emerging from the small box. While the control line from X crossing the SFD boundary points to the (supervisory) function being called, the other control line from X, within the SFD, leads to the function, B, to be invoked upon return from the recursive function. An instance of the execution of this cell

might be as follows: when the supervisory function is called, it calls function A. After A completes, seeing that the recursive condition is true, the supervisory function marks the control line that goes to function B at point X, the return point for control, and calls itself to repeat the procedure in the SFD. It calls function A again. Assuming that this time the recursive condition is not true, the supervisory function takes the "else" control path and calls function B. After completion of function B, the second invocation of the procedure is completed and it returns to where it was left at point X in the earlier invocation. It then calls function B. After completion of function B, the supervisory function of this SFD returns control to its supervisor.

Code for the supervisory function is:

```
Procedure supervisory_function
  call A
  if <recursive_ condition> then call superviso-
ry_function
  call B
return
end
```

3.2.2 Data Flow

There are two types of data flow: data flow between a function and its supervisor, and data flow between a function and a storage area.

3.2.2.1 Data Flow Between a Function and Its Supervisor

Data flow between a function and its supervisor is performed by parameter communication. All functions in an SFD send their output parameter values to the supervisory function and get their input parameter values from the supervisory function. No parameter value communication occurs directly between functions in the same SFD.

Data flow is shown either on control lines or by dashed lines which are separate from the control lines. In Figure 3.12, an SFD illustrates data flow on the control lines, which are used to indicate that the functions have to be

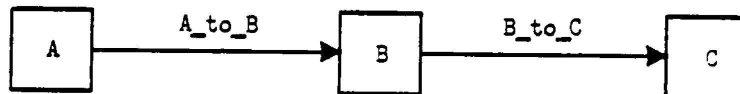

Figure 3.12. Data flow through a sequence of functions.

performed in sequence. The output data of function A is the

variable "A_to_B", which is the input data to function B. The output data of function B is the variable "B_to_C", which is the input data to function C. The corresponding code in the supervisory function is:

```
call A (A_to_B)
call B (A_to_B, B_to_C)
call C (B_to_C)
```

Figure 3.13 illustrates the equivalent form of Figure

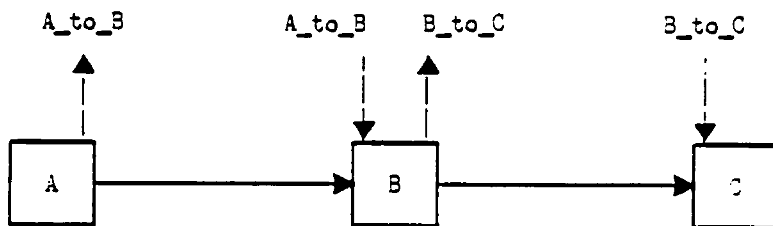

Figure 3.13. Equivalent form of Figure 3.12.

3.12. In this figure, data flow is not shown on control lines. As an alternative, upward arrows show the data received by the supervisory function, and the downward arrows show the data sent by the supervisory function. The solid lines show the control sequence to be executed by the supervisory function.

Figure 3.14 shows a case in which data flow does not occur parallel to control flow. Here, function A has two

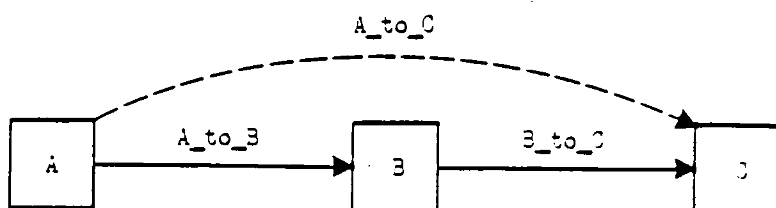

Figure 3.14. A mixed data-control flow configuration.

output values. Although the value of the variable "A_to_B" is used by the immediate successor function B, the value of "A_to_C" is used only by function C. The code corresponding to the supervisory function of Figure 3.14 is:

```
call A (A_to_B, A_to_C)
call B (A_to_B, B_to_C)
call C (A_to_C, B_to_C)
```

Some functions in an SFD (especially the first and the last function of the control flow) might receive data originating from the supervisory function (e.g., data traveling downward through the supervisory structure), but not from another function in the same SFD. Likewise, they might send data to the supervisory function (e.g., data traveling upward through the supervisory structure), but not to another function in the same SFD.

3.2.2.2 Data Flow Between a Function and a Storage Area

Some examples of storage areas are common blocks in FORTRAN, files on secondary storage, encapsulated data

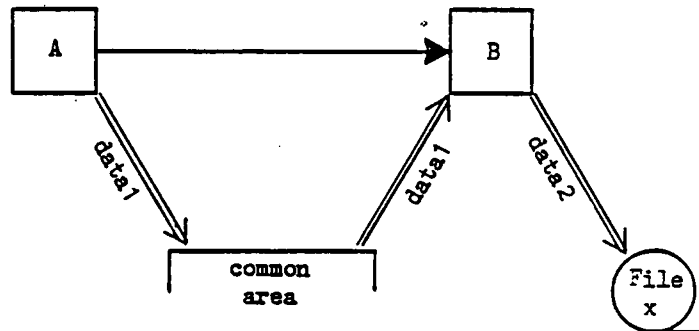

Figure 3.15. An SFD with data flow between functions and storage areas.

structures, and interprocess communication buffers. In Figure 3.15, an SFD with such data flow is shown and, instead of communicating through the supervisor, functions A and B communicate through a common area, and function B puts its output into File X.* This type of data flow is not supervised by the supervisory function of the SFD. To distin-

---

* It is strongly recommended that the reader refer to the works of Yourdon [YOURE79], and Myers [MYERJ78] for a discussion of the problems caused by common coupling of functions (e.g., coupling through a common block of data) in software development. The use of common areas is allowed in the methodology to give the user the freedom to make his or her own trade-offs.

guish this kind of data flow from data flow between a function and a supervisory function, arrows with double lines are used. This type of data flow is to be performed under control of the associated functions (e.g., functions A and B in Figure 3.15). It is shown on the SFD to help system developers show input and output data of functions (other than parameters) and specification of storage areas.

3.2.3 Internal Work of a Function (Internal Code Block)

So far, a supervisory function has been used solely to supervise other functions by performing control and data flow management. An SFD can also indicate computational code internal to the supervisory function, defined in terms of internal code blocks. Figure 3.16 shows a supervisory

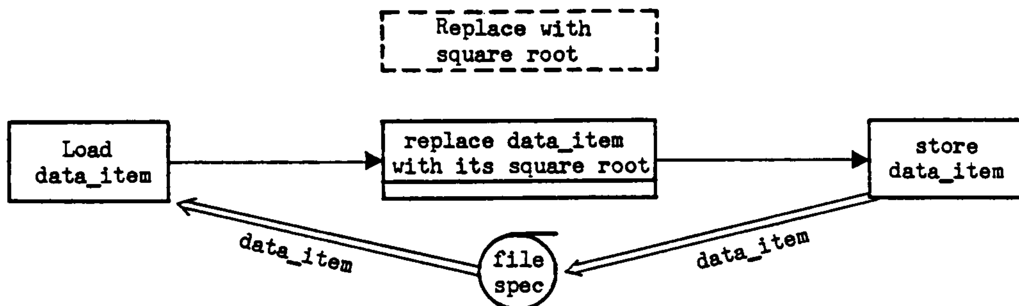

Figure 3.16. A supervisory cell with an internal code block.

cell with an internal code block. An internal code block is

represented by a rectangular box with an added bar, and its work is specified in the box (e.g., in natural language, or in a programming language, or in some symbolic form). When the diagram is compiled, the internal code blocks produce in-line code rather than, for example, procedure calls.

The code corresponding to the supervisory function of Figure 3.16 is:

```
Subroutine Replace_with_sqrt
  real data_item
  call load (data_item)
  data_item = SQRT (data_item)
  call store (data_item)
return
end
```

In the supervised flow diagrams, internal code blocks are used for small amounts of code which do not merit being separate software modules.

3.2.4 Data to be Declared for a Function

All data that appear in the supervisory cell representation must be declared (e.g., in internal code blocks, in data structure diagrams) for its supervisory function, including inputs to the supervisory function, outputs of the supervisory function, and all data that are locally used in

the SFD. Such data can be declared in an internal code block, or in the form of a data structure diagram.

3.2.5 Supervision of Concurrent Functions

A supervisory function can initiate, communicate with, control, synchronize, and terminate the operations of several functions concurrently. As a simple example of precedence control, consider the precedence graph embedded in the SFD of Figure 3.17. The function symbols with the double lines indicate the concurrent functions under the control of the SFD supervisor. Code corresponding to the supervisory function S is:

```
Procedure S
  call A
  spawn B,C
  wait B
  spawn D,E
  wait D,E,C
  call F
return
end
```

A run-time representation of the textual code given for Figure 3.17 is shown in Figure 3.18. In the figure, the semicircles indicate inter-function communication buffers and

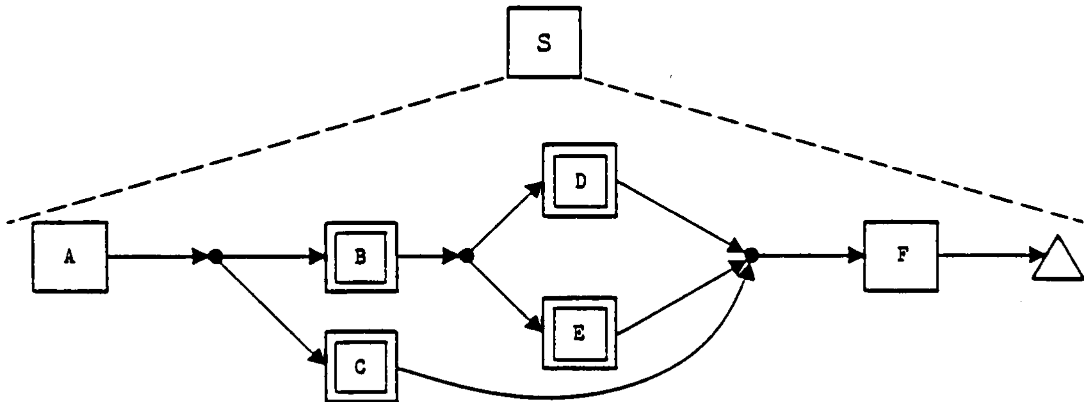

Figure 3.17. A supervisory cell with a precedence graph.

are labeled with the associated function names. The spawned functions B, C, D, and E must, of course, be programmed to perform "signal" operations at their completion. The translation of Figure 3.17 creates the inter-function communication buffers and wait operations shown in Figure 3.18. While Figure 3.18 is an example of signal communication, neither is inter-function communication restricted to signal communication, nor is a supervisory function's concurrent operation restricted to precedence realization. Concurrent functions B, C, D, and E of the above figures have their own supervisory structures. Hence, a system of concurrent functions consists of communicating supervisory structures. For

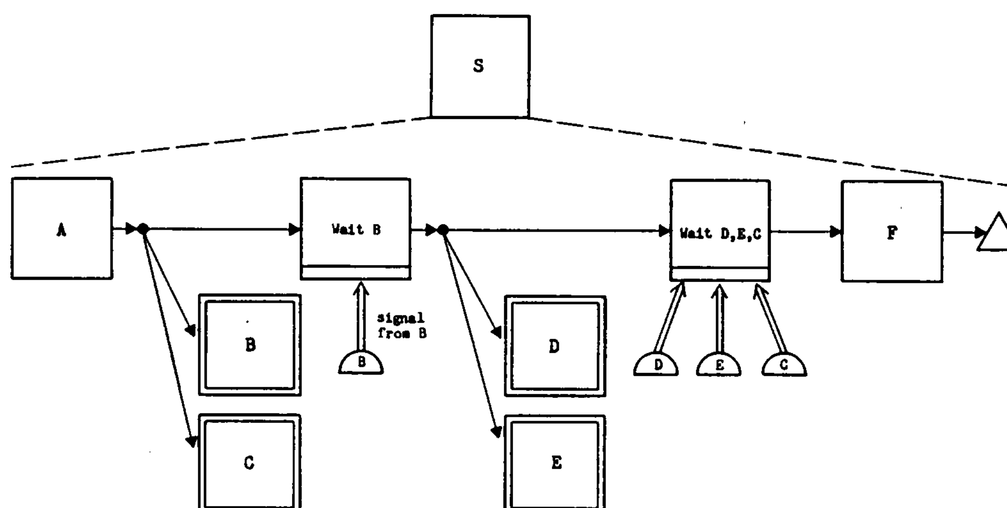

Figure 3.18. Run-time representation of Figure 3.17.

example, in an airline reservation system, several interactive supervisory structures (e.g., one for each airline agent) may communicate with a computational supervisory structure which monitors the access to shared data (e.g., to provide for mutual exclusion).

3.3 AN EXAMPLE

Following is a simple interactive example which includes some of the constructs discussed above. The supervisory cells of this example (Figures 4.19 and 4.20) have been constructed using the Graphical Programming Language editor, and the associated code (Figure 3.21) has been generated by the Graphical Programming Language compiler. It happens that, for ease of implementation, the initial implementation of the Graphical Programming Language compiler produced FORTRAN code as an intermediate high-level language, rather than, say, direct machine language code. However, the concepts of SUPERMAN and the Graphical Programming Language are completely language independent. A new version of the GPL compiler is presently being designed, within the DMS project, to produce code in the "C" language.

The problem in this example is to develop an interactive tool from which the user requests the computation of a trigonometric function (e.g., sine), given an angle value (e.g., 2.3 radians). The top-most supervisory cell is shown in Figure 3.19. The supervisory function is named "determine_calculation_value". It is represented by a D-C symbol. The semantics of the symbol are that, as observed in the supervised flow diagram, the dialogue and the computational functions perform this function together. The first func-

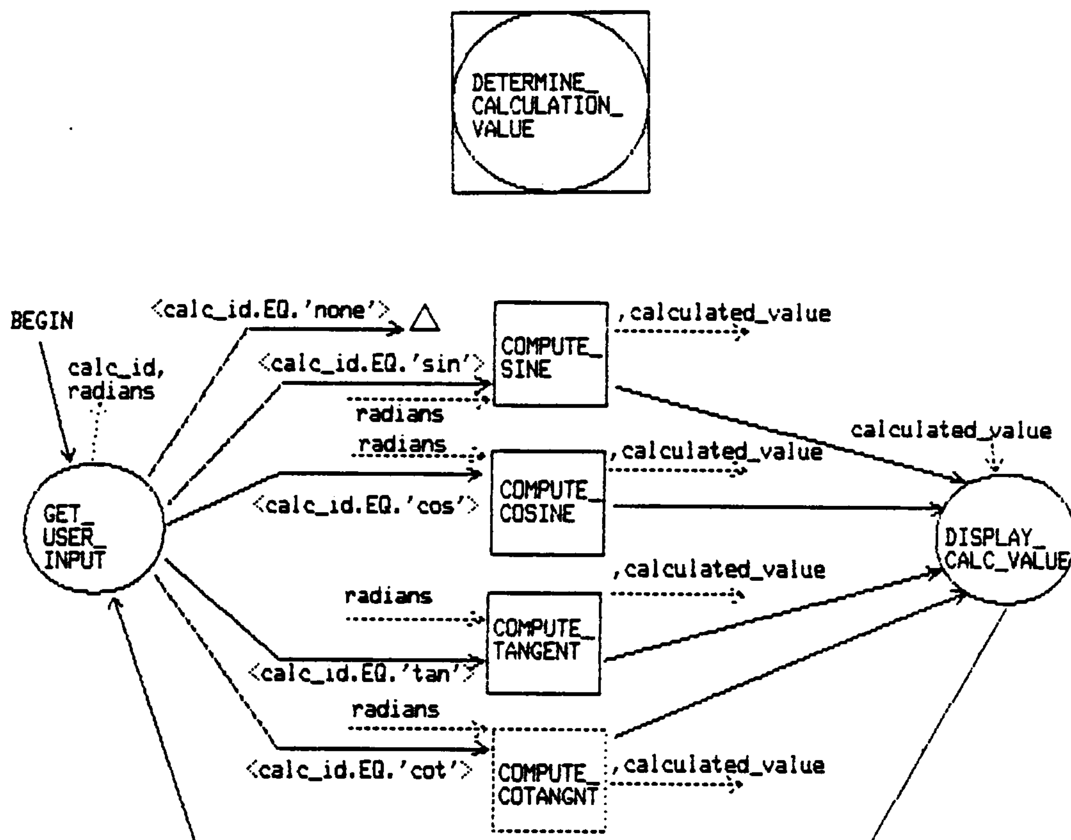

Figure 3.19. Supervisory cell for "deter-
mine_calculation_value".

tion that the supervisor invokes is a dialogue function "get_user_input". This function gets from the user the value of the variable "calc_id" and the radian value, and gives these to the supervisory function (i.e., they are out-

put parameters from the dialogue function). The supervisory function uses a decision construct and invokes one of the computational functions ("compute_sine", "compute_cosine", "compute_tangent", or "compute_cotangent"), according to the value of the "calc_id", and passes it the radian value. The invoked computational function returns the "calculated_value". Then, the supervisory function invokes the dialogue function "display_calc_value", which is next in the sequence, and passes it the "calculated_value". Next, the supervisory function follows the control line and invokes "get_user_input" again (i.e., iteration). Iteration continues until the supervisory function gets from the dialogue function the value "none" for the character string variable "calc_id". When this happens, as indicated by the triangle in Figure 3.19, the supervisory function has completed its task. The control of execution returns to the supervisory function at the next higher-level above "determine_calculation_value".

Note that, in Figure 3.19, the functions "compute_tangent" and "compute_cotangent" are shown with dashed lines. This means that these are both supervisory computational functions (while the rest are worker functions) and each one has a corresponding supervisory cell. The supervisory cell for the function "compute_tangent" is shown in

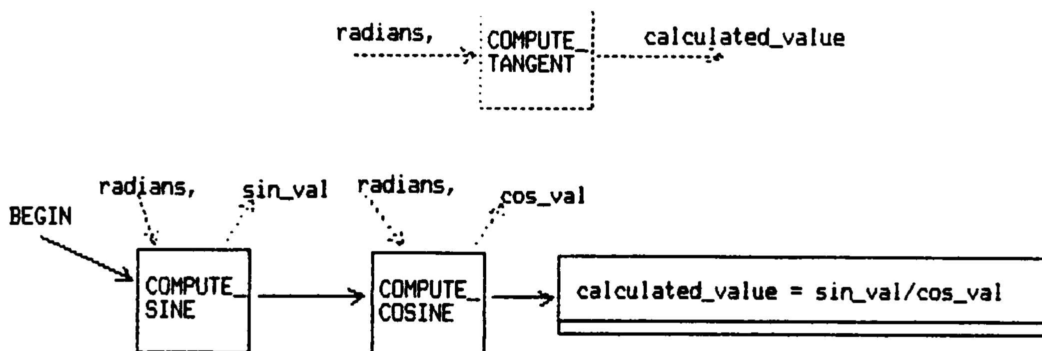

Figure 3.20. Supervisory cell for "compute_tangent".

Figure 3.20. While the rectangular boxes (i.e., the computational functions) will be implemented as separately compiled modules, the rectangular box with a double bar represents an internal code block and is in-line code appearing in the supervisory function. The code for these two supervisory cells, as generated by the Graphical Programming Language compiler, is shown in Figure 3.21.

The example presented above shows that a given function can appear in more than one supervisory cell of the supervisory structure. This is a common situation, where a function is shared, as the "compute_sine" is shared in the example above. These functions require special attention, especially during system maintenance. If a function is used

```
1          Program DETERMINE_CALCULATION_VALUE
2     03     call GET_USER_INPUT(calc_id,radians)
3          if (calc_id.EQ.'none') then
4            goto 999
5          endif
6          if (calc_id.EQ.'sin') then
7            call COMPUTE_SINE(radians,calculated_value)
8     08     call DISPLAY_CALC_VALUE(calculated_value)
9            goto 03
10         endif
11         if (calc_id.EQ.'cos') then
12           call COMPUTE_COSINE(radians,calculated_value)
13           goto 08
14         endif
15         if (calc_id.EQ.'tan') then
16           call COMPUTE_TANGENT(radians,calculated_value)
17           goto 08
18         endif
19         if (calc_id.EQ.'cot') then
20           call COMPUTE_COTANGNT(radians,calculated_value)
21           goto 08
22         endif
23   999   stop
24         end
25
26
27         Subroutine COMPUTE_TANGENT(radians,calculated_value)
28           call COMPUTE_SINE(radians,sin_val)
29           call COMPUTE_COSINE(radians,cos_val)
30           calculated_value = sin_val/cos_val
31         return
32         end
```

Figure 3.21. FORTRAN code generated by the Graphical Programming Language compiler.

by multiple higher-level functions, any change made in this function might destroy the existing relationships with some of the higher-level functions. Therefore it is necessary to keep track of the calling relationships among functions. The "structure chart", shown in Figure 3.22, for the example given above serves this function for the Structured Design

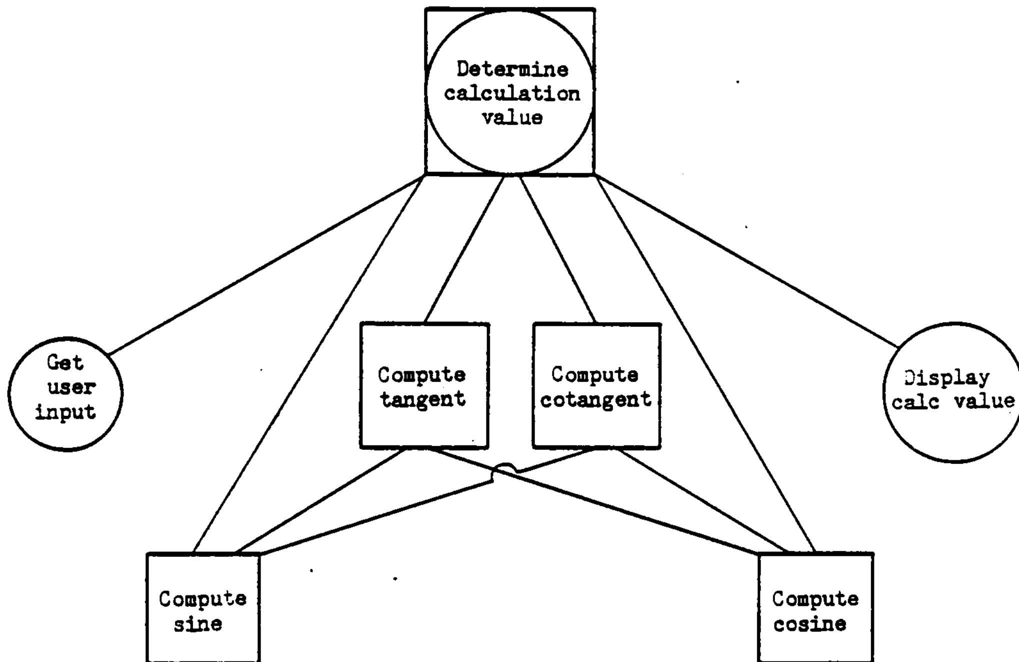

Figure 3.22. Structure chart for the example.

methodology. Information about calling relationships can be retrieved directly in the Dialogue Management System design environment; the GPL editor automatically constructs the internal representation for a structure chart (see next section) and displays this information upon the designer's request.

3.4 COMPARISON WITH OTHER REPRESENTATION TECHNIQUES

The concept of a supervisory structure bears some relationship to structure charts of the Structured Design methodology [MYERJ78, YOURE79, WEINV80, STEVW81], data flow diagrams [WEINV80], and detailed design techniques such as conventional flowcharts and PDLs [CAINS75].

In the Structured Design methodology, a structure chart outlines the calling relationships of the functions, but hides the procedural relationships. However, when a structure chart is used alone, the procedural relationships are manipulated in the minds of the developers. Consequently, a structure chart by itself leads to such difficulties as communication of ideas, analyzing problems with complex algorithms, and modeling human-computer interaction. Usually, data flow diagrams are used as a front-end analysis tool and are converted to a structure chart in the architectural design phase.

While using SUPERMAN, the methodology user has the option of specifying the SFDs at any level of procedural detail (e.g., ignoring iteration, using abstract names for data objects, etc.). A fully-detailed SFD is required only for the compiler. In early phases of the lifecycle, while the system developer can do functional abstraction (i.e., functional decomposition through levels of the hierarchy),

the developer can also partially or fully specify the procedural relationships at a given level in the hierarchy. Hence, the detailed specification process is responsive to communication needs of the developers. When procedural details are omitted, an SFD looks like a data flow diagram. However, the supervisory structure of SFDs is not converted into a structure chart, but is the working representation which, through refinements, evolves into executable code.

Detailed activities for the design of the supervisory structure involve fully specifying the procedural components (e.g., data and control flow) on the SFDs. After the detailed design, an SFD looks somewhat like a conventional flowchart. However, a conventional flowchart shows only control flow, and ignores data communication at a given level and through the levels of a functional structure. The supervisory cell concept of SUPERMAN enables specification of data communication as well as specification of control flow.

PDLs provide an alternative detailed design representation. However, as with all textual languages, the control structures of a PDL are mixed with non-control statements and are not as explicitly recognizable as they are in graphical representations. In addition, the textual representations do not have the notational power of graphics. These are some of the reasons why graphical representations are

preferable for system analysis, with the PDL reserved for detailed program design.

3.5 THE RELATIONSHIP OF THE SFD TO OPERATIONAL SEQUENCE DIAGRAMS

A system dialogue function and the corresponding user action are dual versions of the same thing. If a system dialogue function gives some information, the user gets it. If a dialogue function gets some information, the user gives it. Hence, on an SFD, the dialogue functions, which are software functions, can be replaced by the corresponding user functions (shown as hexagons). When this is done, the SFD, which normally shows only computer functions, shows the relative roles of the user and the computer. This form of Figure 3.19 is shown in Figure 3.23, where circular dialogue function symbols are replaced by hexagonal user function (human-action) symbols (see Chapter 5). The relationship between the two forms is one of duality. In Figure 3.23 one looks in at the computer from the user's point of view, whereas in Figure 3.19 one looks out at the user from inside. The reader who is familiar with operational sequence diagrams [MEISD71] will see that Figure 3.23 is similar to an operational sequence diagram. An operational sequence diagram is an industrial engineering graphical tool, used by the HFE, in modeling human-machine interaction. An opera-

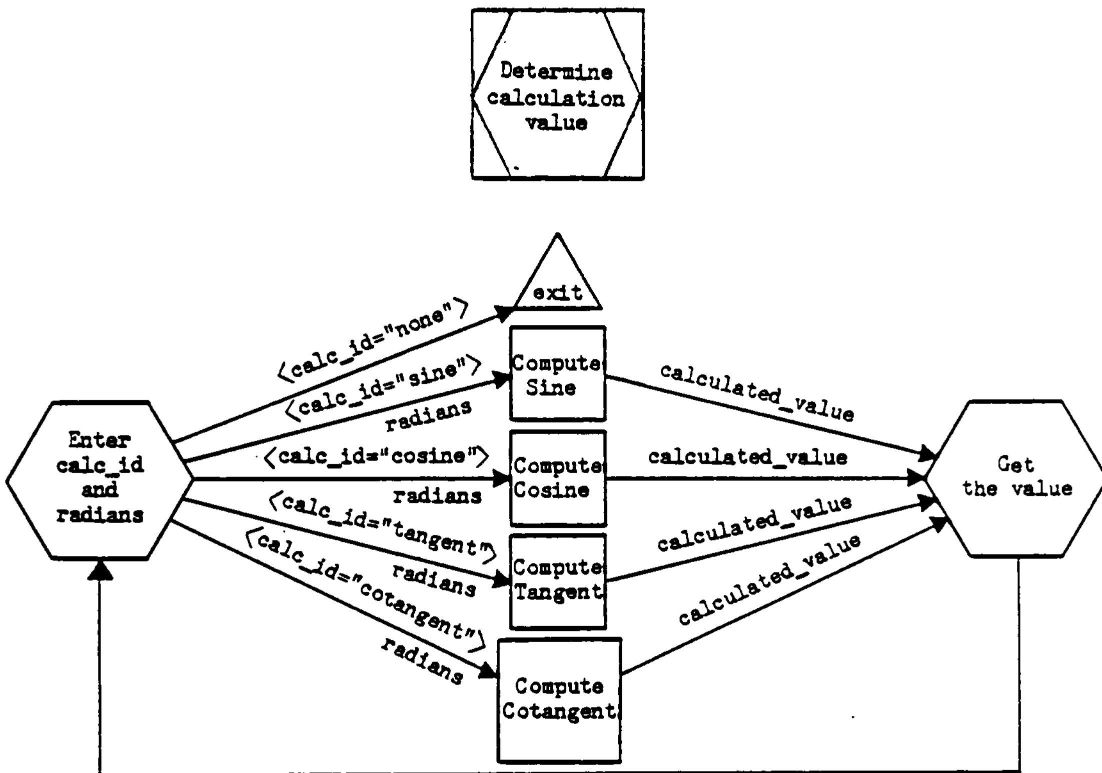

Figure 3.23. Operational sequence diagram form of Figure 3.19.

tional sequence diagram in the context of Figure 3.23 shows the control paths that a user has to follow and shows the data that a user has to provide along these control paths. SUPERMAN structures these functions into levels of abstraction, just as it does for a system composed only of software. If the developers wish to use this outside-in view,

the final structure can still be converted to the interactive software representation simply by replacing user functions (hexagons) with software dialogue functions (circles) and rewording the dialogue functions (e.g., exchanging "give" and "get"). Similarly, an inside-out view can be converted to an outside-in view to provide the user with a structured operation diagram.

Chapter IV

INTERACTIVE SOFTWARE SYSTEM DEVELOPMENT

In this section, the use of SUPERMAN in a developmental approach to human-computer systems will be discussed. Again, the scope is limited to the interactive software part of the system. The holistic approach which models both human and computer functions will be discussed in Chapter 5.

SUPERMAN supports the concept of the software lifecycle, which is shown as a SUPERMAN high-level management procedure in Figure 4.1. The "managers" in this diagram are the system developers.

4.1 REQUIREMENTS SPECIFICATION

The goal of the requirements specification activity is to define the target system's operation. Mistakes and omissions made in this phase of the software lifecycle can result in massive changes and high cost. Therefore the specifications must be complete, correct, and unambiguous. In general, the roles involved in requirements specification are system user (i.e., the customer), application expert (AE), the SWE, and the HFE.

With SUPERMAN, the top part of the supervisory structure is developed in the requirements specification phase,

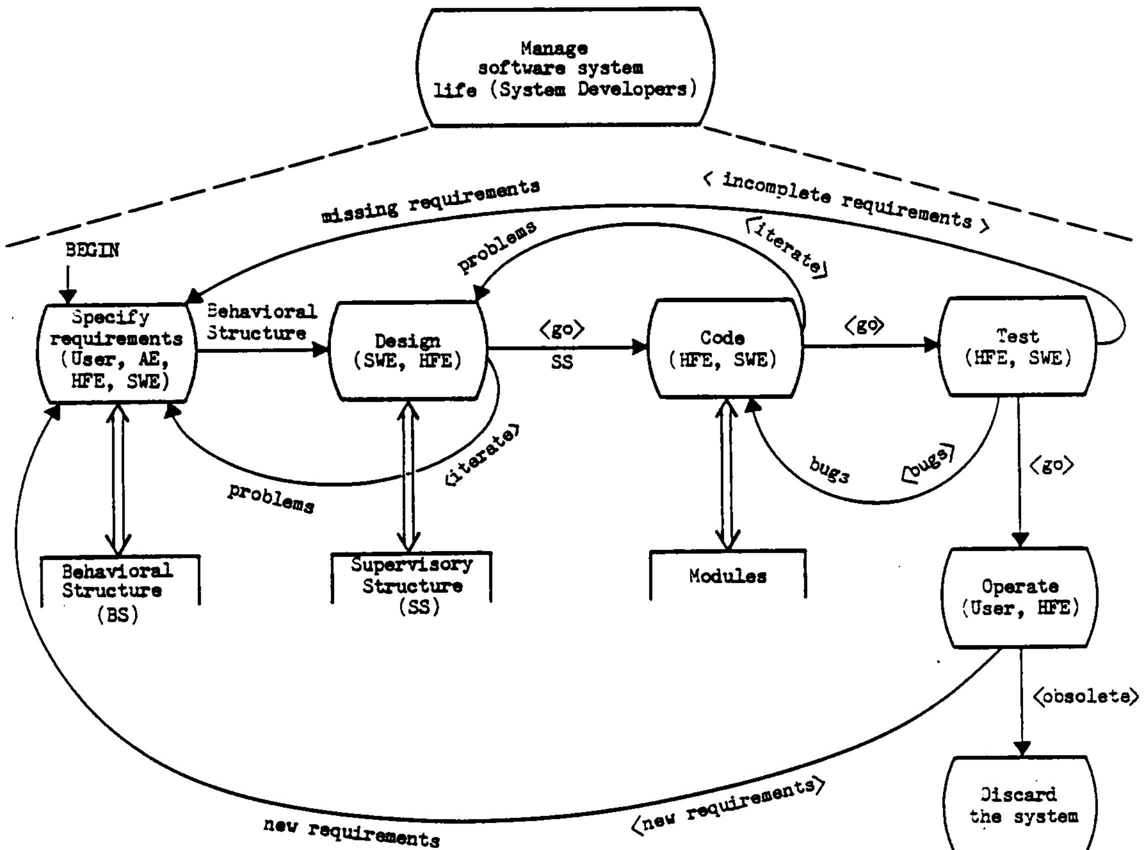

Figure 4.1. A management procedure for software lifecycle.

and the bottom part is developed in the design and implementation phases. The whole structure is used throughout software life.

Figure 4.2 illustrates the overall supervisory structure representation of a system. The portion of the struc-

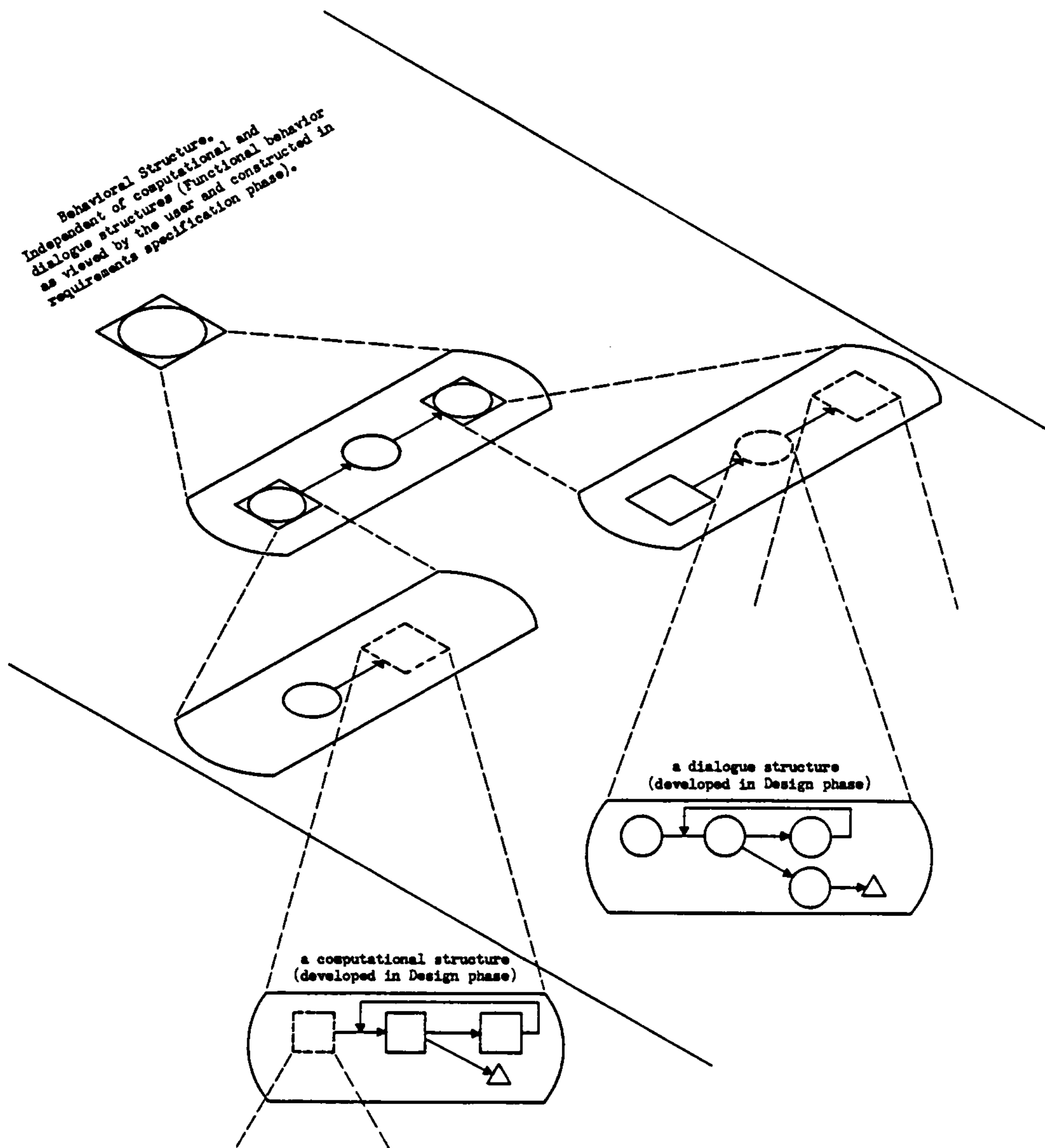

Figure 4.2. Overall supervisory structure representation of a system.

ture on the horizontal plane, constructed in the require-

ments specification phase, represents the behavioral model of the system and is called the behavioral structure. The structures on the vertical planes are the dialogue structure and the computational structure and are developed in the design phase. A general taxonomy for organizing system re-

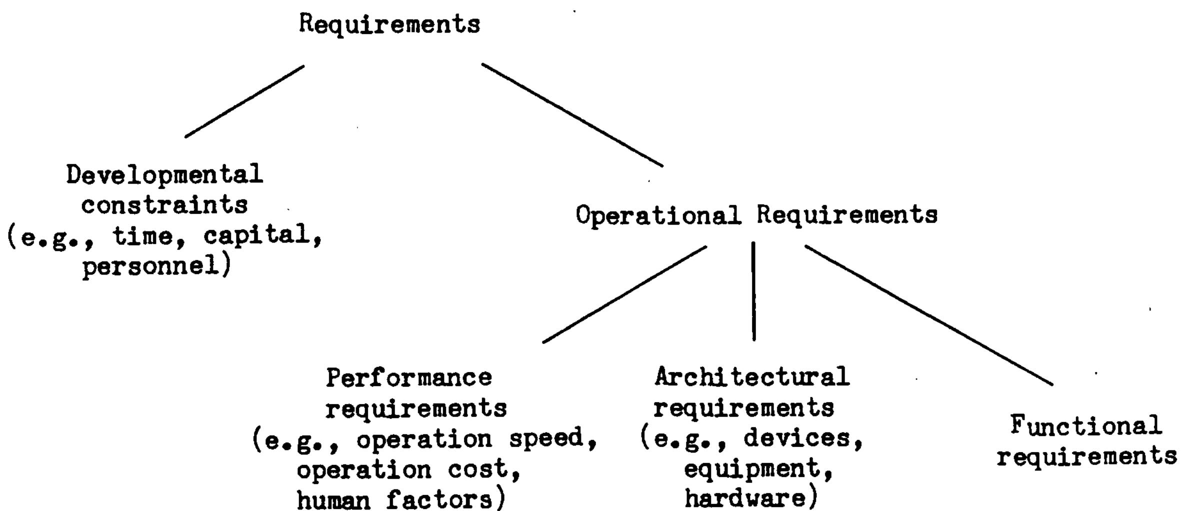

Figure 4.3. A taxonomy for organizing requirements specification.

quirements is illustrated in Figure 4.3. The fundamental activity in the requirements specification phase is analyz-

ing the feasibility of the development process in balancing the constraints imposed by available resources against the target system's operational requirements. The SFD structure of a target system represents the functional portion of its requirements specification. The non-procedural requirements (developmental constraints, performance requirements, and architectural requirements) are documented in lists attached to the appropriate supervisory cells. The GPL support environment for the methodology allows access to this documentation via the corresponding SFD symbols. Requirements specified for a function apply to the whole structure underneath that function. Therefore, a lower-level function might not have any requirements specified in its list. If it does, these requirements are either specific to the lower-level function, or refinements of the requirements specified for the higher-level supervisory functions which use this lower-level function. Just as an SFD for a supervisory function at any level applies to its expansion at all levels below, so its requirements list applies to all nodes lower in the hierarchy. The result is that the overall requirements specifications (procedural requirements in the SFDs and non-procedural requirements lists) are distributed throughout the system representation's hierarchy of abstraction.

Construction of the behavioral structure is complete when there are no more D-C functions to expand. For example, the supervisory cell shown in Figure 3.19 is the behavioral structure for that simple human-computer system. If purely dialogue or computational functions (either supervisory or worker) are encountered during construction of the behavioral structure, they are included, but they are not expanded into their SFDs. Hence, the behavioral structure is a supervisory cell tree bounded with the highest-level dialogue and computational functions as leaves. All other nodes of this tree are D-C functions. The behavioral structure is independent of the dialogue and the computational structures in the sense that the dialogue and computational structures can be developed and maintained without affecting the behavioral structure.

The behavioral structure is an unambiguous medium for functional requirements specification and, by hiding the technical details of design, helps communication among people of differing technical computer knowledge. If the behavioral structure is a correct model of the system requirements, then by definition the system meets its functional requirements at the logical level. This is because the behavioral structure is the target system control structure that executes on the computer, supervising the dialogue

structure and the computational structure. Because the behavioral structure is executable, it allows designers and potential users, with the help of the Behavioral Demonstrator (see Section 4.4.1), to observe the behavior of the target system at a very early stage, before any design and implementation efforts. In this way, the methodology strongly supports rapid prototyping.

4.2 DESIGN

The transition from requirements specification to the design phase of the lifecycle begins when the behavioral structure is completed and the dialogue transactions and purely computational structures are first expanded. It is a feature of SUPERMAN that this transition is very smooth, each phase of the lifecycle leading easily to the next. The transitions are conceptual activities which do not move from one representation technique to another. All phases use the same mechanism (SFDs) for representing the system, each successive phase seeing it developed to further levels of abstraction. In practice, the designers find that each next phase is already well underway due to "fall out" from previous phases. Thus, the lifecycle phases are useful for focusing attention on the proper level of abstraction, without partitioning development activities.

The design phase fixes the physical architecture, many issues of which may have been informally decided during the requirements specification phase. The computational and dialogue structures (Section 4.1) are appended to the behavioral structure in a manner that is consistent with physical architecture decisions. For instance, in information storage and retrieval, databases can be implemented as files. In this case the computational functions have to perform the algorithms for file search, retrieval, and update activities. As an alternative, all data can be stored in a database which is administered by a database management system. In this case, instead of invoking worker functions to manipulate files, the computational functions send queries to the database management system.

4.3 CODING

A software supervisory structure is the coded definition of a computer's operation. If system developers have a compiler for the Graphical Programming Language, this graphical representation of operation is automatically translated into executable code. Otherwise that code is produced by manual translation of the supervisory structure. Conventional textual coding is used for producing the details of dialogue-computation, dialogue, and computational functions.

4.4 TESTING

In conventional application of the lifecycle, testing is a separate phase following design and implementation. In SUPERMAN, tools are provided to initiate the testing process during any phase of the lifecycle. The key to this "test as you go" approach is provided in the Dialogue Management System by a tool called the Behavioral Demonstrator.

4.4.1 The Behavioral Demonstrator

Some systems claiming to have dialogue tools address only the form and content of dialogue. However, even high-quality form and content cannot compensate for poor logical sequencing. And since the sequencing of dialogue is tied to the logical structure of the application software design, it is essential that the logical flow be tested as early as possible in the design process, to refine the specification of system requirements, and to identify and make changes to improve the resultant user interface. This need to visualize dialogue sequences continues all the way through the development process until the application system is implemented and operational, as well as later, during modification cycles. To meet this need, the Behavioral Demonstrator allows the execution of the behavioral structure, to be observed by the application end-users and studied by human-

factors experts to verify the fulfillment of their system requirements and to evaluate whether their requirements were complete and correct, all before any conventional program source code is written. Changes in SFDs are easier to make than changes in partly coded systems. The Behavioral Demonstrator executes the behavioral structure, using the Graphical Programming Language internal representation. An interface is also provided to the user or human-factors experimenter to capture design notes and suggestions as the application system is exercised.

4.5 HUMAN-COMPUTER SYSTEM MAINTENANCE

The operation of a human-computer system demonstrates whether the system development efforts have been successful. Although the testing of software is done before the human-computer system operation, some deficiencies in human-factors and most performance problems can most effectively be discovered when the software is used by its intended users. In an experimental environment the HFE keeps data during system operation (e.g., user response time). Dialogue transactions contain mechanisms for recording data on user behavior.

Deficiencies or changes in requirements result in maintenance. Maintenance can be classified as either local

or global changes. Local maintenance involves changes which are only in the dialogue or computational structures. For example, the dialogue author can alter the contents of a dialogue transaction by changing the format of a display. These activities of the dialogue author do not concern the programmer at all, as long as the changes made by the author are internal to the dialogue. Similarly, the programmer can independently change the contents of the computational modules, for instance, to optimize an algorithm.

Global changes are changes which involve the behavioral structure. During such changes, the two specialists might need to work in cooperation. Examples include the addition of new requirements which need both dialogue and computational parts and a change in the control sequence which might affect the sequencing of dialogue and/or computational functions. To modify the system, both parties work on the current system representation, the overall supervisory structure. The supervisory structure serves as a single unified document for communication of the design among roles. The Graphical Programming Language representation of the SFDs is traversed as the Behavioral Demonstrator exercises the design, in order to locate the part of the system needing modification.

Chapter V

HOLISTIC APPROACH TO HUMAN-COMPUTER SYSTEM DEVELOPMENT

There are at least two different ways of viewing a human-computer system during its development:

1. As an integrated set of computer tools (e.g., a computer-based software system), and
2. As a human-computer system which contains, as a part of it, such a set of tools (e.g., operational procedures to achieve system goals with embedded computer tools).

Usually, in top-down software development, the developmental process starts with specifying the requirements to be met by the software, which is what finally executes on the computer. Hence, the first viewpoint above is most often taken by the software developer at the top-most level of the developmental process. However, the total system that finally operates is what is seen in the second view, and this is what interests the HFE, the systems analyst, and the system customer. Especially in human-computer systems which require highly effective or efficient performance (e.g., air-traffic control systems, collision-avoidance systems), the human's role as part of the system becomes an important analysis issue.

Figure 5.1 represents the operation of a human-computer system, which is enclosed by outer dashed-line. The system consists of humans and computers, and an environment of accessory components such as a radar, machines, etc. Human and computer activities which do not serve the goal of the

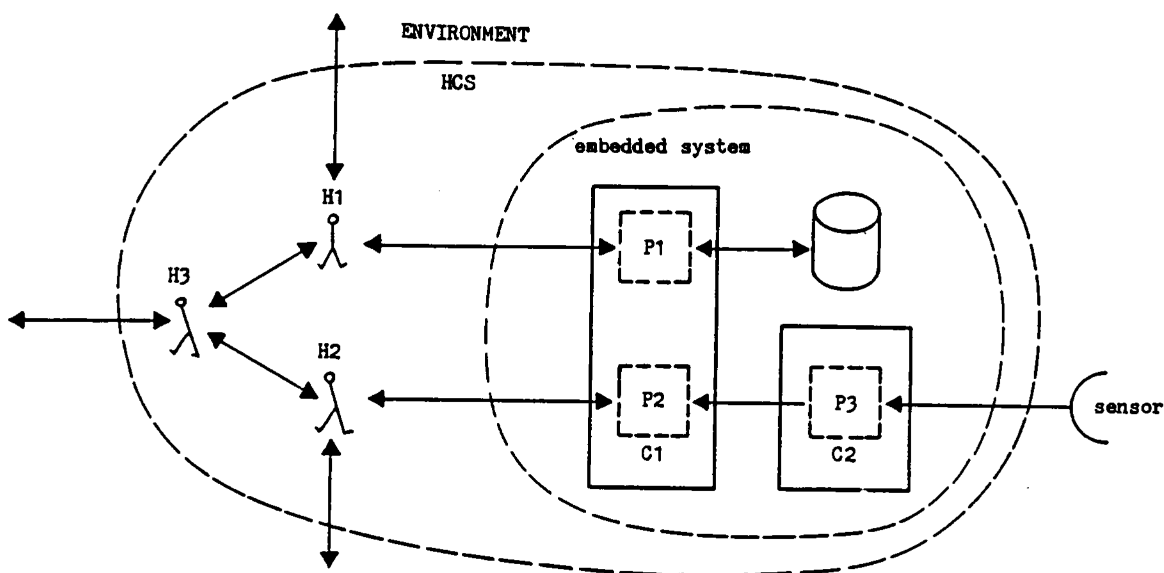

Figure 5.1 Generalized, informal representation of a human-computer system.

system are not included in the definition of the system. In a human-computer system, communication may be among system components (i.e., among computers, among humans, and between

humans and computers), and between system components and the environment (e.g., other humans, machines, etc.).

The graphical symbols and terminology presented so far allow representation of only the interactive part of a human-computer system. This chapter extends the representation technique to cover the human's non-interactive operation as well.

Figure 5.2 illustrates the highest-level functional structure representing the operation of the human-computer

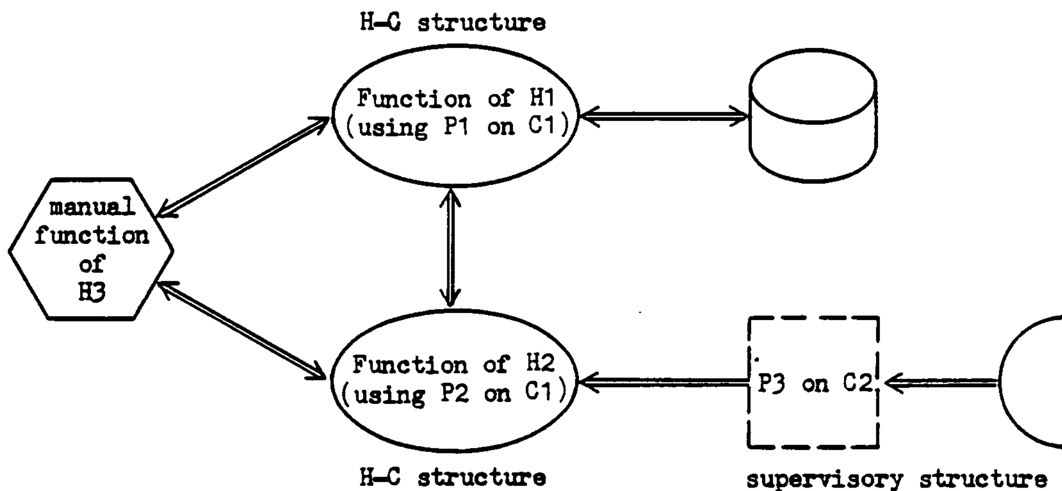

Figure 5.2. Top-view of the structures of Figure 6.1.

system of Figure 5.1 in SUPERMAN's notation. Program P3, which runs on computer C2, is a computational supervisory structure, composed of software supervisory cells. In the figure, operation of the human H3 is represented with a hexagon, denoting an operation with no help from the computer (i.e., a manual operation). The ellipse symbolizes a high-level human-computer (H-C) function and indicates the presence of computer tools in both H1's and H2's operation. The arrows connecting these concurrent functions represent data flow.

The general view of an H-C function structure is illustrated in Figure 6.3. The top-most H-C function is the most abstract definition of what the human does in the operational procedure structure. An H-C function structure identifies the allocation of control and data flow supervision between the human and the computer. The flow of control specified in the operational procedure structure is performed by the human, and that specified in the software structure is performed by the computer. The software structure begins with the D-C functions.

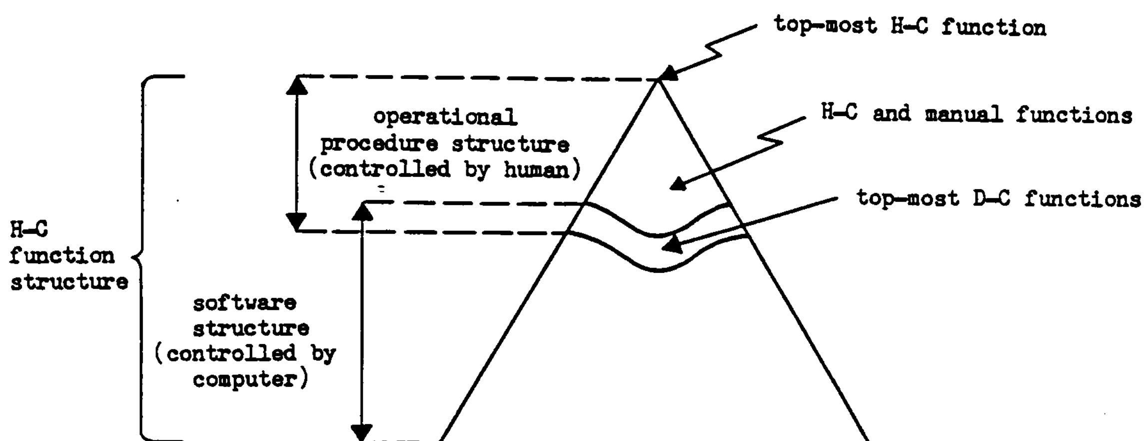

Figure 5.3. The H-C function structure.

5.1 THE OPERATIONAL PROCEDURE STRUCTURE

An *operational procedure structure* defines the procedural operation for the human(s) at abstract levels. In Section 4, we introduced the concept of a supervisory structure composed of supervisory cells, each of which in turn is composed of a supervisory function and a supervised flow diagram (SFD). A human's operation (as part of a human-computer system) is represented in a structurally similar way.

In Figure 5.4 a hypothetical operational procedure structure is illustrated. Just as, at execution time, a

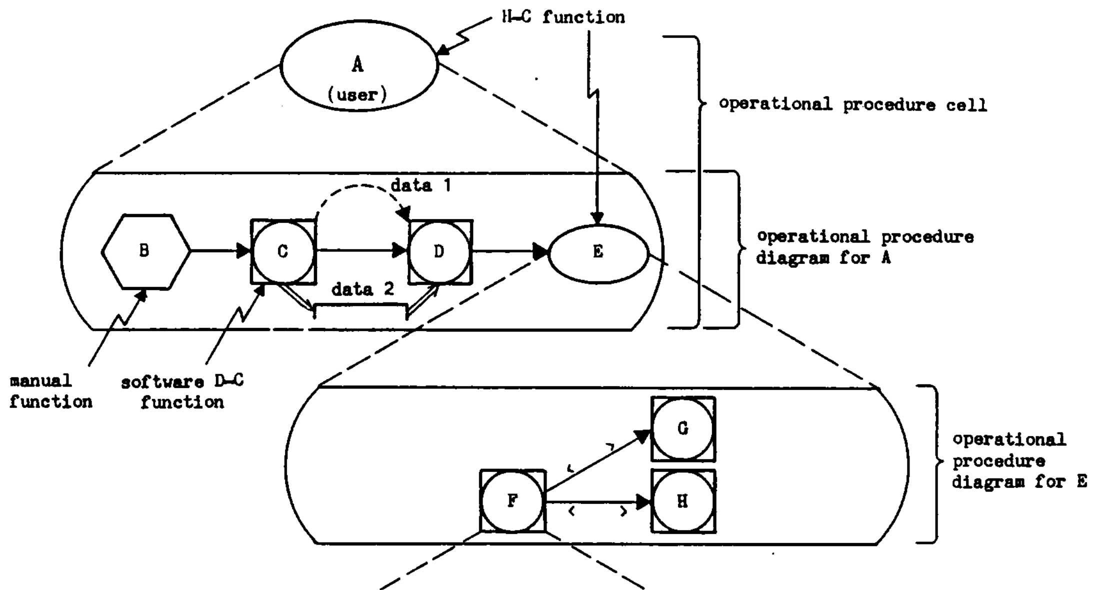

Figure 5.4. An operational procedure structure.

computer traverses the supervisory structure through the levels of SFDs and along the control lines, a human does the same thing in an operational procedure structure. For example, in Figure 5.4, the user performs the function A as explained below. Manual function B is performed first, and then the D-C function C is activated. During this interac-

tive session, control is taken over by the computer, leading the user through the control paths specified in the software structure under the D-C function C. (Figure 5.4 also illustrates allocation of data flow management. While data_2 is stored and retrieved by the computer, data_1 is received and used by the human). After C has been completed, the human supervisor regains control and activates the D-C function D. Next in the sequence is an H-C function and the user follows the control lines of this function's operational procedure diagram. In particular, the user makes a mental (not automated) decision regarding which of G or H will follow function F. As was discussed for a software function (Section 4.2.5), a human can initiate, communicate with, control, synchronize, and terminate the operation of other concurrent functions (e.g., humans, programs). An operational procedure structure can also be used as an operational diagram by the operators, and would be helpful for teaching the system to new personnel.

Analogous to the software supervisory cell, a cell in an operational procedure structure is called an <u>operational procedure cell</u>. This cell consists of a high-level human-computer function (H-C) and an <u>operational procedure diagram</u> (OPD). The constituent symbols of an operational procedure structure are illustrated in Figure 5.5. An operational

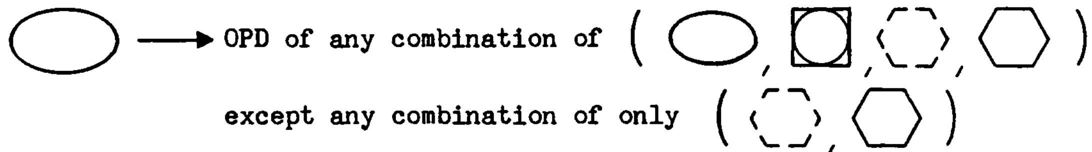

Figure 5.5. Constituents of an operational procedure structure.

procedure structure can contain human-computer functions (ellipses), interactive software functions (D-C functions), and manual functions (hexagons). All interactive tools made available to the human are shown on the operational procedure structure as D-C functions. Referring back to Figure 5.3, the reader will see that D-C functions are on a transition band, where the control is taken over by the computer. The software part of the H-C function structure, shown in Figure 5.3, is developed by expanding these D-C functions in terms of software supervisory cells.

All human functions in an operational procedure diagram are manual functions independent of the computer. Hence, when the HFE looks at the system's operational structure, the HFE can see where the user gets automated support (i.e.,

the D-C functions), and where the user does not (i.e., manual management of control and data flow and manually implemented functions). User management of control must be evaluated with regard to the user's cognitive capabilities.

Figure 5.6 shows the grammar for SUPERMAN's notational specification language, where the arrow means "can be expanded into", and the vertical bar means "or". The grammar allows derivation of system specifications from completely manual operation to fully automated operation. Also, any combination of holistic and software approaches in developing the whole or parts of a human-computer system (e.g., any combination of top-down or bottom-up approaches) is possible. A few examples of operational structures derivable from the grammar are hierarchically organized programs, hierarchically organized humans (e.g., an ellipse representing the top-most supervisor) who operate and communicate with or without computer tools, and a system of concurrently operating components (e.g., humans, programs) with no supervisor.

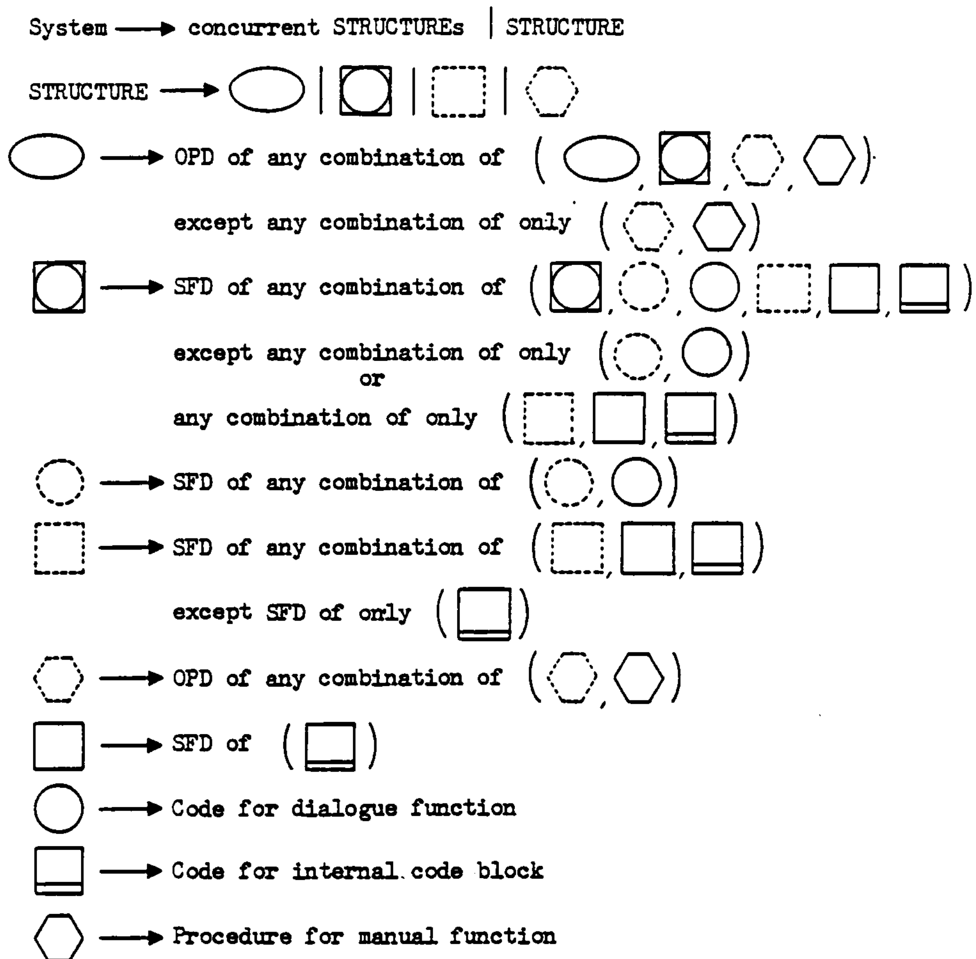

Figure 5.6. The grammar for SUPERMAN's notational language.

5.2 THE HUMAN-COMPUTER SYSTEM LIFECYCLE IN THE HOLISTIC APPROACH

The fundamental difference in the human-computer system lifecycle between the holistic approach and the software approach is in the requirements specification phase. Also, an

operational procedure structure is a valuable tool in human-computer system enhancement and related software maintenance. These points are discussed below.

5.2.1 Requirements Specification

In the human-computer system lifecycle, the operational procedure structure is constructed prior to the software requirements specification in order to identify the necessary software tools (e.g., the D-C functions). In some cases, it might not be possible to decide on the type of a function (e.g., D-C, or H-C, etc.) before thorough analysis. Therefore, before allocating human and computer responsibilities (e.g., manual or computerized functions, human control or computer control), the functional structure of the system might need to be constructed in terms of general function symbols. For example, Figure 4.1 was constructed using general function symbols which are rectangles with oval ends. Performance requirements, if needed, are recorded on the general function symbols. Each general function symbol is eventually replaced with a symbol of the graphical grammar in Figure 5.6. Time scales associated with the functional paths of the operational procedure diagrams can be used in evaluating alternative automation configurations in view of performance requirements. Finally, D-C functions on the op-

erational procedure structure are used to construct the top level of the software behavioral structure.

5.2.2 Role of the Operational Procedure Structure in System Maintenance

Software maintenance is due to changes in the operational policies and needs of humans in the human-computer system. The human-computer system operational representation (i.e., the operational procedure structure) is valuable in system maintenance. Any change in the operational form of the human-computer system would point to the parts of the software which are affected by the change. For example, a decision might be made to automate a currently manual function. The relationship of this function with the surrounding functions, whether manual or automated, can be readily observed and easily integrated into existing software. Changing an operational policy might also affect part of the human-computer system's operation. The operation to be changed directly shows the parts of the behavioral structure that are affected. Those parts are then replaced by the automated functions of the new partial behavioral structure.

Chapter VI

EVALUATION OF SUPERMAN

Research on the evaluation of software methodologies is a relatively new activity within the software engineering discipline. Some recent work addresses applicative evaluation of the existing methodologies [BASIV84, LAPRJ84, WEISD85, JEFFD85]. Applicative evaluation is the process of identifying the problems encountered in using a methodology and the sources of these problems. The approach used for developing SUPERMAN is the so-called formative evaluation, and this activity includes applicative evaluation [WILLR84, DICKW78]. Formative evaluation is an iterative revision process which utilizes the system user's feedback to increase the efficiency and effectiveness of a system being developed. SUPERMAN's development through formative evaluation is guided by a top-down problem solving strategy, and the objective is to eliminate the sources of problems before they become part of the methodology.

6.1 EVALUATION METHOD

The formative evaluation method used for SUPERMAN is based on target system evaluation, which includes the validation activity (e.g., determining how well a system serves or will serve the end-user's needs.) Boehm [BOEHB83], among others, states that validation should be performed continuously to avoid massive and costly changes. According to Adrian, Branstad, and Cherniavsky [ADRIW82], such disciplined manual techniques as walk-throughs, reviews, and inspections, applied to all stages in the lifecycle, are among the most successful validation techniques. Prototyping (see introductory chapter) is another validation technique, and is also used for the human engineering of a system. Figure 6.1 illustrates a formative evaluation process which includes the above validation activities. In the figure, the output of the box "develop the target" may be in the form of a specification (e.g., a design representation) or a working system (e.g., a prototype or a final system). The activity in the box "evaluate the target," depending on the form of the input to the box, involves one or more of the evaluation techniques (e.g., walk-throughs, reviews, inspections, prototyping).

The process of building quality into an automated methodology is similar to that of building quality into a target

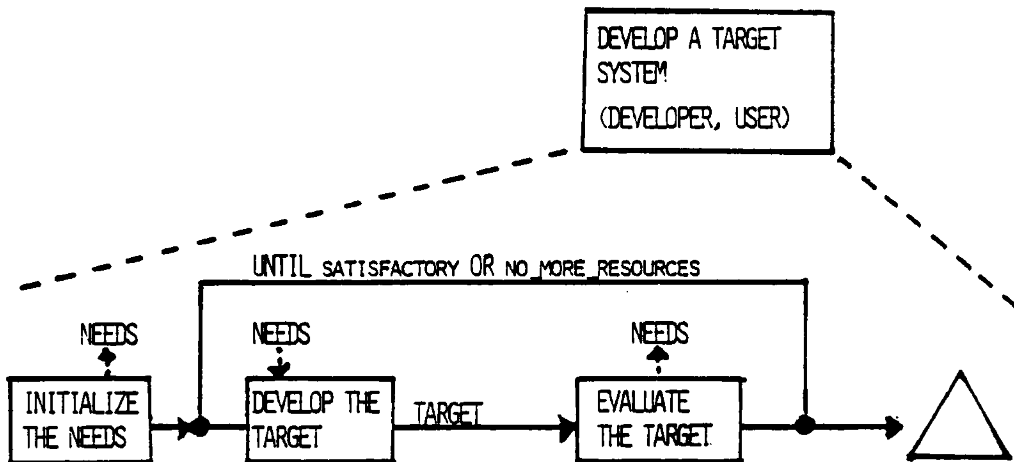

Figure 6.1. A formative evaluation procedure for a target-system.

system. A methodology developer views his product as a target system, and views the target system developers as the end-users. Hence, the approach discussed above (and illustrated in Figure 6.1) applies to methodology development. Figure 6.2 illustrates the development procedure designed for SUPERMAN at the beginning of this research [YUNTT82]. In Figure 6.2, the activity of target system development is contained in the methodology development activity; the methodology developer who works in the upper cell continuously

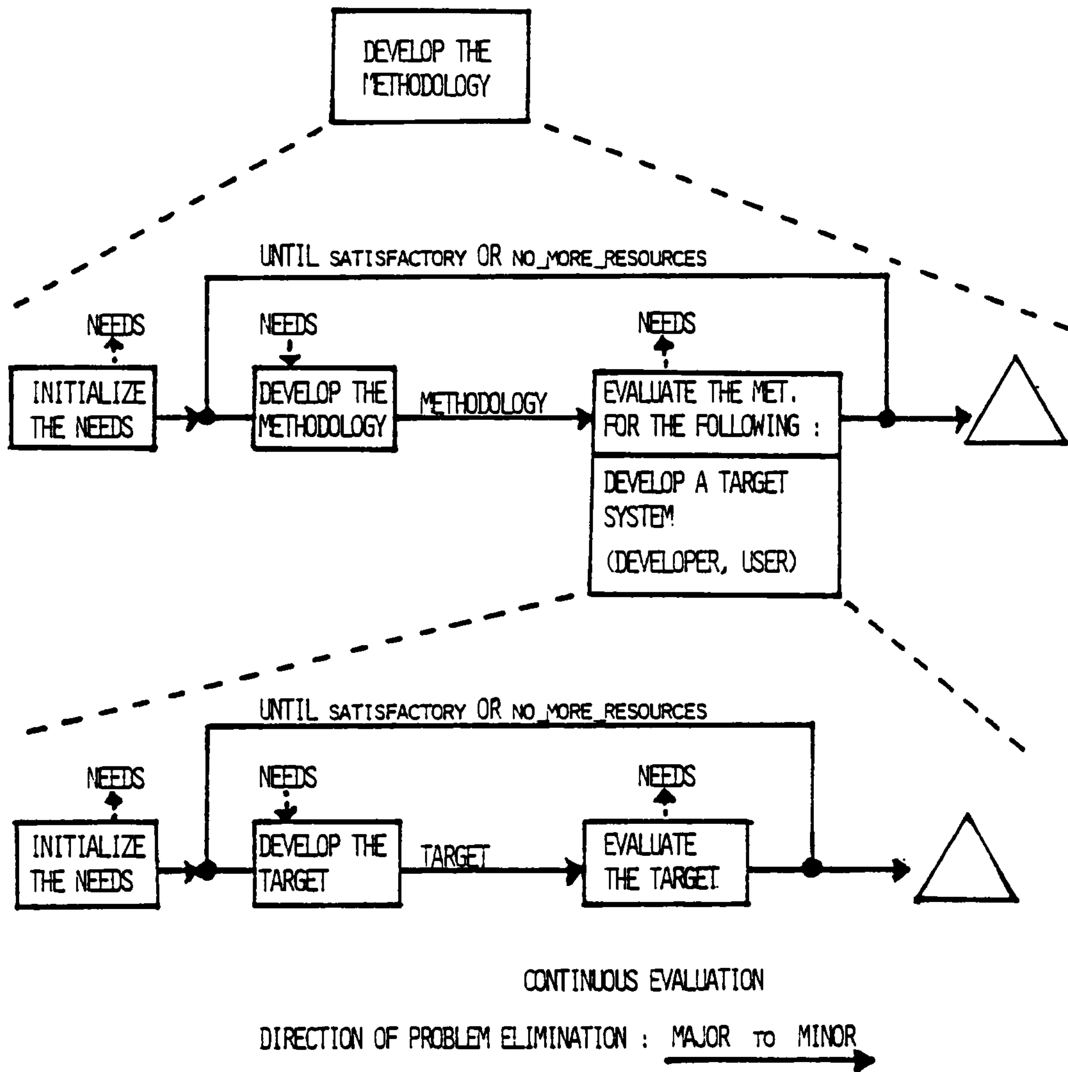

Figure 6.2. Formative evaluation procedure for SUPERMAN.

evaluates his product with regard to the needs of the target system developer who uses (or will use) the methodology in the lower cell. The methodology developer is also a metho-

dology user, and, through mental prototyping, searches for a satisfactory solution. The problem is large, and requires efficient use of developmental resources (e.g., time, capital, cognitive resources). The strategy of the developer is to identify and eliminate the highest level problems first, and, in the presence of major problems, to avoid applicative evaluation. This is because, eliminating a major problem will eliminate consequent minor problems. Otherwise, applicative evaluation of a system with major problems will make visible the consequences of abstract major problems; but will not give solutions to existing major problems. Example of a high level problem, with which the methodology developer is particularly concerned, is the lack of a unified representation technique which results in phase-product transformations.

SUPERMAN has evolved through the iterative revision and top-down problem solving process described above. During this process, the human factors engineering and software engineering literature has been a valuable source of user-needs such as abstraction, modularization, structure, concurrency, etc. The evaluators actively involved in SUPERMAN's evolution have been the people in the Dialogue Management System (DMS) project.

The next section presents a recent survey-based applicative evaluation.

6.2 A SURVEY-BASED EVALUATION OF SUPERMAN

This section contains an overview of a survey study whose goal is to improve SUPERMAN. (A copy of the survey is in appendix A.) The survey was completed by six computer scientists who have used SUPERMAN in the DMS project.

Figure 6.3 illustrates an evaluation criteria chart constructed for SUPERMAN. This chart serves as a work structure for evaluation of SUPERMAN features, and the evaluation is divided into five major parts (see the top level nodes of the chart.) Nodes of this tree represent evaluation criteria at abstract levels within each part. Each leaf criterion is linked to a list of numbers which represent 20 separate SUPERMAN features. For example, the left-most path represents: ease of learning is supported by the consistency criterion which, in turn, is supported by the 4th feature of the list (i.e., unified representation). The 20 SUPERMAN features considered in the evaluation are:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

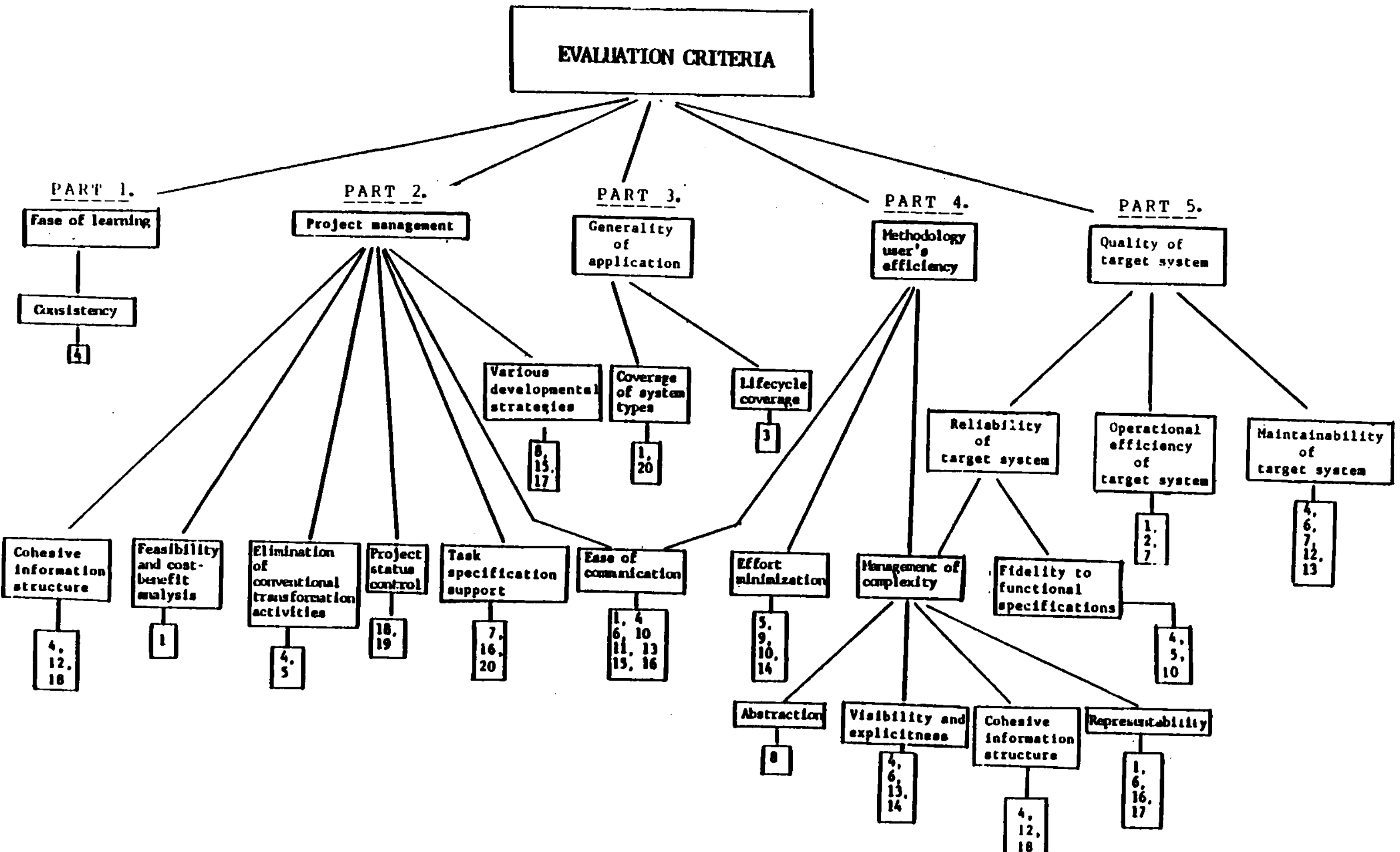

Figure 6.3. Evaluation Criteria Chart for SUPERMAN.

2) Integrates human-factor inputs from the beginning of a system design.
3) Is used throughout the system lifecycle.
4) Offers a single, unified system representation at all levels of development.
5) Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.
6) Embodies both data flow and control flow in a single representation.
7) Supports directly the separation of human-computer dialogue and computation.
8) Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development.
9) Is rich with supporting automated tools.
10) Produces executable requirements specifications which hide the technical aspects of design.
11) Produces a behavioral model of the system that can be demonstrated ("simulated") before any code is written.
12) Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

13) Represents programs in a Graphical Programming Language (GPL).

14) Reduces textual coding to non-control statements of conventional textual languages.

15) Supports non-procedural as well as procedural representation.

16) Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.

17) Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.

18) Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).

19) Notationally specifies the boundaries of lifecycle phases.

20) Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

For each path of the evaluation criteria chart, the methodology users were asked to rate the effectiveness of

each feature using the following scale: V (very effective), M (moderately effective), N (not effective), I (interfering), S (strongly interfering). Also, users were asked to include their comments.

Table 1 illustrates a global view of the ratings. Each line of Table 1 contains a summation of ratings for all occurrences of a feature in the evaluation chart (i.e., Figure 6.3). For example, in the evaluation chart, feature 3 has one occurrence, and, as the entry in Table 1 indicates, this feature has been rated by the group of users once (i.e., by 5 of the 6 users). As another example, feature 4 has several occurrences and therefore has been rated several times. The following sections present a global view of the survey results. The details for each of the five parts are in APPENDIX B.

6.2.1 Comments on Strengths of SUPERMAN

The survey showed that the strongest characteristic of the methodology is the graphical and executable representation technique (i.e., GPL) which integrates functional decomposition, the lifecycle phases, separation of dialogue and computation, and non-functional requirements. The users' comments on the overall strengths of the methodology follow.

TABLE 1

FEATURE	V	M	N	I	S
1. Function allocation	8	10	4		
2. Integration of human factors inputs........	1	2	1		
3. Lifecycle coverage	4	1			
4. Unified representation	26	26	3		
5. Conceptual phase transitions	11	6			
6. Data flow & control flow	9	13	1		
7. Separation of dialogue and computation	14		1		
8. Top-down and/or bottom up development	3	7			
9. Rich with automated tools	3	2			
10. Executable specifications	10	5			
11. Behavior demonstration	2	1		1	
12. Cohesive functional structure	8	9		1	
13. GPL	10	7			
14. No textual coding of control..............	7	2	1		1
15. Procedural & non-procedural representations		2	2		
16. "Who", "What", "How"	6	7	2		
17. OSD & SFD	2	2			
18. Structured information directory	8	6			
19. Notationally specified phase boundaries ...	1	1	1		
20. Applicable to any procedural system	3	6			

"To me, its strength is graphical representation of the target system at all levels, and especially the one-to-one conversion of the graphics to source code."

"Very easy to learn and use - small alphabet and simple syntax, but high expressive power. SUPERMAN produces very clear and modular representations, produces very modular designs, promotes reusability of functions, and eases communication among designers."

"Single representation throughout the lifecycle is the most powerful part of the methodology. It allows the tasks to be broken up into nearly independent units. This is very handy for team projects."

"Nice framework in which to display control flow and specify user-computer interaction. Serves as a technical documentation of completed systems."

> "Dialogue independence"
>
> "SFD's are much more expressive than code. This encourages spending more time in design. I preferred to work exclusively with SFD's and never looked at the code."

Users found SUPERMAN to be simpler and more comprehensive than other methodologies:

> "SFD's are more descriptive of hierarchy and sequence than either flowcharting or Structered Design calling tree used singly or together. SUPERMAN is less elaborate (fewer symbols, simpler diagrams) then SADT. Thus, SUPERMAN is easier to learn and to use."
>
> "Represents data flow and control flow, visually displays functional hierarchies, visually separates function types, has fairly well defined support environment, is complete (in comparison) and consistent."

The above strengths originate in the supervisory concept, which allows one to represent the conception of a system's functional operation. As feature 18 indicates, this concept also integrates the non-functional requirements and development related information into a target system representation (a desirable characteristic which is partially achieved by other methodologies [ROMAG85].) Some users found this to be "<u>very</u> <u>very</u> helpful" and "a very important feature in any non-toy methodology."

6.2.2 The Stated Needs of the Users

The difficulties encountered by the users did not involve high level problems, but more detailed matters. These are presented below.

Information management

Most users stated the need for mechanisms and automated support to construct and manage development related information. Particularly, users asked for

- tools to help trace from non-functional specifications (e.g., design decisions) to functions of the supervisory structure and vice versa;
- means for managing status information (e.g., which modules are finished, and are incomplete); and
- tools for managing various kinds of documentation such as developer's notes, personnel allocation, or project history.

Team-work

Users did not have difficulties working within a team, but did experience problems with regard to cooperation among teams. Some problems involved: maintaining inter-team consistency, and determining what teams are affected by changes (e.g., "who needs to know this decision?", "what modules are affected?")

Data structures and data dictionary

The majority of users felt the need for a formal representation of data structures to alleviate the inconsistent definitions, communication difficulties, and difficulties in conceptualizing data flow which have resulted from the informal structure definitions. In response to a survey question, all users agreed that a data dictionary, accessed via data selection on the screen, and used for automatic code generation, would be very useful. For a data dictionary, the users agreed on the following proposed information: information on simple variables (e.g., name, type, dimension, range, format, common block, source module, narrative description), information on data structures (combination of graphical structures and textual descriptions), information on files. In addition, the users suggested the following: user defined data types, abstract data types, data instances, scalar list of allowable values.

Contents of Internal Code Blocks

For producing optimized code, some users needed the control constructs of low-level languages (e.g., JUMP), and, it was generally agreed that the ICBs should not be restricted to non-control statements. Also, for data declarations, in addition to the data dictionary discussed above, the users stated that the ICB's should be available as an option.

Testing

For testing activity (i.e., for the Behavioral Demonstrator), the users stated the following needs: asynchronous testing, interrupt testing, a tool which turns the results of testing into system modifications, a tool to report all conditions that must hold in order for execution to be at any particular point in the supervisory structure, and, tools for managing test plans, test procedures, and test results.

GPL

Users remarked that the GPL editor-compiler is a most desirable tool, but they added that some activities need automated support, such as the verification of notational rules and syntax, configuration management, and construction of SFD's with varying levels of detail.

Notational extensions

All the users agreed that the current "recursion" and "iteration" constructs are difficult to use. SUPERMAN's notation was criticized for not including device control, asynchronous events, and interrupt handling. Other notational needs are: function and system version numbers, configuration paths (e.g., highlighted nodes), CPM or PERT like charts for project management, notation for metering and performance monitoring, and notation for exceptional conditions. Also, a large overview diagram, to help see the relationships among widely separated functions, was suggested.

6.2.3 The Author's Comments on the Evaluation Activity

Though this survey provoked useful criticisms of SUPERMAN, it produced a partial evaluation of the methodology. Most of the users who participated were experienced in software development, and so evaluated SUPERMAN in this context rather than in a broader context of human-computer system development. Also, some users had little knowledge of the other methodological approaches and concepts. A future evaluation activity should involve a team of experts from both software engineering and human factors engineering disciplines. Walk-throughs and discussions would provide valuable data. This survey also showed that some users had

incomplete knowledge of SUPERMAN. Consequently, the evaluation of some features were negatively affected. It would be helpful if future evaluators were adequately schooled in the use of software engineering concepts (e.g., data encapsulation, modularization).

The evaluation model (Figure 6.3) used for this survey has been compiled by the author, and is based on the author's experience, knowledge, and the desirable methodological characteristics cited in the software engineering literature. For a future evaluation activity, the completeness and the accuracy of this model should also be validated.

Chapter VII

CONCLUSION AND FUTURE DIRECTIONS

The goal of SUPERMAN is to provide for low development cost and high product quality. The users' feedback indicate that this research is progressing in the right direction to achieve its goal. In general, the users rate SUPERMAN as an effective methodology with powerful features. However, as discussed in Chapter 6, most of such judgement is based on limited experience in developing medium-size interactive software. Hence, the formative-evaluation of SUPERMAN will be continued by utilizing extensive inputs of experts in software engineering and human factors engineering fields. In the past, the top-down problem solving strategy used for developing SUPERMAN proved to be very effective; it was observed that many of the problems addressed in current literature were consequently solved in SUPERMAN by solving higher-level problems. Future research will be guided by the same strategy.

The evaluation of the methodology showed that the strongest characteristic of the methodology is the unified, graphical, and executable representation technique. Future work will primarily focus on the extension of this graphical programming language, and on the conceptual and automated

tools to help manage the information organized around this language. Many of these issues were discussed in Chapter 6.

An ultimate goal of this work is to automatically generate system designs from requirements specifications. Currently, most of design activities are performed manually. This process can be largely automated by extending the notational semantics of the current specification language to include software engineering concepts and the host system behavior. The supervisory concept allows for such an iconic language, and makes possible the design of an automatic system generator.

REFERENCES

ADRIW82 Adrion, W. R., Branstad, M. A., Cherniavsky, J. C., "Validation, Verification, and Testing of Computer Software", ACM Computing Surveys, Vol 4., No. 2, pp. 159-192, (1982).

ALAVM84 Alavi, M., "An Assessment of the Prototyping Approach to Information Systems Development", CACM, Vol. 27, No. 6, pp. 556-563, (1984).

ALFOM85 Alford, M., "SREM at the Age of Eight; The Distributed Computing Design System", IEEE Computer, Vol. 18, No. 4, pp. 36-40, (1985).

ANDEB75 Anderson, B. F., Cognitive Psychology, Academic Press, New York, (1975).

BABBR85 Babb II, R. G., Keiburtz, R., Orr, K., Mili, A., Gearhart, S., Martin, N., "Workshop on Models and Languages for Software Specification and Design", IEEE Computer, Vol 18, No. 3, pp. 103-108, (1985).

BASIV84 Basili, V. R., Weiss, D. M., "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 728-738, (1984).

BOEHB83 Boehm, B. W., "Seven Basic Principles of Software Engineering", The Journal of Systems and Software, Vol. 3, pp 3-24, (1983).

BOEHB84a Boehm, B. W., "Verifying and Validating Software Requirements and Design Specifications", IEEE Software, Vol. 1, No. 1, pp. 75-88, (1984).

BOEHB84b Boehm, B. W., Gray, T. E., Seewaldt, T., "Prototyping Versus Specifying: A Multiproject Experiment", IEEE Transactions on Software Engineering, Vol. SE-10, No. 3, pp. 290-302, (1984).

BOURL81 Bourne, L. E., Dominowsky, L. R., Loftus, E. F., Cognitive Processes, Prentice Hall, Inc., Englewood Cliffs, NJ, (1981).

CAINS75 Caine, S., and Gordon, E., "PDL - A Tool for Software Design", in Tutorial on Software Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 161-0, pp. 380-385, (1980).

CHAND80 Chand, D. R., Yadav, S. B., "Logical Construction of Software", CACM, Vol. 23, No. 10, pp. 546-555, (1980).

DAVIA82 Davis, A. L., Keller, R. W., "Data Flow Program Graphs", IEEE Computer, Vol. 15, No. 2, pp. 26-41, (1982).

DICKW78 Dick, W., Carey, W., Systematic Design of Instruction, Scott, Foresman and Company, Glenview, Illinois, (1978).

DRAPS85 Draper, S. W., Norman, D. A., "Software Engineering for User Interfaces", IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, pp. 252-258, (1985).

ENOSJ78 Enos, J. C., "Tools for Embedded Software Design Verification", in Tools for Embedded Computing System Software, pp. 93-96, NASA Conference Publication 2064, Preprint for a workshop held in Hampton, Virginia, November 7-8, (1978).

FITTM79 Fitter, M., Green, T.R.G., "When Do Diagrams Make Good Computer Languages?", Int. J. Man-Machine Studies, Vol. 11, No. 2, pp. 235-261, (1979)

FREEP83a Freeman, P., "Requirements Analysis and Specification: The First Step", in Tutorial on Software Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 205-5, pp. 79-85, (1983).

FREEP83b Freeman, P., "Nature of Design", in Tutorial on Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 205-5, pp. 2-22, (1983).

FUJIM78 Fujii, M. S., Ikezawa, M. A., "Use of Symbolic Execution in Verification and Validation", in Tools For Embedded Computing Systems Software, NASA Conference Publication 2064, Preprint for a workshop held in Hampton, Virginia, November 7-8, (1978).

GLINE84 Glinert, E. P., Tanimoto, S. L., "Pict: An Interactive Graphical Programming Environment", IEEE Computer, Vol. 17, No. 11, pp. 7-24, (1984).

GOMAH82 Gomaa, H., Scott, D. B. H., "Prototyping as a Tool in the Specification of User Requirements", Proceedings of the 6th International Conference on Software Engineering, September 13-16, Tokyo, Japan, pp. 333-339, (1982).

HAMIM76 Hamilton, M., Zeldin, S., "Higher Order Software - A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, pp. 9-25, (1976)

HAMIM83 Hamilton, M., Zeldin, S., "The Functional Lifecycle Model and Its Automation: USE.IT", The Journal of Systems and Software, Vol. 3, No. 1, pp. 25-62, (1983).

HARTH82 Hartson, H. R., Ehrich, R. W., "The Management of Dialogue for User/Software Interfaces", Department of Computer Science, V.P.I. & S.U., Technical Report (1982).

HARTH84 Hartson, H. R., Johnson, D. H., Ehrich, R. W., "A Human-Computer Dialogue Management System", INTERACT '84, First IFIP Conference on Human-Computer Interaction, London, England, pp. 57-61, (1984).

HEITC83 Heitmeyer, C. L., "Abstract Requirements Specification: A New Approach and Its Application", IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, pp. 580-589, (1983).

HOSIJ78 Hosier, J. (ed.), Structured Analysis and Design, Vol. 1: Analysis and Bibliography, Infotech State of the Art Report, Infotech International, Maidenhead, England, pp. 195-208, (1978).

IVERK80 Iverson, K. E., "Notation as a Tool of Thought", Communications of the ACM, Vol. 23, no. 8, pp. 444-465, (1980).

IWAMK81 Iwamato, K., Shigo, O., "Unifying Data Flow and Control Flow Based Modularization Techniques", Proceedings of IEEE Compcon 81, Washington D. C., pp. 271-277, (1981).

JACKM75 Jackson, M. A., Principles of Program Design, Academic Press, New York, (1975).

JACKM83 Jackson, M. A., System Development, Prentice-Hall, Inc., NJ, (1983).

JEFFD85 Jeffery, D. R., Lawrence, M. J., "Managing Programming Productivity", The Journal of Systems and Software, Vol. 5, No.1, pp. 49-58, (1985).

JOHND82 Johnson, D. H., and Hartson H. R., "The Role and Tools of a Dialogue Author in Creating Human-Computer Interfaces", Department of Computer Science, V.P.I. & S.U., Technical Report CSIE-82-8 (1982).

KLINR81 Kling, R., "The Organizational Context of User Centered Designs", in Tutorial: Software Development Environments", Anthony I. Wasserman (Ed.), IEEE Catalog No. EHO 187-5, pp. 337-348, (1981).

LAPRJ84 Laprie, J-C, "Dependability Evaluation of Software Systems in Operation", IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 701-714, (1984).

LEHMM81 Lehman, M. M., "The Environment of Program Development and Maintenance - Programs, Programming, and Programming Support", in Tutorial: Software Development Environments, Anthony I. Wasserman (Ed.), IEEE Catalog No. EHO 187-5, pp. 3-14, (1981).

LUNDM79 Lundeberg, M., Goldkuhl, G., and Nilsson, A., "A Systematic Approach to Information System Development", Information Systems, Vol. 4, no. 2, pp. 93-118, (1979).

LUNDM83 Lundenberg, M., "An Approach to Involving User in the Specification of Information Systems", in Tutorial on Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 205-5, pp. 133-155, (1983).

MARTJ82 Martin, J., Program Design Which is Provably Correct, Savant Research Studies, 2 Nero Street, Carnforth, Lancashire, England, (1982).

MASOR83 Mason, R. E. A., Carey T. T., "Prototyping Information Systems", CACM, Vol. 26, No.5, pp 347-354, (1983).

MEISD71 Meister, D., Human-Factors : Theory and Practice, John Wiley & Sons, Inc., New York, (1971).

MYERJ75 Myers, G. J., Reliable Software through Composite Design, Mason/Charter Publishers, New York, (1975).

MYERJ78 Myers, G. J., Composite/Structured Design, Van Nostrand Reinhold, New York, (1978).

NASSI73 Nassi, I., Shneiderman, B., "Flowchart Techniques for Structured Programming", SIGPLAN Notices of the ACM, Vol. 8, No. 8, pp. 12-26, (1973).

NELSR83 Nelson, R. E., Haibt, L. M., Sheridan, P. B., "Casting Petri nets into Programs", IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, pp. 590-602, (1983).

OLIVA82 Olive, A., "DADES: A Methodology for Specification and Design of Information Systems", Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noorwijkerhoutt, The Netherlands, pp. 285-334, (1982).

PAIGM77 Paige, M. R., "On Partitioning Program Graphs", IEEE Transactions on Software Engineering, Vol. SE-3, no. 6, pp. 386-393, (1977).

PAIVA71 Paivio, A., Imagery and Verbal Processes, Holt, Rinehart, and Wiston, New York, (1971).

PARND72 Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", CACM, Vol. 5, No. 12, pp 1053-1058, (1972).

PENEM81 Penedo, M. H., "An Algorithm to Support Code-Skeleton Generation for Concurrent Systems", Proceedings of the 5th International Conference on Software Engineering, pp. 125-135, (1981).

PETEJ77 Peterson, J. L., "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, pp. 223-249, (1977).

PORCM82 Porcella, M., Freeman, P., Wasserman, A. I., "Ada Methodologies: Concepts and Requirements", Department of Defense, Ada Joint Program Office document, (1982).

PRATT71 Pratt, T. W., "Pair Grammars, Graph Languages and String-to-Graph Translations", Journal of Computer and System Sciences, Vol. 5, No. 6, pp. 560-595, (1971).

RAMAC84 Ramamoorthy, C. V., Prakash, A., Tsai, W., Usuda, Y., "Software Engineering: Problems and Perspectives", IEEE Computer, Vol. 17, No. 10, pp. 191-207, (1984).

RAMSH83 Ramsey, H. R., Atwood, M. E., Van Doren, J. R. "Flowcharts versus Program Design Languages: An Experimental Comparison", Communications of the ACM, Vol. 26, No. 6, pp. 445-449, (1983).

RIDDW81 Riddle, W. E., "An Assesment of Dream", in Tutorial: Software Development Environments, Anthony I. Wasserman (Ed.), IEEE Catalog No. EHO 187-5, pp. 195-210, (1981).

ROMAG85 Roman, G-C., "A Taxonomy of Current Issues in Requirements Engineering", IEEE Computer, Vol. 18, No. 4, pp 14-22, (1985).

ROSSD77 Ross, D. T. and Schoman, K. E., "Structured Analysis for Requirements Definition", in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wasserman (Eds.), IEEE Catalog No. 76CH1145-2C, pp. 86-95, (1977).

ROSSD85 Ross, D. T., "Applications and Extensions of SADT", IEEE Computer, Vol. 18, No. 4, pp. 25-34, (1985).

RZEPW85 Rzepka, W., Ohno, Y., "Requirements Engineering Environments: Software Tools for Modeling User Needs", IEEE Computer, Vol. 18, No. 4, pp 9-12, (1985).

SCHEP85 Scheffer, P. A., Stone, A. H., Rzepka, W. E., "A Case Study of SREM", IEEE Computer, Vol. 18, No. 4, pp. 47-54, (1985).

SHEPS81 Sheppard, S. B., Kruesi, E., Curtis, B., "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications", in Tutorial: Human factors in Software Development, Bill Curtis (Ed.), IEEE Catalog No. EHO 185-9, pp. 302-309, (1981).

SHNEB81 Shneiderman, B., Mayer, R., "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results", in Tutorial: Human factors in Software Development, Bill Curtis (Ed.), IEEE Catalog No. EHO 185-9, pp. 9-23, (1981).

SHNEB83 Shneiderman, B., "Direct Manipulation: A step beyond Programming Languages", Computer, Vol. 16, No. 8, pp. 57-69, (1983).

SMITD82 Smith, D.C., *et al.*, "Designing the Star User Interface", Byte, Vol. 7, No. 4, pp. 242-282, (1982).

STEVW74 Stevens, W. P., Myers, G. J., and Constantine L. L., "Structured Design", IBM Systems Journal, Vol. 13, No. 2, pp. 115-139, (1974).

STEVW81 Stevens, W. P., *Using Structured Design*, John Wiley and Sons, Inc., New York, (1981).

TRACW81 Tracz, W. J., "Computer Programming and the Human Though Process", in Tutorial: Human factors in Software Development, Bill Curtis (Ed.), IEEE Catalog No. EHO 185-9, pp. 24-34, (1981).

TEICD76 Teicherow, D., "PSL/PSA --A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", in Tutorial on Software Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 161-0, pp. 211-218, (1980).

THOMJ83 Thomas, J. J., Hamlin, G., "Graphical Input Interaction Technique", Computer Graphics, Vol. 17, No. 1, pp. 5-30, (1983).

WARNJ81 Warnier, J., *Logical Construction of Systems*, Van Nostrand Reinhold, New York, (1981).

WASSA77 Wasserman, A. I., "On the Meaning of Discipline in Software Design and Development", in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wasserman (Eds.), IEEE Catalog No. 76CH1145-2C, pp. 37-44, (1977).

WASSA80 Wasserman, A. I., "Information System Design Methodology", in Tutorial on Software Design Techniques, Peter Freeman and Anthony I. Wasserman (Eds.), IEEE Catalog No. EHO 161-0, pp. 25-44, (1980).

WASSA81a Wasserman, A. I., "Programming Systems - Language-Based Environments", in Tutorial: Software Development Environments, Anthony I. Wasserman (Ed.), IEEE Computer Society Press, pp. 53-55, (1981).

WASSA81b Wasserman, A. I., "The Ecology of Software Development Environments", in Tutorial: Software Development Environments, Anthony I. Wasserman (Ed.), IEEE Computer Society Press, pp. 47-51, (1981).

WASSA82a Wasserman, A. I., "User Software Engineering Methodology", in Information Systems Design Methodologies, ed. Olle Sol, and Verrijn-Stuart, North-Holland, pp. 589-628, (1982).

WASSA82b Wasserman, A. I., "User Software Engineering and the Design of Interactive Systems", Proceedings of the 6th International Conference on Software Engineering, September 13-16, Tokyo, Japan, pp. 387-393, (1982).

WEINV80 Weinberg, V., Structured Analysis, Prentice-Hall, Inc., Englewood Cliffs, NJ, (1980).

WEISD85 Weiss, D. M., Basili, V. R., "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory", IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, pp. 157-168, (1985).

WILLR84 Williges, R. C., "Evaluating Human-Computer Software Interfaces", Proceedings of the 1984 International Conference on Occupational Ergonomics, Toronto, Canada, (1984).

YAUSS81 Yau, S. S., Grabow, P. C., "A Model for Representing Programs Using Hierarchical Graphs", IEEE Transactions on Software Engineering, Vol. SE-7, no. 6, pp. 556-574, (1981).

YEHRT82 Yeh, R. T., "Requirements Analysis - A Management Perspective", Proc. COMPSAC'82, pp 410-416, (1982)

YOURE79 Yourdon, E., Constantine, L. L., Structured Design, Prentice Hall, Inc., Englewood Cliffs, NJ, (1979).

YUNTT82 Yunten, T., Hartson, H. R., "Human-Computer System Development Methodology for the Dialogue Management System", Department of Computer Science, V.P.I. & S.U., Technical Report CSIE-82-7 (1982).

ZAVEP82a Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Transactions on Software Engineering, Vol. SE-8, no. 3, pp. 250-269, (1982).

ZAVEP82b Zave, P., Yeh, R. T., "Executable Requirements for Embedded Systems", Proceedings of the 6th International Conference on Software Engineering, September 13-16, Tokyo, Japan, pp. 295-304, (1982).

ZAVEP84 Zave, P., "The Operational Versus the Conventional Approach to Software Development", CACM, Vol. 27, No. 2, pp. 104-118, (1984).

Appendix A

A QUALITATIVE EVALUATION SURVEY FOR SUPERMAN

PARTICIPANT INFORMATION

- Name, organization and position, address, telephone

- Means of contacting you for future communication, if different from above

- A methodology is an integrated set of concepts, methods, and tools to enhance productivity of system developers in producing high quality target systems. In the following, please outline your _experience_ in relation to system development and methodologies.

activities	length of time (months)	position
---------------	-----------------------	--------
---------------	-----------------------	--------
---------------	-----------------------	--------
---------------	-----------------------	--------
---------------	-----------------------	--------

- Please rate your knowledge of system development methodologies

______ very broad

______ broad

______ moderate

______ poor

- Means by which you learned SUPERMAN (check one or more)

_____ reading

_____ using

_____ discussions

_____ presentations

- Please rate your knowledge of SUPERMAN

______ very broad

______ broad

______ moderate

______ poor

TO THE EVALUATOR

The goal of this survey is to improve SUPERMAN by utilizing your inputs. A paper on SUPERMAN is enclosed for your reference. For assistance in SUPERMAN and/or in this survey, the following means of contacting Tamer Yunten are offered:

For evaluators on-campus:

- send VAX1 mail to YUNTEN;
- leave a note in Yunten's mailbox in McBryde 555
- office phone number: 961-5853
-

For evaluators off-campus:

- Mailing Address: Computer Science Department, McBryde 555, Virginia Tech, VA 24061
- office phone number: (703)961-5853
-

This survey is a tool to trigger your thinking. Please feel free to comment on anything at any level of detail. If you are not confident about your criticisms, please so indi-

cate. You do not have to answer all questions. If you encounter a question which you have already answered earlier, please so indicate. Please go through the survey sequentially.

OVERVIEW OF SUPERMAN
AND
AN EVALUATION MODEL

SUPERMAN attempts to integrate the concepts of human-machine system development and state-of-the-art software development methodologies. SUPERMAN can be used in the development of any procedural system, including completely manual systems, human-computer systems (e.g., automated environments, embedded systems, interactive systems) and non-interactive systems. The _supervised flow diagrams_ of SUPERMAN, in addition to being executable, integrate functions of many other representation techniques (e.g., data flow diagrams, Petri nets, flowcharts, operational sequence diagrams, PDLs, textual languages, etc.) into a single unified representation.

The product of the methodology is a graphical and executable structure called the _supervisory structure_. In developing software, the user builds a supervisory structure, which is translated into code to execute (i.e., a top-down, graphical specification and programming language). Figure A.1 illustrates an abstract representation of the supervisory structure and the lifecycle activities which operate on parts of the supervisory structure. As observed in the fig-

ure, instead of having separate work products for each life-

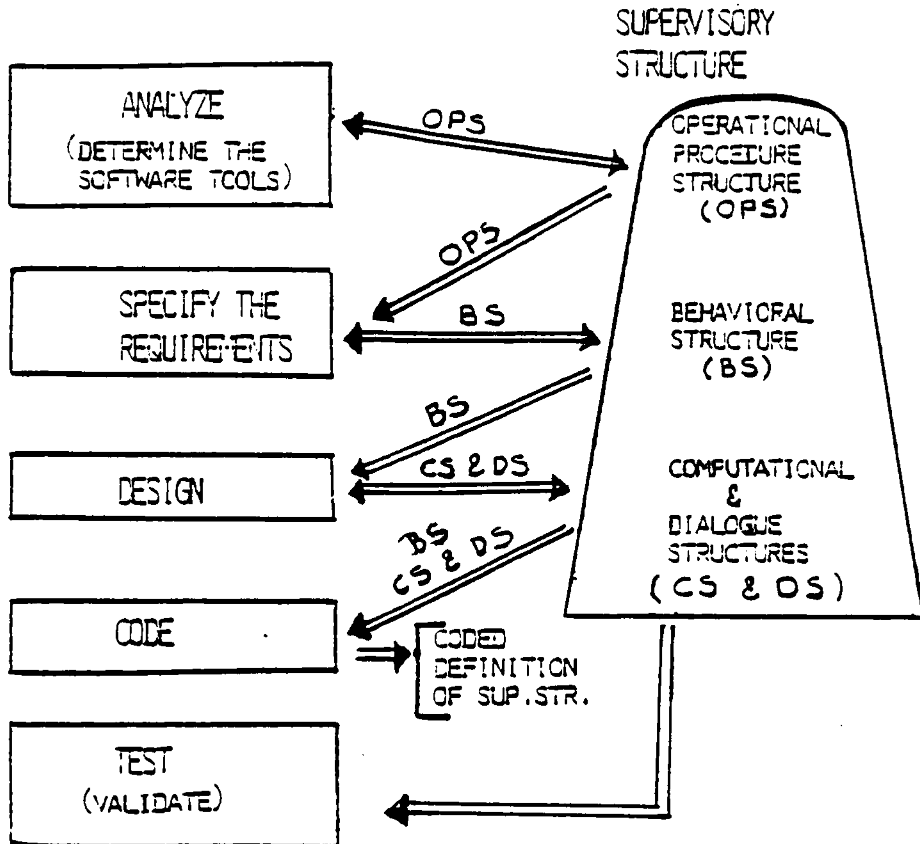

Figure A.1. The supervisory structure and the lifecycle activities.

cycle phase, there is one continuous structure. The features of the SUPERMAN methodology, in conjunction with an evaluation criteria chart, are listed below.

An Evaluation Model

Figure A.2 illustrates an evaluation criteria chart constructed for SUPERMAN. This chart serves as a work structure for evaluation of SUPERMAN features, and the evaluation is divided into five major parts. Nodes of this tree represent evaluation criteria at abstract levels within each part. Each leaf criterion is linked to a list of numbers which represent 20 separate SUPERMAN features. For example, the left-most path represents: ease of learning is supported by the consistency criterion which, in turn, is supported by the 4th feature of the list.

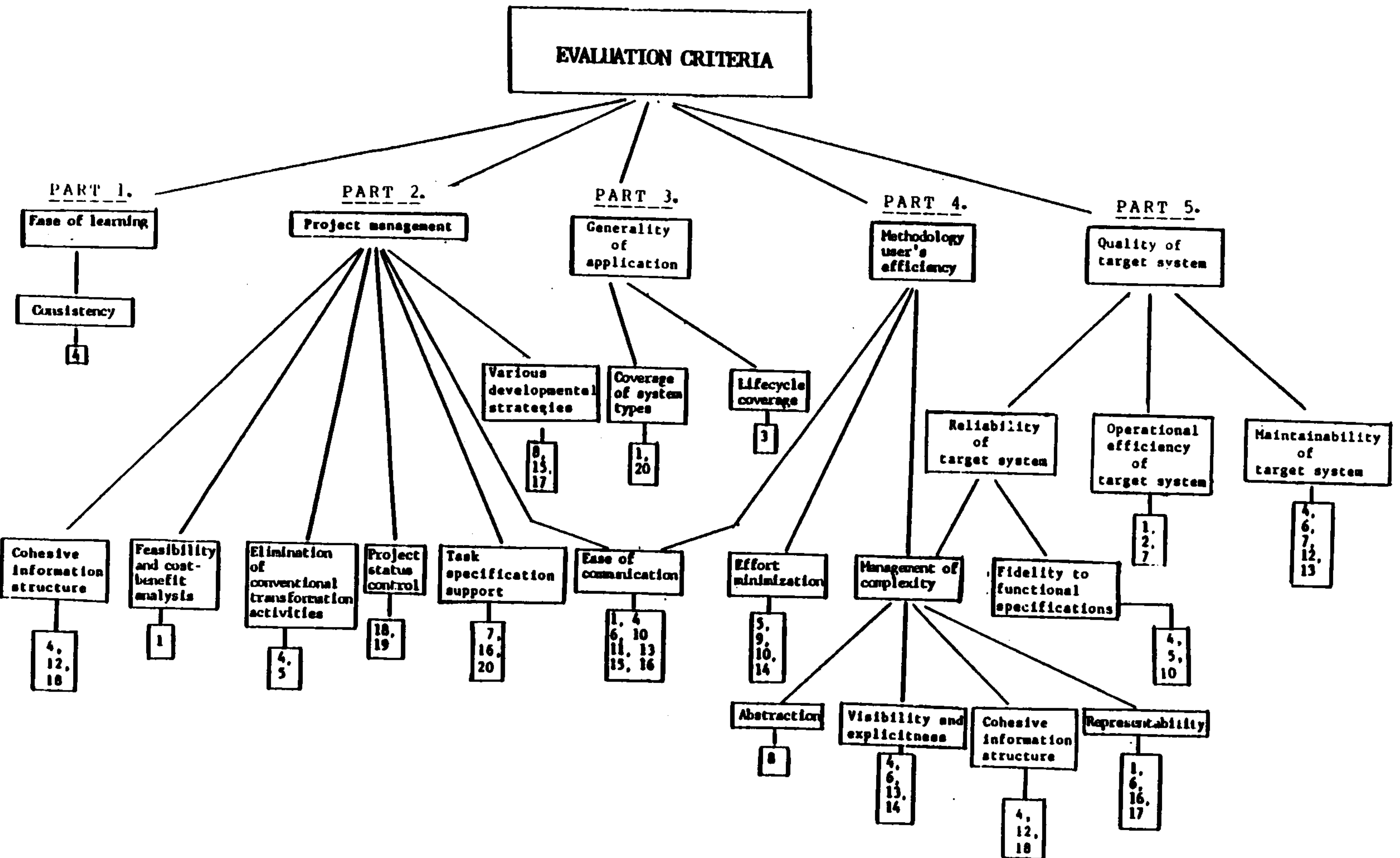

Figure A.2. Evaluation Criteria Chart for SUPERMAN.

The 20 SUPERMAN features considered in the evaluation are:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).
2) Integrates human-factor inputs from the beginning of a system design.
3) Is used throughout the system lifecycle.
4) Offers a single, unified system representation at all levels of development.
5) Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.
6) Embodies both data flow and control flow in a single representation.
7) Supports directly the separation of human-computer dialogue and computation.
8) Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development.
9) Is rich with supporting automated tools.
10) Produces executable requirements specifications which hide the technical aspects of design.
11) Produces a behavioral model of the system that can be demonstrated ("simulated") before any code is written.

12) Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

13) Represents programs in a Graphical Programming Language (GPL).

14) Reduces textual coding to non-control statements of conventional textual languages.

15) Supports non-procedural as well as procedural representation.

16) Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.

17) Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.

18) Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).

19) Notationally specifies the boundaries of lifecycle phases.

20) Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

Contents of the Survey

This survey consists of the following six parts:

1. Ease of Learning;
2. Project Management;
3. Generality of Application;
4. SUPERMAN User's Efficiency;
5. Quality of target system;
6. Overall Strengths and Weaknesses of SUPERMAN.

As shown in Figure A.2, the first five parts above stand for the first level nodes of the evaluation criteria chart, in left to right order. As you will see, later in the survey, each of these five parts is divided into two consecutive sections: a) Evaluation of SUPERMAN features and b) Miscellaneous questions. In all these five parts, it is very important that you specify problems or difficulties that you have encountered, observe, or foresee with SUPERMAN. Space is provided for your comments where appropriate. The last part addresses the overall evaluation of the strengths and weaknesses of SUPERMAN.

PART 1

EASE OF LEARNING

This part addresses the following subtree of Figure A.2.

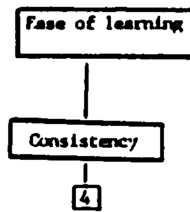

Figure A.3. Evaluation criteria and related SUPERMAN support feature for ease of learning.

A) PART1: EVALUATION OF SUPERMAN FEATURES IN SUPPORTING EASE OF LEARNING

Consistency - Uniformity in thought and practice.

Supporting feature:

4) Offers a single, unified system representation at all levels of development.

Below please circle the effectiveness of this feature in supporting ease of learning via supporting consistency.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	

B) PART1: GENERAL QUESTIONS ON EASE OF LEARNING

1. Please suggest (any) other features, which SUPERMAN lacks (i.e., not listed on pages 138 through 140), to support ease of learning.

2. Is it easy to learn SUPERMAN?

    _____ easy to learn

    _____ moderately difficult

    _____ difficult

   Please discuss what made it easy or difficult to learn.

PART 2

PROJECT MANAGEMENT

This part addresses the following subtree of Figure A.2.

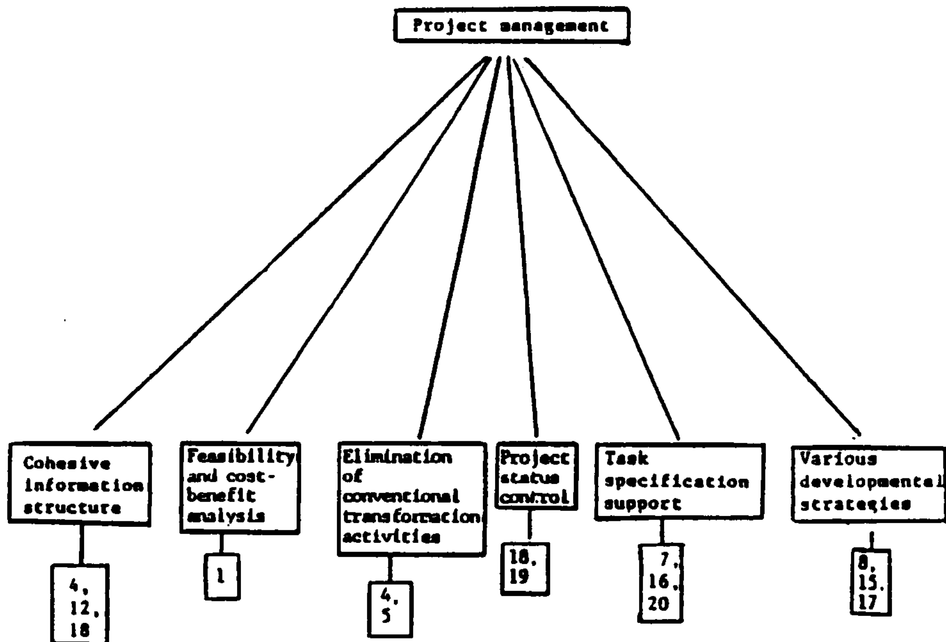

Figure A.4. Evaluation criteria and related SUPERMAN support features for project management.

In the following, you will evaluate each path above in left to right order. Please include descriptive comments, problems, and difficulties.

A) PART2: EVALUATION OF SUPERMAN FEATURES IN SUPPORTING PROJECT MANAGEMENT

Cohesive information structure - An information structure which causes the parts of a system to be clearly interrelated.

Supporting features:

4) Offers a single, unified system representation at all levels of development.

12) Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

18) Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).

Below please circle the effectiveness of these features in supporting project management via supporting cohesive information structure.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
12	V M N I S	
18	V M N I S	

Feasibility and cost-benefit analysis - The process of deciding whether a goal can be achieved (feasibility analysis) and, provided that it is feasible, the process of deciding whether an activity is worth undertaking (cost-benefit analysis).

Supporting feature:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
1	V M N I S	

Elimination of conventional transformation activities - Reduction of work load due to the conventional approach of: a) constructing different lifecycle phase products (e.g., requirements specification, design specification, code) through consecutive transformations, and b) verifying that the transformations are done correctly.

Supporting features:

4) Offers a single, unified system representation at all levels of development.
5) Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
5	V M N I S	

Project status control - The activity of evaluating the status (e.g., nearness to completion, incomplete parts, quality, parts to be changed, etc.) of a system and deciding on further action.

Supporting features:

18) Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).
19) Notationally specifies the boundaries of lifecycle phases.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
18	V M N I S	
19	V M N I S	

<u>Task specification support</u> - Means for specifying and managing allocation of tasks among components of a human-computer system.

<u>Supporting features</u>:

7) Supports directly the separation of human-computer dialogue and computation.

16) Represents "<u>who</u>" performs the activities along with "<u>what</u>" activities are performed, and "<u>how</u>" they are performed.

20) Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
7	V M N I S	
16	V M N I S	
20	V M N I S	

Allowing various developmental strategies - Providing the project management with the flexibility of choosing, rather than enforcing, a developmental strategy that best suits the developmental environment.

Supporting features:

8) Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development.

15) Supports non-procedural as well as procedural representation.

17) Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
8	V M N I S	
15	V M N I S	
17	V M N I S	

B) PART2: MISCELLANEOUS QUESTIONS ON PROJECT MANAGEMENT

1. Please suggest (any) other features, which SUPERMAN lacks (i.e., not listed on pages 138 through 140), to further support project management.

2. Does SUPERMAN permit standard management techniques of estimation, in process checking, and control to be applied?

   ____yes ____no ____not sure

   Comments:

3. Does SUPERMAN aid or hinder team work?

   ____aids ____hinders ____not sure

   Comments:

4. If you were managing a project, would you prefer having a complete set of management procedures or prefer being flexible in defining your own management policies and techniques?

_____complete set _____prefer to be flexible

Comments:

5. Does SUPERMAN allow or support configuration management (e.g., organization, tracking, and maintenance of emerging work products including control and releases of multiple versions)?

_____ allows _____ supports _____ both _____ none

Comments:

6. In the following cases, in using SUPERMAN, do you have difficulties in coping with specification changes (e.g., difficulties in knowing what was decided and keeping track of current and replaced specifications, design decisions, etc.)

   (a) working as an individual

   _____ yes _____ no

   Comments:

   (b) working in a team

   _____ yes _____ no

   Comments:

7. A supervisory structure serves as a graphical directory for a system's documentation. Each function symbol in SUPERMAN has the following items of documentation:
   - list of requirements (functional, performance, etc.)
   - design decisions
   - progress report
   - notes concerning the development (distributed to concerned parties via mail)
   - unresolved problems
   - graphical data structure representations.

   Can you point to any problems in using the above organizational style for documentation?

8. Below please circle the usefulness of the listed potential documentation.

RATING: V (very useful), M (moderately useful), N (not useful), I (interfering), or S (strongly interfering).

Documentation	Circle rating	Comments
Development plan	V M N I S	
Priority specification	V M N I S	
Resources allocated	V M N I S	
Scheduling	V M N I S	
Re-evaluation points	V M N I S	
Personnel allocation	V M N I S	
Developer's notes	V M N I S	
Project history	V M N I S	
Test plans	V M N I S	
Test procedures	V M N I S	
Test results	V M N I S	

9. What other documentation can you suggest?

PART 3

GENERALITY OF APPLICATION

This part addresses the following subtree of Figure A.2.

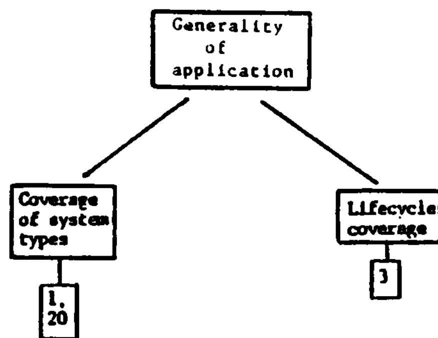

Figure A.5. Evaluation criteria and related SUPERMAN support features for generality of application.

In the following, you will evaluate the above two paths.

A) PART3: EVALUATION OF SUPERMAN FEATURES FOR GENERALITY OF APPLICATION

Coverage of system types - Types of systems that can be developed using SUPERMAN.

Supporting features:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

20) Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
1	V M N I S	
20	V M N I S	

<u>Lifecycle coverage</u> - The lifecycle phases (e.g., analysis, requirements specification, design, coding, testing, evolution) in which SUPERMAN can be used.

Supporting Feature:

3) Is used throughout the system lifecycle.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
3	V M N I S	

B) PART3: MISCELLANEOUS QUESTIONS ON GENERALITY OF APPLICATION

1. Please suggest (any) other features, which SUPERMAN lacks (i.e., not listed on pages 138 through 140), to further support generality of application.

2. Which of the following can be developed using SUPERMAN?

_____ embedded systems/process control/device control

_____ scientific/engineering systems

_____ systems programming

_____ software tools

_____ data processing/database systems

_____ expert systems/artificial intelligence

Comments:

3. Is SUPERMAN appropriate for

_____ small systems (< 2000 lines of code)?

_____ medium systems (2000 - 10,000 lines of code)?

_____ larger systems?

Comments:

(Questions 3 and 4 are borrowed from PORCM82.)

4. Does SUPERMAN allow data encapsulation (e.g., the concept of hiding information about the implementation of a data type and providing a set of implementation independent functions for use of a data type)?

   _____ yes _____no _____have difficulties

   Comments:

5. In developing a system, which of the following suits you best?

   _____ data structures first, function structure next

   _____ function structure first, data structures next

   _____ combined development of both

   Comments:

6. For the following lifecycle phases please indicate the level of help you get from SUPERMAN.

RATING: V (very satisfactory), M (moderately satisfactory), N (not satisfactory), I (interfering), or S (strongly interfering).

Lifecycle phase	Circle rating	Comments
Analysis	V M N I S	
Requirements Specification	V M N I S	
Design	V M N I S	
Coding	V M N I S	
Testing	V M N I S	
Maintenance	V M N I S	
Operation	V M N I S	

7. Does a behavioral structure (e.g., a hierarchy of dialogue-computation functions bounded by dialogue and computational functions) serve your needs for specification of functional requirements?

_____ yes _____ no _____ partially

Comments:

8. In SUPERMAN, non-functional requirements (e.g., constraints) are distributed and/or specified at abstract levels in the documentation associated with the supervisory cells. Do you need to distribute the requirements as mentioned above?

_____ yes _____no _____ sometimes

Comments:

9. Which of the following policies best suit you in conceptualizing and specifying system requirements?

_____ list of requirements first, functional structure next

_____ functional structure first, list of requirements next

_____ simultaneous construction of both on need basis

Other:____

10. Do you have difficulties in mapping the lists of requirements into supervised flow diagrams?

_____ yes _____no

Comments:

11. Are there requirements specification activities which would increase your productivity if supported by SUPERMAN?

_____ yes _____no

Comments:

12. Design decisions, like other documentation in SUPERMAN, are associated with the supervisory cells at abstract levels. Do you have difficulties in managing design decisions?

    _____ yes _____ no

    Comments:

13. Do you have difficulties in expanding the behavioral structure with dialogue and computational structures?

    _____ yes _____ no

    Comments:

14. Do you have difficulties in manual translation of supervised flow diagrams (SFD) to code?

_____ yes _____ no

Comments:

15. Do you translate SFDs into code with one to one translation or depart from SFD representation?

_____ one_to_one _____ depart from SFDs

Comments:

16. What additions to the specification technique would make coding easier?

17. Does the choice of a target language (e.g., FORTRAN, PASCAL) affect the ease of using SFD representation?

_____ yes _____no

Comments:

18. Currently, internal code blocks (ICBs) contain no control statements (i.e., decision, iteration). Some users have the need to include these in ICBs because once they get down to a manageable level of complexity, they prefer textual control to graphical control. Allowing control flow in ICBs results in the following potential disadvantages:

- inconsistency in representation of control flow,
- ICBs of large size,
- loss of control flow visibility (important especially during maintenance).

In spite of these disadvantages, would you prefer to have control flow in ICBs?

_____ yes _____ no _____ don't care

Comments:

19. Verification of notational rules (e.g., a computational function can supervise only computational functions) can be integrated into the graphical programming language (GPL) editor or compiler. How would you rate the usefulness of such verification?

_____ very useful

_____ moderately useful

_____ is not worth the effort

Comments:

20. In GPL, a data library (e.g., a data dictionary) is an alternative to using internal code blocks (ICB) for data definitions. A data library stores the definitions of data items which appear on a supervisory structure and is accessed via data selection on the screen. (Information in a data library is used by the compiler to create the associated code.) Following are some types of information a data library can have:

- Information on simple variables (e.g., name, type, dimension, range, format, common block, source module, narrative description, etc).
- Information on data structures (combination of graphical structures and textual descriptions).
- Information on files.

a) Please indicate the usefulness of a data library in SUPERMAN.

_____ very useful

_____ moderately useful

_____ not necessary

Comments:

b) Can you suggest other types of information for a data library?

Suggestions:

c) For data definitions, would you prefer a data library or using internal code blocks.

_____ data library _____ ICB _____ both

Comments:

21. Testing may be described as a series of controlled experiments to provide evidence that a program behaves properly. In general, testing is performed on the following:

   - procedural requirements (functions and control flow);
   - broad class of anticipated inputs;
   - non-procedural requirements (e.g., equipment, performance).

   In SUPERMAN, testing of the first and the second items above is performed during development (e.g., by inspection or by using the Behavioral Demonstrator). Testing of the third item can most effectively be performed during the operation of a system function. Testing may require generation of test cases (e.g., operational conditions to test human-factors, list of input data and expected output, etc.). Some possible tools are an interactive test case generator and an interactive test case executor (e.g., retrieves the test cases, executes, reports results).

   Can you suggest other tools for the testing activity?

   _____ yes _____no

   Suggestions:

PART 4

METHODOLOGY USER'S EFFICIENCY

This part addresses the following subtree of Figure A.2.

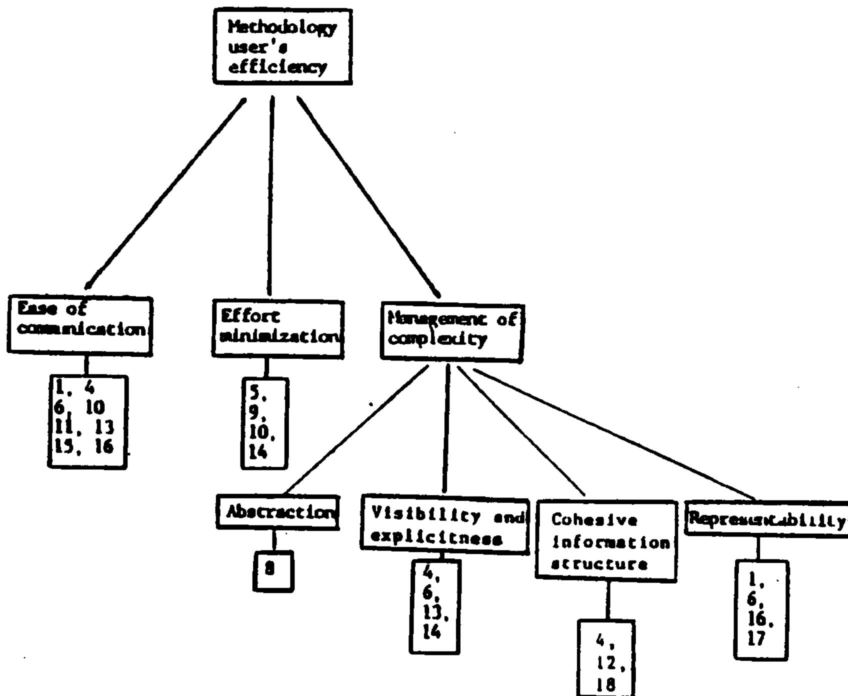

Figure A.6. Evaluation criteria and related SUPERMAN support features for user efficiency.

In the following, you will evaluate the paths of the above figure in left to right order.

A) PART4: EVALUATION OF SUPERMAN FEATURES IN SUPPORTING SUPERMAN USER'S EFFICIENCY

Ease of communication - Ease of exchanging information (among people) and understanding the meaning of the exchanged information.

Supporting features:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).
4) Offers a single, unified system representation at all levels of development.
6) Embodies both data flow and control flow in a single representation.
10) Produces executable requirements specifications which hide the technical aspects of design.
11) Produces a behavioral model of the system that can be demonstrated ("simulated") before any code is written.
13) Represents programs in a Graphical Programming Language (GPL).
15) Supports non-procedural as well as procedural representation.
16) Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.

In the following, please state the effectiveness of the above features in supporting ease of communication.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
1	V M N I S	
4	V M N I S	
6	V M N I S	
10	V M N I S	
11	V M N I S	
13	V M N I S	
15	V M N I S	
16	V M N I S	

Effort minimization - Reducing cognitive effort (e.g., understanding, conceptualizing) and physical (e.g., manual) effort for developing a system.

Supporting Features:

5) Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.
9) Is rich with supporting automated tools.
10) Produces executable requirements specifications which hide the technical aspects of design.
14) Reduces textual coding to non-control statements of conventional textual languages.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
5	V M N I S	
9	V M N I S	
10	V M N I S	
14	V M N I S	

Management of complexity - The act of comprehending and manipulating the relationships among the entities (e.g., system parts, constraints, developers, etc.) of a system development activity.

Support for management of complexity will be evaluated in the following subcategories: abstraction, visibility and explicitness, cohesive functional structure, representability. (Please refer to Figure A.2 and see that management of complexity, in addition to supporting the methodology user's efficiency, also supports the reliability of target system. In your evaluation of the following features, please consider all of these criteria.)

(i) Abstraction - Hiding irrelevant details.

Supporting feature:

8) Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
8	V M N I S	

(ii) Visibility and explicitness - The degree of clearness of system parts and their interrelationships.

Supporting features:

4) Offers a single, unified system representation at all levels of development.
6) Embodies both data flow and control flow in a single representation.
13) Represents programs in a Graphical Programming Language (GPL).
14) Reduces textual coding to non-control statements of conventional textual languages.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
6	V M N I S	
13	V M N I S	
14	V M N I S	

(iii) Cohesive information structure - An information structure which causes the parts of a system to be clearly interrelated.

Supporting features:

4) Offers a single, unified system representation at all levels of development.
12) Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)
18) Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).

Earlier you evaluated the above features in supporting project management. Here, please evaluate these features in supporting management of complexity via supporting cohesive information structure.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
12	V M N I S	
18	V M N I S	

(iv) Representability - The ability to specify notationally the development related entities (e.g., concepts, objects, operations) and their interrelationships.

Supporting features:

1) Enables modelling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).
6) Embodies both data flow and control flow in a single representation.
16) Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.
17) Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
1	V M N I S	
6	V M N I S	
16	V M N I S	
17	V M N I S	

B) PART4: MISCELLANEOUS QUESTIONS ON SUPERMAN USER'S EFFICIENCY

1. Please suggest (any) other features, which SUPERMAN lacks (e.g., not listed on pages 138 through 140), to further support SUPERMAN user's efficiency.

2. Is SUPERMAN easy to use?

_____ very easy to use

_____ easy to use

_____ moderately difficult

_____ difficult

Comments:

3. Do you have difficulties in understanding and using work products (e.g., parts of a supervisory structure) produced by someone else?

_____ yes _____ no

Comments:

4. Do supervised flow diagrams make communication among people difficult or easy?

   _____ difficult   _____easy

   Comments:

5. The following quote is taken from SUPERMAN paper:

   > While using SUPERMAN, in the analysis and design phases, the methodology user has the option of specifying the SFDs at any level of procedural detail (e.g., ignoring iteration, using abstract names for data objects, etc.). A fully-detailed SFD is required only for the compiler. In early phases of the lifecycle, while the system developer can do functional abstraction (i.e., functional decomposition through levels of the hierarchy), the developer can also partially or fully specify the procedural relationships at a given level in the hierarchy. Hence, the detailed specification process is responsive to communication needs of the developers.

   Do you have difficulties in using the approach above?

   _____ yes   _____ no

   Comments:

6. As discussed in question 5, a supervisory structure can be represented at several levels of detail. For example, the level of minimum detail is a structure which has only functions but no data and control flow (e.g., HIPO-like diagrams). Likewise, the level of maximum detail is a structure which can pass through the GPL compiler (i.e., the graphical code).

   Would you like to have the automated support to construct and/or review a structure with several versions of varying levels of detail?

   _____ yes _____ no

   Comments:

7. Some Dialogue Management System tools built (or being built) around the supervisory structure are: GPL editor and compiler, Behavioral Demonstrator, storage and retrieval of various documentation (e.g., requirements lists, design decisions, progress reports, data structures, project development plan, etc.)

a. Other than the tools mentioned above and earlier in this survey, what other tools would you suggest?

b. If you could obtain any one automated software tool to assist you in your use of SUPERMAN, what would this tool do?

8. Do you prefer graphical or textual representations for functional specifications?

   _____ graphical _____ textual _____ combined

   Comments:

9. Are there control constructs or notational aspects (e.g., decision, iteration, recursion, data flow, function symbols, semantics of symbols, etc.) in SUPERMAN that are unclear or confusing?

   _____ yes _____ no

   Comments:

10. Notation of SUPERMAN can be extended, for instance, to support project management (e.g., tagging functions with priority of development), to support human factors analysis (e.g., attaching device indicators to dialogue functions).

    Can you suggest similar notational extensions?

    _____ yes _____ no

    Suggestions:

PART 5

QUALITY OF TARGET SYSTEM

This part addresses the following subtree of Figure A.2.

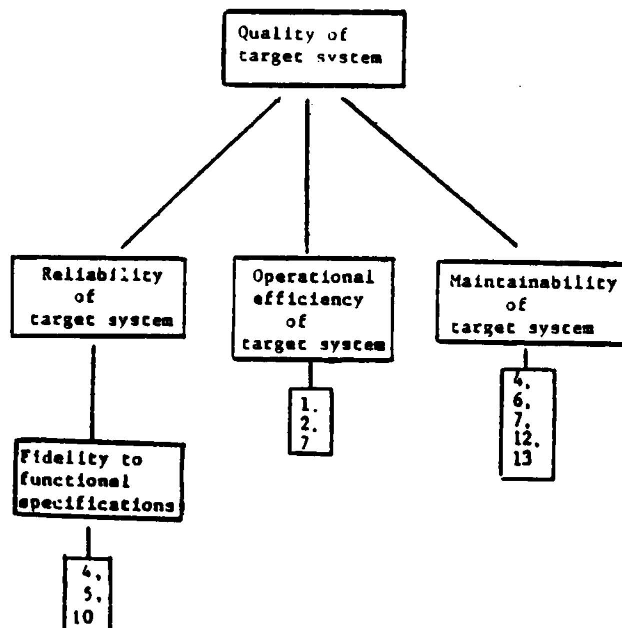

Figure A.7. Evaluation criteria and related SUPERMAN support features for quality of target system.

In the following, you will evaluate the paths of the above figure in left to right order.

A) PART5: EVALUATION OF SUPERMAN FEATURES IN SUPPORTING QUALITY OF TARGET SYSTEM

Fidelity to functional specifications - The extent a finished product (e.g., code) performs the functions as specified in earlier lifecycle phases.

Supporting Features:

4) Offers a single, unified system representation at all levels of development.
5) Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.
10) Produces executable requirements specifications which hide the technical aspects of design.

RATING: V (very effective), M (moderately effective), N (not effective), I (interfering), or S (strongly interfering).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
5	V M N I S	
10	V M N I S	

Operational efficiency of target system - Speed of operation of a human-computer system in achieving system goals.

Supporting features:

1) Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).
2) Integrates human-factor inputs from the beginning of a system design.
7) Supports directly the separation of human-computer dialogue and computation.

Feature	Circle rating	Comments/problems/difficulties
1	V M N I S	
2	V M N I S	
7	V M N I S	

<u>Maintainability of target system</u> - The ease of making changes to a system while preserving the system's integrity.

Supporting Features:

4) Offers a single, unified system representation at all levels of development.

6) Embodies both data flow and control flow in a single representation.

7) Supports directly the separation of human-computer dialogue and computation.

12) Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

13) Represents programs in a Graphical Programming Language (GPL).

Feature	Circle rating	Comments/problems/difficulties
4	V M N I S	
6	V M N I S	
7	V M N I S	
12	V M N I S	
13	V M N I S	

B) PART5: MISCELLANEOUS QUESTIONS ON QUALITY OF TARGET SYSTEM

1. Please suggest (any) other features, which SUPERMAN lacks (i.e., not listed on pages 138 through 140), to further support quality of target system.

2. One verification activity, in the conventional lifecycle, is making sure that transformation of work products of different phases is done correctly (e.g., converting functional specification to design representation, design representation to code). Having a unified system representation, SUPERMAN eliminates these transformations and the associated verification activities.
   Can you suggest verification activities SUPERMAN should provide help with?

   _____ yes _____ no

   Suggestions:

3. Validation, in the conventional software lifecycle, is the process of determining: (a) that a system (i.e., the final product) performs those functions described in functional specification (an early phase product), (b) and that it performs them correctly.

Please evaluate the following statement:

In SUPERMAN, due to elimination of work product transformations, the process of validating that the software performs the functions recorded in specifications is eliminated. This is because functional specification is the software (i.e., the behavioral structure - what you specify is what you get). A finished product will not correctly perform the intent of the behavioral structure only if the dialogue and computation functions appended in design phase (and finished in coding phase) do not realize this intent.

a. What are your objections to the statement above?

b. Do you have difficulties in understanding the intent of the tasks assigned to you?

_____ yes _____no

Comments:

c. Can you describe situations which lead to misunderstanding of this intent?

_____ yes _____ no

Comments:

4. Most methodologies begin the system development with specifying what the system should do (i.e., requirements specification). Then, this work product is converted into how it should be done (i.e., design). Finally design is transformed into code. In SUPERMAN, the above approach (i.e., all of what first, all of how next) is supported at the global level. However, each of these global work products (i.e., the behavioral structure, and the design structures) are internally developed in terms of alternating what and how sequences (i.e., hierarchy of supervised flow diagrams).

   In view of the above discussion, please indicate whether you agree or do not agree (and discuss why you do not agree) with the following statement:

   Alternating what-how development enforces the developer to continuously evaluate his/her understanding of the target system (i.e., conceptualization) as well as to evaluate the operational characteristics of the target system (e.g., functions, human engineered sequencing, compatibility of a part with other parts, correctness, etc.) Another advantage of this approach is capturing this information before it is lost.

   _____ agree _____ do not agree _____ not sure

   Comments:

5. How would you rate the maintainability of a system developed using SUPERMAN?

   _____ very easy to maintain

   _____ moderately easy to maintain

   _____ difficult to maintain

   Comments:

6. Please list the difficulties that you have in maintaining a supervisory structure.

7. Do you think the functions you produce using SUPERMAN are reusable?

_____ yes _____no

Comments:

8. What methodology characteristics reduce reusability?

PART 6

OVERALL STRENGTHS AND WEAKNESSES OF SUPERMAN

1. Please outline the overall strengths of SUPERMAN in the following two categories.

   a) In comparison with other methodologies

   b) With regard to your own experience

2. Please outline the overall weaknesses of SUPERMAN.

   a) In comparison with other methodologies

   b) With regard to your own experience

Appendix B

A DETAILED VIEW OF THE SURVEY RESULTS

B.1 EASE OF LEARNING

Table 2 illustrates the ratings for ease of learning. The comments on the features follow the table.

TABLE 2

FEATURE	V	M	N	I	S
4. Unified representation	2	3	1		

FEATURE 4: Offers a single, unified system representation at all levels of development.

SUPPORTED CRITERION: Consistency.

AVERAGE RATING: Moderately effective.

COMMENTS: The SUPERMAN users all agreed that the methodology is easy to learn. The majority of the users attributed this not to the consistency of representation, but to: "small vocabulary of symbols combined in standard ways", "clear and simple concepts", "hierarchical structure", "well integration of the behavioral structure with its extension", "use

of block structured language constructs". Some comments are worth mentioning: "Very nice - much easier than any programming language I had learned before", "Learn a handful of symbols and rules and you know the system. Do a handful of SFD's and you are comfortable with it."

A constructive suggestion was: "Many people in learning SUPERMAN experience a _when to divide, what to subdivide syndrome_. Perhaps guidelines should be included."

B.2 PROJECT MANAGEMENT

Table 3 illustrates the ratings on project management. The comments on the features follow the table.

TABLE 3

FEATURE	V	M	N	I	S
1. Function allocation	1	2	1		
4. Unified representation	6	3	2		
5. Conceptual phase transitions	4	1			
7. Separation of dialogue and computation	6				
8. Top-down and/or bottom-up development	1	3			
12. Cohesive functional structure	3	2	1		
15. Procedural & non-procedural representations			2		
16. "Who", "What", "How"	2	3			
17. OSD & SFD	1	1			
18. Structured information directory	5	4			
19. Notationally specified phase boundaries ...	1	1	1		
20. Applicable to any procedural system	2	3			

FEATURE 1: Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

SUPPORTED CRITERION: Feasibility and cost-benefit analysis.

AVERAGE RATING: Moderately effective.

COMMENTS: The majority of the users stated that they did not have experience with the function allocation activity. Hence, the evaluation was mostly intuitive. A comment was:

"SUPERMAN allows for modeling of human-computer systems; there are no provisions for analysis of the model." Also, a user suggested the provision of " . . . software metrics for cost-benefit analysis of certain design decisions".

FEATURE 4: Offers a single, unified system representation at all levels of development.
SUPPORTED CRITERIA: Cohesive information structure, elimination of conventional transformation activities.
AVERAGE RATING: Very effective.
COMMENTS: Most users agreed that this is a very helpful feature. A large overview diagram, to help see the relationships among far apart functions, was suggested.

FEATURE 5: Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.
SUPPORTED CRITERION: Elimination of conventional transformation activities.
AVERAGE RATING: Very effective.
COMMENTS: By allowing for a phased approach, this feature enriches feature 4 (discussed above). Most users stated that these two (4 and 5) are among the strongest features of SUPERMAN.

FEATURE 7: Supports directly the separation of human-computer dialogue and computation.

SUPPORTED CRITERION: Task specification support.

AVERAGE RATING: Very effective.

COMMENTS: None.

FEATURE 8: Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development.

SUPPORTED CRITERION: Abstraction.

AVERAGE RATING: Moderately effective.

COMMENTS: A user stated the need for a support tool to help associate the high-level and low-level system components.

FEATURE 12: Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

SUPPORTED CRITERION: Cohesive information structure.

AVERAGE RATING: Moderately effective.

COMMENTS: Most users stated that they have not used the supervisory structure as a directory for development-related information. Consequently, the evaluation of this feature was based on intuition.

FEATURE 15: Supports non-procedural as well as procedural representation.

SUPPORTED CRITERION: Various developmental strategies.

AVERAGE RATING: Not effective.

COMMENTS: In the above feature, the intended meaning of the term non-procedural is: hiding the procedural relationships of a procedural system (e.g., DFD like representations). However, the users evaluated SUPERMAN with regard to its use in rule-based language environments.

FEATURE 16: Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.

SUPPORTED CRITERION: Task specification.

AVERAGE RATING: Moderately effective.

COMMENTS: Some users stated that they have not used this feature. A user criticized this feature for that: "who" and "what" are represented very clearly; "how" is a little less clear. It should be noted that, in SUPERMAN, "how" is defined by the representation of a supervisor's operation.

FEATURE 17: Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.
SUPPORTED CRITERION: Various developmental strategies.
AVERAGE RATING: Moderately effective.
COMMENTS: Most of the users were not familiar with the OSD form of an SFD. Hence, this feature was not evaluated by the majority. One user stated that this is a "good integration of user's environment with underlying software support."

FEATURE 18: Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).
SUPPORTED CRITERION: Cohesive information structure, project status control
AVERAGE RATING: Very effective.
COMMENTS: A user expressed the shared opinion of the majority as: "This is a <u>very</u>, <u>very</u> effective feature." Most users stated the need for mechanisms and automated support for construction and management of this information. Some needs were: traceability from non-functional specifications (e.g., design decisions) to functions of the supervisory structure

and vice versa, means for managing status information (e.g., which modules are finished, which modules are incomplete, expected due dates), and the need to know who is working on which modules.

FEATURE 19: Notationally specifies the boundaries of life-cycle phases.

SUPPORTED CRITERION: Project status control.

AVERAGE RATING: Moderately effective.

COMMENTS: Some users stated that they never used this feature. A user's statement was: "only the boundary between requirements specification and design is identified."

FEATURE 20: Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

SUPPORTED CRITERION: Task specification support.

AVERAGE RATING: Moderately effective.

COMMENTS: Most users were not aware of SUPERMAN's notation for modeling concurrent operations. Hence, the rating of this feature was negatively affected. A current weakness of the notation, as mentioned by a user, is the implicitness of

asynchronous event handling. Currently, SUPERMAN expects a user to implement such operations by means of host system services.

B.2.1 Miscellaneous Comments on Project Management

Documentation

Table 4 illustrates the users' ratings on the desirability of listed documentation.

TABLE 4

Documentation	V	M	N	I	S
Development plan	3	2	1		
Priority specification	2	4			
Resources allocated	1	5			
Scheduling	1	4			1
Re-evaluation points	1	2	2		
Personnel allocation	2	2	1		
Developer's notes	5	1			
Project history	1	4	1		
Test plans	3	2	1		
Test procedures	4	1	1		
Test results	4		1	1	

Users suggested other documentation be included as well, namely software metrics to predict maintainability, and some means of saving the reasonings which lead to design decisions.

Team work

All users agreed that SUPERMAN aids (rather than hinders) team work. Some comments were: "Very helpful in breaking up a task", "SFDs provide a good communication tool for all team members", "Within a team, walk-throughs of SFD's

rather than code were very successful as a means to evaluate alternative designs."

Major difficulties surfaced with regard to cooperation among the teams. Yet at least one user attributed this "not to a weakness of SUPERMAN, but to a lack of knowledge of each team of the subject area of the other team." Related difficulties had to do with maintaining inter-team consistency (e.g., inconsistent data definitions) and determining what teams are affected by changes (e.g., "who needs to know this decision?", "what modules are affected?")

Configuration management

The users had different views on whether SUPERMAN supports or allows configuration management. One user stated that configuration management is supported, another said it is only allowed for, while a third said that SUPERMAN neither allows nor supports this activity. As stated by a user "each drawing of a function structure can be subjected to configuration management."

B.3 GENERALITY OF APPLICATION

Table 5 illustrates the ratings on generality of application. The comments on the features follow the table.

TABLE 5

FEATURE	V	M	N	I	S
1. Function allocation	1	3	1		
3. Lifecycle coverage	4	1			
20. Applicable to any procedural system	1	3			

FEATURE 1: Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

SUPPORTED CRITERION: Coverage of system types.

AVERAGE RATING: Moderately effective.

COMMENTS: The majority of the users stated that they did not have experience with the function allocation activity. Hence, the evaluation was mostly intuitive.

FEATURE 3: Is used throughout the system lifecycle.

SUPPORTED CRITERION: Lifecycle coverage.

Average Rating: Very effective.

COMMENTS: (A detailed evaluation with regard to lifecycle phases is presented in the following sub-section)

FEATURE 20: Can be used in development of any procedural system (sequential or concurrent) including systems for developing other systems (e.g., automated management procedures).

SUPPORTED CRITERION: Coverage of system types.

AVERAGE RATING: Moderately effective.

COMMENTS: A lack of knowledge in SUPERMAN's notation for concurrent operations affected the evaluation negatively. Some users stated that SUPERMAN is not effective when used with rule-based languages.

B.3.1 Miscellaneous Comments on Generality of Application Lifecycle Phases

Table 6 illustrates the users' ratings with regard to lifecycle phases.

TABLE 6

Lifecycle phase	V	M	N	I	S
Analysis	1	3	1		
Requirements Specification	2	1	3		
Design	4	2			
Coding	3	2			1
Testing	1		3		1
Maintenance	2		1	1	
Operation			2		

With regard to analysis and requirements specification activities, some users had conflicting views: while one user rated SUPERMAN as "very effective", another user criticized it as "not effective", adding that, for these activities, SUPERMAN "relies on techniques from other methodologies." On the other hand, in response to a later survey question, most users (including these just quoted) agreed that

Alternating what-how development enforces the developer to continuously evaluate his/her understanding of the target system (i.e., conceptualization) as well as to evaluate the operational characteristics of the target system (e.g., functions, human engineered sequencing, compatibility of a part with other parts, correctness, etc.)

However, it should be noted that, while this statement describes a goal-directed analysis activity, the product of this activity is the <u>specification</u> of the <u>required</u> operation.

For design activity, a majority of users rated SUPERMAN as "very effective". Some comments were: "SFD's are the basis and are excellent", "Awesome! Breath-takingly powerful." However, some users had problems with data encapsulation. Although one user stated that SUPERMAN is "excellent for this purpose", another user's statement was: "global nature of the name space makes writing of transparent data managers impossible." In response to this, it is important to note that data encapsulation is a design issue, and SUPERMAN hides this from the non-technical observer of the behavioral structure. If desired, the behavioral structure can be constructed with data encapsulation. This is also an example of conceptual phase transition.

For coding activities, the rating of the majority was "very effective" due to "identity of representation". However, informality of data representations affected both coding and testing activities. In response to a survey question, users agreed that a data dictionary, accessed via data selection on the screen, and used for automatic code generation, would be very useful. For a data dictionary, users

agreed on the following proposed inclusions: information on simple variables (e.g., name, type, dimension, range, format, common block, source module, narrative description), information on data structures (combination of graphical structures and textual descriptions), and information on files. Additionally, users suggested the following information: user defined data types, abstract data types, data instances, scalar list of allowable values.

For testing activities, users stated the following needs: "asynchronous testing", "interrupt testing", "a tool which turns the results of testing into system modifications", and "a tool to report all conditions that must hold in order for execution to be at any particular point in the supervisory structure."

System types

Ratings with regard to various system types are illustrated below. On each line, the number in parenthesis indicates the number of users who thought that SUPERMAN could be used in developing the associated system type.

(1) embedded systems/process control/device control
(5) scientific/engineering systems
(3) systems programming
(5) software tools
(6) data processing/database systems
(1) expert systems/artificial intelligence

Some users rated SUPERMAN only for areas in which they were experienced (e.g., scientific/engineering systems, software tools, data processing/database systems). The lack of knowledge of SUPERMAN's notation for concurrent operations affected some users' rating with regard to embedded systems, and systems programming. Also, SUPERMAN's notation was criticized for not including device control, asynchronous events, and interrupt handling.

With regard to system size, most of the users agreed that there are "No restrictions as to size once the automated tools are available."

B.4 SUPERMAN USER'S EFFICIENCY

Table 7 illustrates the ratings on SUPERMAN user's efficiency. The comments on the features follow the table.

TABLE 7

FEATURE	V	M	N	I	S
1. Function allocation	4	4	1		
4. Unified representation	11	16			
5. Conceptual phase transitions	3	2			
6. Data flow & control flow	7	9	1		
8. Top-down and/or bottom-up development	2	4			
9. Rich with automated tools	3	2			
10. Executable specifications	6	4			
11. Behavior demonstration	2	1		1	
12. Cohesive functional structure	3	3			
13. GPL	8	4			
14. No textual coding of control	7	2	1		1
15. Procedural & non-procedural representations		2			
16. "Who", "What", "How"	4	4	2		
17. OSD & SFD	1	1			
18. Structured information directory	3	2			

FEATURE 1: Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

SUPPORTED CRITERIA: Communication, Representability.

AVERAGE RATING: Moderately effective.

COMMENTS: The majority of the users stated that they did not have experience with the function allocation activity. Hence, the evaluation was mostly intuitive.

FEATURE 4: Offers a single, unified system representation at all levels of development.

SUPPORTED CRITERIA: Communication, Management of complexity.

AVERAGE RATING: Moderately effective.

COMMENTS: A large overview diagram which illustrates the relationships among system components was suggested.

FEATURE 5: Offers conceptual transition between the lifecycle phases and does not require conversion from one representation to another.

SUPPORTED CRITERION: Effort minimization.

AVERAGE RATING: Very effective.

COMMENTS: A majority of users stated the need for a formal representation of data structures, because informality has resulted in inconsistent definitions, as well as difficulties in communication and in conceptualizing data flow.

FEATURE 6: Embodies both data flow and control flow in a single representation.

SUPPORTED CRITERIA: Communication, Management of complexity.

AVERAGE RATING: Moderately effective.

COMMENTS: (See data structure problems discussed above.)

FEATURE 8: Features top-down levels of abstraction, but allows any combination of top-down and bottom-up development. development.

SUPPORTED CRITERION: Abstraction.

AVERAGE RATING: Moderately effective.

COMMENTS: To some users, this feature was very effective, though one evaluator stated that "the effectiveness of this feature depends on the methodology user." For example, while a user (experienced in analysis and design) objected to bottom-up development, an implementation-oriented user had difficulties with a strict top-down approach.

FEATURE 9: Is rich with supporting automated tools.

SUPPORT CRITERION: Effort minimization.

AVERAGE RATING: Very effective.

COMMENTS: The above rating was made with the reservation that the tools are easy to use.

FEATURE 10: Produces executable requirements specifications which hide the technical aspects of design.

SUPPORTED CRITERION: Communication, Effort minimization.

AVERAGE RATING: Very effective.

COMMENTS: One user argued that behavioral structure is a design representation and does not provide much help for requirements specification. But as mentioned earlier, behavioral structure is the specification of the required operation, and includes non-functional requirements as well as development-related information.

FEATURE 11: Produces a behavioral model of the system that can be demonstrated ("simulated") before any code is written.

SUPPORTED CRITERION: Communication.

AVERAGE RATING: Moderately effective.

COMMENTS: Users agreed that a behavioral model reduces ambiguities and and helps communication.

FEATURE 12: Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)

SUPPORTED CRITERION: Cohesive information structure.

AVERAGE RATING: Very effective.

COMMENTS: None.

FEATURE 13: Represents programs in a Graphical Programming Language (GPL).

SUPPORTED CRITERION: Communication, Management of complexity.

AVERAGE RATING: Very effective.

COMMENTS: Users preferred GPL over textual representations. Some comments were: "So much more descriptive than code, I don't feel the need to look at code again", "simple, unambiguous, graphically clear representation of code", "better than any other approach for communication", "understandable by non-computer personnel".

However, difficulties were experienced with regard to some of the notation. Users found the current "recursion" and "iteration" constructs difficult to use. For producing optimized code, some users needed the control constructs of low-level languages (e.g., JUMP), and it was generally agreed that the ICBs should not be restricted to non-control statements. Also, for data declarations, in addition to the data dictionary discussed earlier, the users stated that internal code blocks should be available as an option.

FEATURE 14: Reduces textual coding to non-control statements of conventional textual languages.

SUPPORTED CRITERION: Effort minimization, Visibility and explicitness.

AVERAGE RATING: Very effective.

COMMENTS: Some users stated that this feature aids traceability and communication.

FEATURE 15: Supports non-procedural as well as procedural representation.

SUPPORTED CRITERION: Communication.

AVERAGE RATING: Moderately effective.

COMMENTS: Some users complained that SUPERMAN is not effective when used with rule-based languages.

FEATURE 16: Represents "who" performs the activities along with "what" activities are performed, and "how" they are performed.

SUPPORTED CRITERIA: Communication, Representability.

AVERAGE RATING: Moderately effective.

COMMENTS: While one user said that this feature is "absolutely required for communication", another stated that it is "not important for software representation". However, in addition to representing an human-computer system operation, this feature is also used for representing the operation of concurrent software functions.

FEATURE 17: Supports Operational Sequence Diagram (OSD) approach as well as software approach to system modelling and provides straightforward transformation from OSD representation to software representation and vice versa.
SUPPORTED CRITERION: Representability.
AVERAGE RATING: Moderately effective.
COMMENTS: While none of the users had had experience with the OSD, an evaluator observed that "SUPERMAN is the only real methodology that does this."

FEATURE 18: Uses the target system functional representation as a structured directory for development related information (e.g., design decisions, operational constraints, management information, etc.).
SUPPORTED CRITERION: Cohesive information structure.
AVERAGE RATING: Very effective.
COMMENTS: A user stated his opinion as: "This is a very important feature in any non-toy methodology". On the other hand, some users were not aware of the feature and the context of its use.

B.4.1 Miscellaneous Comments on SUPERMAN User's Efficiency

Ease of use

All users agreed that SUPERMAN is easy to use, and provided that consistency is maintained, SFDs are readily understood and easily communicated.

Automated support

Users stated that the GPL editor-compiler is a most desirable tool. Some activities which need automated support are: verification of notational rules and syntax, communication (e.g., transmission of changes), configuration management, construction of SFD's with varying levels of detail.

Notational extensions

Users particularly mentioned the following notational additions: function and system version numbers, configuration paths (e.g., highlighted nodes), CPM or PERT like charts for project management, notation for metering and performance monitoring, notation for exceptional conditions.

B.5 QUALITY OF TARGET SYSTEM

Table 8 illustrates the ratings on project management. The comments on the features follow the table.

TABLE 8

FEATURE	V	M	N	I	S
1. Function allocation	2	1	1		
2. Integration of human factors inputs........	1	2	1		
4. Unified representation	7	4			
5. Conceptual phase transitions	4	3			
6. Data flow & control flow	2	4			
7. Separation of dialogue and computation	8		1		
10. Executable specifications	4	1			
12. Cohesive functional structure	2	4			
13. GPL	2	3			

FEATURE 1: Enables modeling and analysis of human-computer systems, with allocation of functions (manual versus automated functions, human versus computer management of control and data flow).

SUPPORTED CRITERION: Operational efficiency.

AVERAGE RATING: Moderately effective.

COMMENTS: Most users evaluated this feature in the context of software systems rather than in a wider context of human-computer system operations. This resulted in comments like "Irrelevant as supporting feature," or "I do not know how

this affects the efficiency of a system". Users who realized the purpose of the feature stated that they had not used "human versus computer aspects" of SUPERMAN.

FEATURE 2: Integrates human-factor inputs from the beginning of a system design.
SUPPORTED CRITERION: Operational efficiency.
AVERAGE RATING: Moderately effective.
COMMENTS: As in the case of feature 1, most users did not evaluate this feature with regard to human-computer system efficiency, but with regard to software efficiency. Also, some users seemed to disregard the effects of function sequencing on ease of using a software system. The following comment illustrates these points: "Human factors inputs come only after implementation."

FEATURE 4: Offers a single, unified system representation at all levels of development.
SUPPORTED CRITERIA: Fidelity to functional specification, Maintainability.
AVERAGE RATING: Very effective.
COMMENTS: None.

FEATURE 5: Offers conceptual transition between the lifecycle phases

SUPPORTED CRITERIA: Fidelity to functional specifications.

AVERAGE RATING: Very effective.

COMMENTS: None.

FEATURE 6: Embodies both data flow and control flow in a single representation.

SUPPORTED CRITERION: Maintainability.

AVERAGE RATING: Moderately effective.

COMMENTS: "not a maintenance issue", "does not provide for branching statements of low-level languages".

FEATURE 7: Supports directly the separation of human-computer dialogue and computation.

SUPPORTED CRITERIA: Operational efficiency, Maintainability.

AVERAGE RATING: Very effective.

COMMENTS: With regard to maintainability, all users rated this feature as "very effective". With regard to operational efficiency, the ratings were divided between "very effective" and "not effective" (i.e., some users ignored the human's efficiency.)

FEATURE 10: Produces executable requirements specifications which hide the technical aspects of design.
SUPPORTED CRITERION: Fidelity to functional specifications.
AVERAGE RATING: Very effective.
COMMENTS: None.

FEATURE 12: Naturally encourages cohesion among modules of the target system (i.e., an expansion of a supervisory function results in functions which serve only the supervisor's goal.)
SUPPORTED CRITERION: Maintainability.
AVERAGE RATING: Moderately effective.
COMMENTS: A user stated that "it is easy to put together a set of functions which are logically related (e.g., logical strength)", and SUPERMAN interferes with the construction of stronger modules. But in the author's opinion, the supervisory concept naturally encourages functional strength modules (i.e., the highest in Structured Design scale), and, if one is concerned with module strength, SUPERMAN does not interfere with the construction of stronger modules but demands the designer's attention.

FEATURE 13: Represents programs in a Graphical Programming Language (GPL).

SUPPORTED CRITERION: Maintainability.

AVERAGE RATING: Moderately effective.

COMMENTS: All complaints were on drawing and managing the diagrams manually.

The vita has been removed from the scanned document