BEHAVIORAL DEMONSTRATION : An Approach To Rapid Prototyping and Requirements Execution

by

James E. Callan III

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
COMPUTER SCIENCE AND APPLICATIONS

APPROVED:

H Rex Hartson, Chairman	Sallie M. Henry
James D. Arthur	Robert C. Williges

July 3, 1985

Blacksburg, Virginia

BEHAVIORAL DEMONSTRATION : An Approach To Rapid Prototyping and Requirements Execution

by

James E. Callan III

H. Rex Hartson, Chairman

COMPUTER SCIENCE AND APPLICATIONS

(ABSTRACT)

This thesis presents an approach to rapid prototyping called **behavioral demonstration** that allows a system to be demonstrated at **any** point during its development. This approach is based on the operational specification approach to software design and uses a new, automation based life-cycle paradigm. This work describes a tool that supports behavioral demonstration called the **behavioral demonstrator** that collects and manages information typically lost during software system design but critically needed during maintenance. The tool also supports project-personnel management and software complexity and cost estimation. The research takes place in the context of a dialogue management system and software design methodology that features the logical and physical separation of the input, processing, and output components of interactive systems.

ACKNOWLEDGEMENTS

The author would like to express the greatest appreciation for help, support, and encouragement that others have granted, making this thesis possible. First, I would like to thank my major professor, friend, and colleague Dr. H. Rex Hartson for supporting me financially through my graduate studies, allotting time in a busy schedule for short meetings that often grew too long, and tolerating abrupt interruptions during the course of my work. I would like to thank Dr. Sean Arthur for helpful suggestions, helpful philosophical chats, and use of his personal library. I would like to thank Dr. Sally Henry for impressing on me the need for more information in the maintenance phase of the life-cycle and for suggestions leading to the support of software metrics in later versions of the Dialogue Management System. I would like to thank Denise Bice, Reba Taylor, and Dr. Debby Johnson for proofreading this thesis and making numerous helpful suggestions.

DEDICATION

This thesis is dedicated to three women: Denise Bice, Reba Taylor, and Norma C. Callan, for without whose encouragement and support would not have been possible. What is a man's life without caring women who help him to forget his work so that life may go on?

TABLE OF CONTENTS

1.0 Purpose and Setting for this Research 1

1.1 Setting for this Research 1

1.2 Purpose of this Research 2

2.0 Related Literature and Motivation 4

3.0 Previous Work 20

3.1 Scenarios 21

3.2 Operational Specification 28

4.0 Attributes of a New Approach to Prototyping . . . 35

4.1 Evaluation of Current Designs 35

4.2 Future Prototyper Features 37

4.3 New Tools for the Future 39

5.0 SUPERvisory Methodology And Notation (SUPERMAN) . 46

5.1 SUPERMAN vs. Other Methodologies 46

5.2 SUPERMAN Notation 48

6.0 The Dialogue Management System (DMS) 57

7.0 The Behavioral Demonstrator (BD) 67

7.1 BD Prototype (1983) 68

7.2 BD (1985): A View of the Future 73
7.3 BD Design 89
7.3.1 Human-Interface 90
7.3.2 Architecture 108
7.3.3 Design Decisions Affecting Design 125

8.0 Conclusions 130

9.0 Future Work 132

Bibliography 133

Appendix A 139

Appendix B 144
Information Space Description 144
Requirements/Specifications 144
Project Status 145
Design Decisions 145
Critical Time Statistics 146
Notes and Comments 146
Development Reminders 147

VITA 148

1.0 PURPOSE AND SETTING FOR THIS RESEARCH

1.1 SETTING FOR THIS RESEARCH

Software system designs may be characterized by three major parts: input, processing, and output. Traditional system design methodologies have been concerned primarily with efficient and reliable construction of a correct processing part, or computational component, in a system design. Modern software applications, however, have migrated from principally computational areas, where the computational component directs flow of control, to interactive application areas, where the input part directs the flow of control. In such systems, the input and output parts, or dialogue components, of the design are paramount. Unfortunately, since traditional system design methodologies have concentrated on reliable and correct computational portions, they have grossly neglected the dialogue components that pervade modern software system design. If the all-important human-computer interface that gives personality to the dialogue components in these designs is neglected, users care little if the computational component works correctly ("If I can't use it, what good is it if it works?").

To meet the need for better human-computer interfaces, entirely new research areas in computer science have emerged. Human-computer interaction and dialogue management are two such areas that are evolving to meet this need. In the early efforts of dialogue management, research began at Viginia Tech on a Dialogue Management System (DMS) [HART84a] and a new software system design methodology (SUPERMAN) [YUNTE85] that could quickly and efficiently produce reliable computational and dialogue components in a system design. It is within the DMS environment that this research in rapid prototyping takes place.

1.2 PURPOSE OF THIS RESEARCH

The purpose of this thesis is to generalize the previous work done on rapid prototypers and provide a new life-cycle concept in which executable requirements and rapid prototyping play a major role. This work, using a new, holistic software design methodology (SUPERMAN) and a supporting design environment (DMS), presents a new design for a prototyping, project management tool (Behavioral Demonstrator) which allows users to observe system behavior at **any** stage during system specification, design, and implementation, and which encourages increased user participation in each stage of development. In particular, this work provides a means, using the facilities of DMS, by which the users of

a system can try out an interface before it has been completely implemented. Using SUPERMAN, system dialogue components may be developed independently of computational components; this work provides an integrated approach, using SUPERMAN under DMS, for independently, empirically validating these components and integrating them at any stage of development for demonstration and execution. The prototyping tool described in this thesis, the behavioral demonstrator, supports software cost estimation, automatic code generation, project-personnel management, interactive global control testing and refinement, system data flow tracking, management of design decision, project progress, time constraints, and top-down testing, refinement, and execution of system requirements and specifications.

Behavioral demonstration reflects the activities and concerns of system analysts, application experts, human factors experts, programmers, project managers, and users during development of any computer system, particularly interactive systems. Through a radical new software life-cycle concept, this work supports graceful transition from system specification to system realization and provides a new method by which systems may be maintained. Through the concept of behavioral demonstration, and its supporting tool, this thesis moves software development closer an age where humans specify, and computers program themselves.

2.0 RELATED LITERATURE AND MOTIVATION

The once-held notion that software is first created, then used, and finally retired has been replaced by the software engineering life-cycle which consists of the following developmental phases:

- requirements specification,
- design,
- implementation,
- testing, and
- system operation and maintenance.

Each of these life-cycle phases will be briefly discussed.

Requirements specification is usually a two step process involving first requirements definition, then system specification. During requirements definition designers seek to completely understand a proposed system's behavior and functionality as perceived by users. After the functionality can be completely described, system specification begins. System specification involves the formal presentation of the complete functionality, behavior, and environmental constraints that will exist in the system design. The impor-

tance of a complete specification is described as follows [WASSE77]:

- all participants in the project development process can evaluate the specifications, and the specifications can be brought to a final form which is accepted by everyone (users, analysts, etc.),
- the complete specifications lead to the complete design of the system, and
- testing of the requirements can be planned and performed systematically only if the requirements are defined.

Because inadequate or incomplete requirements analysis can lead to complete project failure, requirements analysis, considered one of the most critical phases in the life-cycle, should involve users early. One of the hardest problems to detect prior to implementation and one often encountered during requirements analysis is that a system requirements list may have internal inconsistencies which impedes workable solutions. Use of formal system specification languages such as PSL or RSL can insure unambiguous and consistent specification but not correct or complete specifications [BOEHM78]. Only when complete requirements have been agreed upon can progress toward an acceptable system design continue.

Design is the process by which complete, consistent, and correct specifications are translated and refined into a feasible solution that has realistic expectations of success

when implemented on a host system. Wasserman [WASSE80] divides design into two distinct phases - architectural design and detailed design. Architectural design is intended to define intermodule structure, important data structures, intramodule communication interfaces between modules and external databases, and intermodule communication interfaces. This phase produces the structure and framework for the underlying details of design. Detailed design involves definition of internal logic within modules and logical design of the databases. This phase refines and focuses the architectural design so that implementation may begin and assumes the complete, consistent, and correct system specification of the previous phase. In many regards this assumption is unrealistic. The design phase produces certain design decisions which impact dynamic behavior of the completed system. Faithful translation of specifications into design depends strongly upon communication among project members for success. The hopelessness of these expectations being met is anticipated by three facts which follow [BOARH85]:

1. It is extremely difficult for users to prespecify their requirements in total and final detail,
2. Graphic and narrative techniques are inadequate for communicating the dynamics of a proposed system, and
3. Miscommunication is endemic among project members.

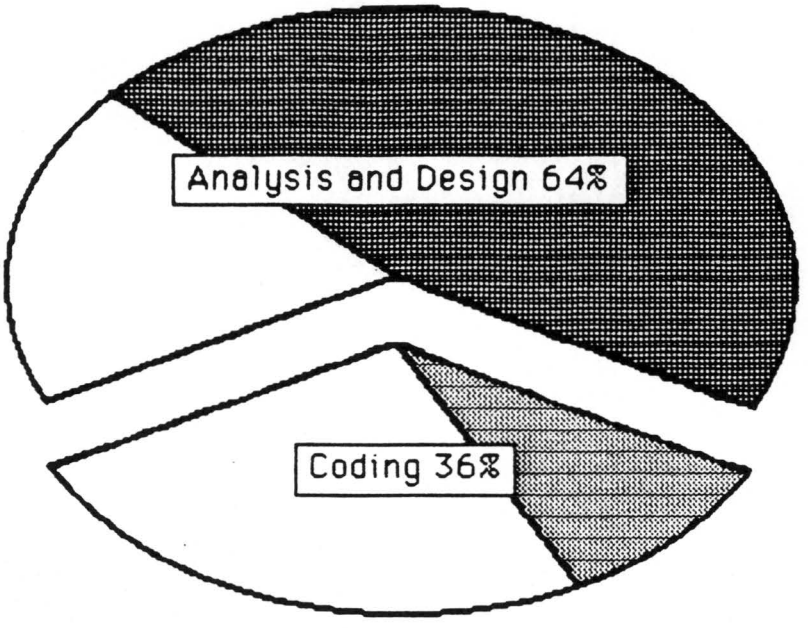
Thus, a means for displaying to users the dynamics of systems prior to the design stage within the life-cycle is desperately needed. Without prototype designs and mock-up systems this need cannot be met and programmer productivity remains stagnant.

Implementation is the realization of a system design through coding. Coding is the conversion of the system design into an optimized, machine-executable form. Once considered to be the most important and time consuming phase, coding accounts for only one-sixth of system development; the other five-sixths include integration, testing, and design [BROOK75]. During the coding stage, many design decisions, deferred during the design phase, are made and often lost, resulting in problems during maintenance.

After implementation, the code is next integrated and tested, using "real" data under "real" conditions, with "real" users, who were not present during design or implementation. This phase is the most painful for designers to endure, for during this phase incomplete requirements are usually revealed. One study [ROSSD77] showed that 20 percent of the errors revealed during this phase were attributed to missing information, and 30 percent to incorrect requirements definition. Another ([BEOHM75]) asserts that almost two-thirds of errors detected may be traced to faults in spec-

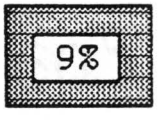
ification and early design. These errors are usually not detected by the system developer, but by the customer, and remain a primary source of user dissatisfaction. Figure 2.2.1, adapted from [ROSSD76], illustrates this point.

Analysis and Design errors are frequently not found by developers, but by users.





Analysis and design errors found during or after acceptance test.



Coding errors found during or after acceptance test.

Errors detected by users are a major source of user dissatisfaction.

Figure 2.2.1 Errors detected during testing

Having tested the system and revealed flaws, designers begin maintenance. Maintenance arises naturally when requirements have been left out or user needs have changed. This often involves addition of new functionality and performance to a completed design; sometimes revision of a very old design. In interactive systems, the user interface becomes a site for much revision; in large systems changes in the interface tend to propagate throughout the entire system, making interface improvement expensive and tedious. The maintenance phase is usually the most labor intensive, least cost effective, and most time consuming in the system life-cycle. Many programmers are employed on "program maintenance," but few relish the thought of being known as program maintenance engineers [DODDW80]. Figure 2.2.2 [ROSSD76] perfectly conveys this immense contribution of the maintenance phase to the entire life-cycle cost.

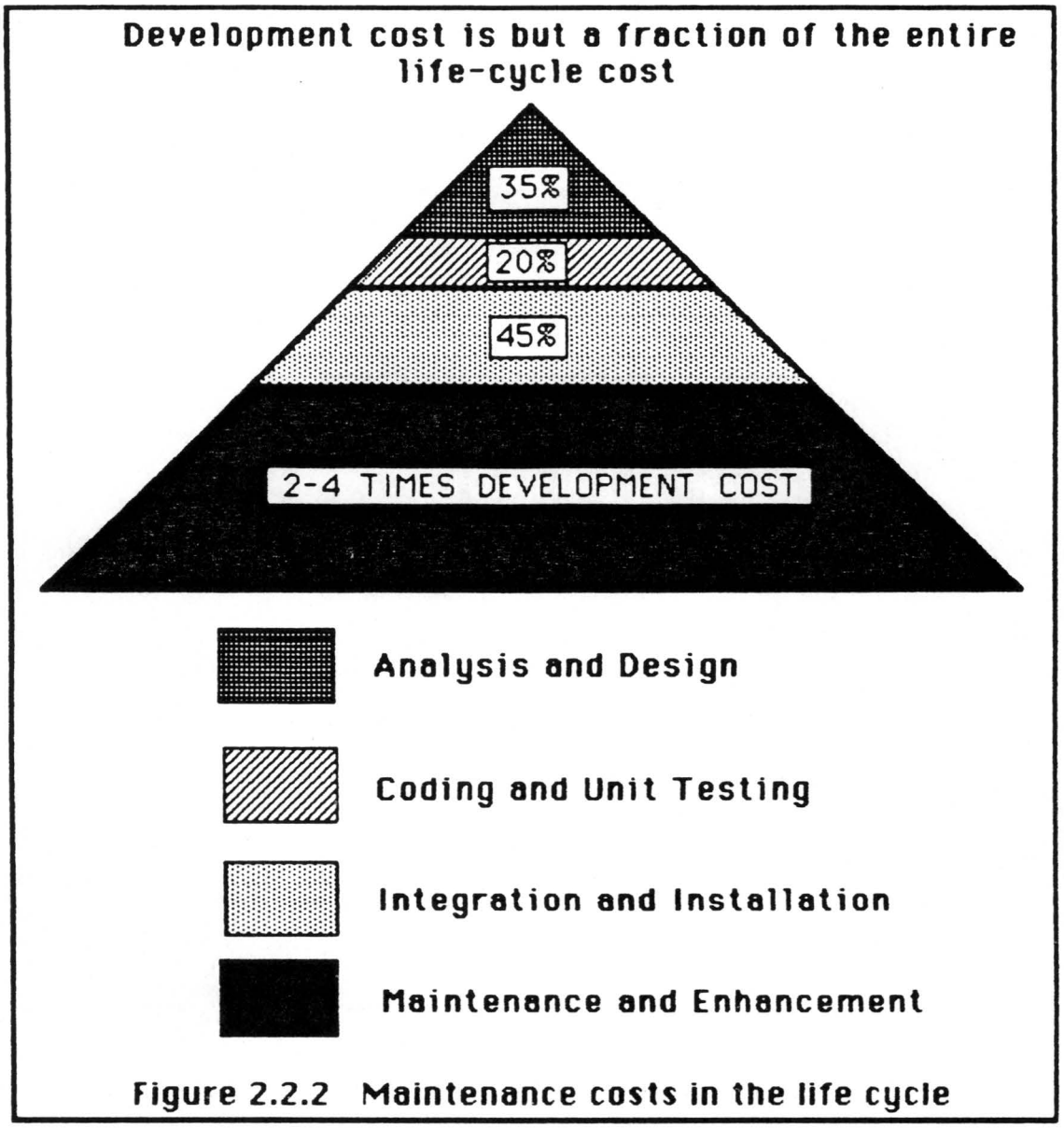


Figure 2.2.2 Maintenance costs in the life cycle

There are three basic techniques commonly used for developing software systems, each of which use some if not all of the phases in the life-cycle. One technique, called the "Build and Fix Technique [BOEHM84]," is simply to define the system, build it, then debug and refine it. This soon leads to what may be appropriately termed the "jalopy jury-rigging syndrome." Soon the system limps from patch-job to patch-job awaiting a satisfied, more appropriately bored, broke, or impatient user to approve the system.

Another approach, called the "Complete Specification Technique [BOEHM84]," involves a long period of requirements specification, followed by an even longer period of system design. Requirements are "ping-ponged" back and forth between customers and developers until the customers become convinced that what is on paper is what they want. This often results in the "starchy-shirt syndrome," wherein once the system has been designed and/or implemented, it becomes nearly impossible to remove the effect of certain design decisions in order to affect change. Users are bound by contract to accept a system they may not want, and they occasionally view the developers as being "stiff and unpliable." This invariably will occur in large systems because clients are usually companies with dynamic needs and policies. By the time a large system has been designed by this technique, the needs have changed (the company has

probably issued stock to pay for the long term development costs as well).

Another technique, called simply the "Prototype Technique [BOEHM84]," is becoming increasingly popular because of its responsiveness to change. The basic steps involved with this approach are listed below:

1. Identify the basic needs of the system.
2. Develop a working model (prototype).
3. Demonstrate the model to potential users.
4. Implement revisions and enhancements to satisfy users.
5. Build a fully functional system based on the prototype.

This approach at least intuitively provides better, more correct requirements and involves users early in the design phase of systems.

Below are a few quotations that illustrate how prototyping may be used in the design process and support this technique as the most useful of the three techniques discussed above :

> "There are three basic approaches to software design: build and fix, specify, and prototype. Of these three, prototyping is the best; the rest are obsolete [MCCRA82]."
>
> "We build prototypes today, but we call them the end product and deliver them to the end user [REPOR81]."
>
> "Errors in requirements specifications are usually the last to be detected and the most costly to correct [BOEHM78]."
>
> "The cost of amending software grows exponentially as the project progresses through the life-cycle [BOARH85]."
>
> "From the user's point of view, the best way to determine whether a system will satisfy his requirements is 'hands-on' use of the system [GOMAA81]."
>
> "By performing maintenance on the specification, we modify the form that is closest to the user's conceptual model, least complex, and most localized [BALZE83]."
>
> "The predominance of rework that occurs after conversion is not due to design or coding errors but to the ambiguous definition of requirements [BOARH85]."
>
> "The processes which convert requirements into a specification and then into an implementation, are informal, labor intensive, and largely undocumented... [BALZE83]."

The issues in prototyping are user satisfaction, early specification and design approval, correct requirements de-

finition, and the ability to have early system "test drives." Proponents of rapid prototyping as a design initiative seek to alleviate, if not eliminate, the maintenance phase in software system design.

For the current life-cycle paradigm to improve, a few advancements in each area of the life-cycle must proceed. These advancements involve at least the following:

- Requirements must be complete, correct, and unambiguous.
- Specifications must possess at the very least omniscient foresight.
- The design must be flawless and perfectly communicated and understood to all project members.
- Implementation must proceed quickly and accurately.
- Testing should be exhaustive and reveal less errors.
- Maintenance should evolve a finished design into a perfect product.

If these advancements seem impossible to achieve it is because they are. The only way advancements such as these could occur is through the use of prototypes and a refined life-cycle wherein phases are not so clearly delineated.

One may be easily convinced that prototypes are useful requirements tools prior to full implementation, but continue to resist their use. This may be due to a variety of reasons. Many believe that prototyping is prohibitively expensive.

In a 1980 international conference on software engineering, prototyping as a design tool was dismissed because of cost [DODDW80]. However, since 1980, recent advances in prototyping technology have made this approach quite cost effective. Recent uses of prototypes in the specification and design stages of development indicate that prototypes only contribute 10 percent of the development costs and that these costs are quickly balanced by reduced training costs for users [GOMAA81]. Since maintenance is usually the longest and the most expensive phase in the life-cycle ([ROSSD76] and [BALZE83]), any reduction in maintenance will reduce costs significantly throughout the entire life-cycle.

Another argument held against prototypes is that there is no evidence preferring their use over any other design technique. This argument is basically incorrect. Of the three design techniques currently being utilized (build and fix, completely pre-specify, and prototyping), prototypes have been shown to be best suited for large systems, interactive systems in particular [BOEHM84].

In 1984 Barry, Gray, and Seewaldt compared the prototyping approach and the complete specification approach in an interesting multi-project experiment [BOEHM84]. Seven teams of programmers were evaluated during the design of a 2000 to 4000 line software product. Four teams used the

complete specification approach, while the three remaining teams used a prototyping approach. The results were statistically significant and indicated that both approaches produced roughly equivalent product performance, the prototyping approach produced 40 percent less code and required 45 percent (including prototype development) less development effort. Prototyping produced products that were somewhat lower on functionality and robustness, but this finding was balanced by code that was easier to maintain and products that were easier to use and learn.

In a 1981 software productivity workshop a group investigating software productivity identified requirements definition and analysis, and rapid and early prototyping as two of the most important functional areas where very high level tools are needed [REPOR81]. Prototyping was identified as a very important tool in the development of consistent and correct requirements. The need for executable requirements and quick, cheap, and accurate prototypes was promoted as an important area of contribution for future researchers in programmer productivity. These findings are supported by the current trend in the industry of moving toward "malleable software" (Software that is easier to change, reuse, and enhance [COXBJ83]). Application areas identified as being best suited for prototyping include [BOARH85]:

- Operational Support Systems,
- Management Information Systems,
- Report Analysis Systems,
- Decision Support Systems, and
- Information Storage and Retrieval Systems.

Such systems are now occupying larger and larger percentages of the industry. To meet this growing need for prototypes, this committee ([REPOR81]) recommended a horizontal integration of tools which are supported by a chosen design methodology and common project database. Such an integrated set of tools, called "coordination tools" (coined by Anatol Holt [HOLTA79]), would require user friendly, graphic interfaces for untrained users. Use of such a tool set, prototyping tools in particular, could lead to favorable increases in productivity. Gomaa and Scott ([GOMAA81]) indicate that prototyping tools provide very helpful information in the specification of user requirements. They list the following as examples [GOMAA81]:

- Misunderstandings which inevitably occur between developer and user are quickly revealed.
- Ambiguities and inconsistencies in requirements and specification are identified.
- Omissions in requirements are discovered.
- Users realize essential features which are missing.
- Errors due to incorrect or missing requirements are discovered.

- Functions that do not provide users with the information they need are identified.
- Features which users find difficult or confusing to use are uncovered.
- Uncertain system functionality is resolved so that users can fully specify system functionality.
- System architectural requirements and data structure requirements are revealed to the designers.

The next chapter presents previous work in rapid prototyping which has sought to eliminate the problems of complete specification. This work has concentrated primarily on interactive systems, so is well suited for prototyping.

3.0 PREVIOUS WORK

This chapter discusses previous work in rapid prototyping and serves as a spring-board for the research contained in this thesis. Early work in prototyping involved the concept of structured programming and top-down refinement [DIJKS72]. The distinction of the dialogue component of interactive systems from the computational component did not exist, so this work involved automatic "stub" generating tools. These tools would quickly produce modules having no functionality that could be used during top-down testing of an evolving design.

After the distinction between dialogue and computational components in a design was made ([EHRIC81]), prototyping efforts took two approaches. One approach, sometimes called "dialogue simulation," involves the construction of sample user interfaces or dialogue prototypes, sometimes called "scenarios," very quickly and cheaply [HART84a]. These scenarios could then be evaluated and reviewed by customers at nominal expense to the developers.

The alternative approach to rapid prototyping involves construction of a more detailed prototype, often including some computational functionality of the production system as

well as dialogue. This approach is characterized by strict adherence to a methodology or structuring formalism which allows specifications to be "executed." These two approaches and their impact upon the computer industry are discussed in the following sections.

3.1 SCENARIOS

Scenario-based prototyping techniques model interactive systems as being dialogue controlled or dialogue dominant. The dialogue component of applications governs the order in which computational procedures are invoked, thus are given primary control. Generally, tools exist which are used to build primitive dialogue portions of a design. These dialogue components are then structured into a scenario which can be demonstrated to users. Users see the logical sequencing of a system through dialogue scenarios during these demonstrations. Each scenario is intended to trace one sequence of dialogue "transactions" through the system. Groups of scenarios would comprise a "psuedo-specification" of an interactive system when demonstrated.

Since users generally view interactive systems as being dialogue dominant, scenarios produced by these tools are useful for specification of interactive systems. Unfortunately, the displays produced by these scenario tools are

usually static, so can only convey partial information. They totally ignore modelling system behavior beyond the user interface; therefore, can only be used to produce a facade of the production system.

One such tool called ACT/1 supports construction of scenarios, called scripts, interactively and has been used to significantly increase programmer/analyst productivity [MASON81]. Mason and Carey attribute this increase in productivity to the tool's ability to describe the functional specifications of a system in sufficient detail for designers and in a language understandable by users. ACT/1 supports only fixed data in scenarios, but still remains a useful design tool for interactive systems. Early experiences with the tool suggest that changes requested most often by users are in fact in the user interface. Common changes in screens include the addition of information lines, reducing space for titles, rearrangement and addition of fields, and changes in the logical flow between screens. This tool has been evaluated and compared with previous techniques in a production environment. System designers reported that in production environments changes requested by users were similar to those in experimental environments but were typically more extensive. In such cases, multiple scenarios were used to fully demonstrate the requested changes.

The Functional Language Articulated Interactive Resources (FLAIR), another tool similar to ACT/1, uses sequences of menus to direct designers in the translation of scenarios to executable forms [WONG82]. FLAIR manipulates static frames and aids end-users through voice directed menus in structuring hierarchies of menus and screens. This tool is unique in its definition facility, which utilizes a variety of input media, making definition of scenarios easy enough for end-users to perform. **Human factorability in the interface of a prototyping tool is an important and desirable feature because it allows customers to be actively involved in the prototype process. If users are given an easy method in which to refine and better communicate requirements, they are more likely to receive systems that they not only accept but also recognize [BOARH85].**

Both ACT/1 and FLAIR are unstructured approaches to prototyping which do not rely on formal models or methodologies. Most other scenario approaches use BNF-type [LENOR77] description grammars or finite state machine descriptions ([JACOB83] and [WASSE83]) for structuring scenarios. These other approaches are more formal, hence in some respects more attractive, but lack the ease of use so necessary for end-user involvement.

A BNF grammar is used in the Interactive Dialogue Synthesizer (IDS) to describe scenarios and perform dialogue simulations [LENOR77]. Displays are described in machine-independent semantic actions which execute on an abstract machine. The interface specification are executed as simulations by language processors, automatically generated by IDS during translation of the BNF. Dynamic scenarios may be constructed by sequencing a number of static displays with dynamic displays interpreted by the language processors. IDS has been evaluated in a military setting and has experienced great success in a variety of military command and control systems ([HANA80a] and [HANA80b]).

State transition diagrams have also been successfully used as a description technique for scenarios ([JACOB83] and [WASSE83]). Jacob ([JACOB83]) models dialogue components in software systems as states and user interactions with tokens, labelling transition arcs. Transitions can only be made on either input or output, not both, and no provision is made for concurrent I/O activity. Complexity of the state diagrams is kept to a minimum by introducing subdiagrams into the notation. These augmented diagrams are then translated into a purely textual form which are interpreted for demonstration. Jacob illustrates the use of this technique in a Military Message System (MMS) application.

The most promising approach taken in scenario construction uses an interactive information system design methodology in which prototyping is an important phase in the life cycle. The User Software Engineering (USE) design methodology and dialogue management system was developed to involve users early in the development life cycle as opposed to the traditional separation of developers and users ([WASSE79], [WASS82a], [WASS82b], [WASSE84]). The major premise in this research effort is that the user interface is important to the overall quality and user satisfaction of a product. A number of design objectives for USE follow [WASSE85]:

1. "The end result is a working, evolvable system that the user likes.
2. There is a balanced approach between prototype development and a phased life cycle approach.
3. There is a straightforward mapping from the specification to a prototype to a production program.
4. The methodology has both a formal and an informal aspect, making it feasible to undertake formal specification and verification of an interactive information system."

RAPID, the rapid prototyping tool in USE, represents interface design by state transition diagrams in a manner similar to Jacob's tool. Integration of RAPID in a dialogue management system and design methodology such as USE has made this approach more holistic than that taken by Jacob. Scenarios are created with a state diagram editor and may in-

clude dynamic displays which are later linked using a semantic action linker. As with Jacob's approach, dialogue is the dominant model of control and state diagrams are translated into a textual format which is then interpreted during scenario demonstration. Subdiagrams are used to control complexity and provide a structured method for top-down decomposition. The state transition diagrams have been extended to include variables which may contain results of dialogue interactions. These variables become the medium through which computational components communicate with dialogue components and vice-versa. The textual language is device independent and provides full screen and cursor control. Unfortunately, the specification language for screen formats is quite cryptic and not very human factored.

The USE approach, unlike that of Jacob, supports gradual evolution of prototypes into completed systems. This evolution is a two step process involving first the Transition Diagram Interpreter (TDI), the prototyping tool, and secondly, the Action Linker, a tool that links computational components to dialogue components. Once a group of scenarios has been user-approved, the computational components can be added to the textual description and linked to compiled object modules, using the action linker. Use of RAPID without the action linker leads to a number of shortcomings, three of which Wasserman and Shewmake identify [WASSE85]:

1. Realistic displays are difficult to provide if displays are complex.
2. Dialogues may result from computational actions as well as user input.
3. The user really needs a working system, not a facade.

Solutions to these problems have only been partially solved. Realistic displays can be created using static data. Displaying the possible branches and allowing users to guide control solves the second problem. **Lastly, "there is no solution to the third problem, other than to build all or part of the system [WASSE85]."**

An important feature of the Transition Diagram Interpreter is that it allows scenario sessions to be saved for future use. With this feature, variables may be recorded and scenarios "re-played." This feature prevents the prototyping effort from "looping," that is, designers discarding prototypes, only to discover later that a new prototype is in fact a prototype that has been demonstrated previously.

The RAPID system is an example of "executable requirements," which is based on conventional requirements definition, wherein requirements are first identified, functionality of the system is specified, then specifications are designed using some executable representation. The next section describes an alternative to conventional specifica-

tion and presents previous efforts in prototyping which rely on this new approach.

3.2 OPERATIONAL SPECIFICATION

In the conventional approach to software design, requirements analysis precedes system specification and specification precedes design. This assumes that users can completely identify and communicate all of their needs, and developers can produce specifications that not only meet all the user's needs, but also those needs left incomplete or misunderstood. Essentially, complete specification must take place prior to design and implementation. The inability of the current life-cycle paradigm to meet this lofty requirement is beautifully illustrated by Winston [WINST75]: forgotten and misunderstood requirements have caused complete failure in some large software products, and in others, 95 percent re-design. Solutions to this dilemma reveal themselves once we realize that "every specification is an implementation of some other higher level specification [SWART82]." Designers seek a valid implementation of a desired system (Valid meaning that the behavior of the finished product matches that defined during specification). In practice, however, this validity constraint is violated. Certain design decisions, known only during design or implementation, mandate that designers redefine system specifica-

tions. A complete and valid specification, if such could be defined, would remain static throughout the life cycle. It is this inherent flexibility of specifications that motivates Swartout and Balzer to argue on "the inevitable intertwining of specification and implementation [SWART82]."

Realization that specifications are indeed intertwined with design has lead many researchers to seek executable specifications, which clarify implementation implications and interactions early in the life cycle. This approach to system design relies on a modified life-cycle concept and makes executable specifications a reality. Currently, executable specifications and program transformations (new ideas believed to improve programmer productivity) have no place in the current life-cycle [ZAVEP84]. **The successful integration of these concepts will only be possible through some radical new approach to software development and a new life-cycle paradigm.**

In chapter 2, the conventional life-cycle approach was explained from a phase viewpoint. Figure 3.2.1 (adapted from [ZAVEP84]) presents the conventional approach from an activity point of view. This approach can be contrasted with the operational approach, depicted similarly in Figure 3.2.2 (adapted from [ZAVEP84]). The operational approach is characterized by specification of systems in terms of

implementation-independent structures which may be "executed" to demonstrate the behavior of the specified system. It is this ability to execute the behavior of a system at any point during development which may be termed **behavioral demonstration** [CALLA83]. Unlike scenarios, this approach to specification allows rapid prototyping for all systems, not only interactive ones. The specifications are problem-oriented rather than implementation-oriented, so little expense is involved in specification over that of the conventional approach.

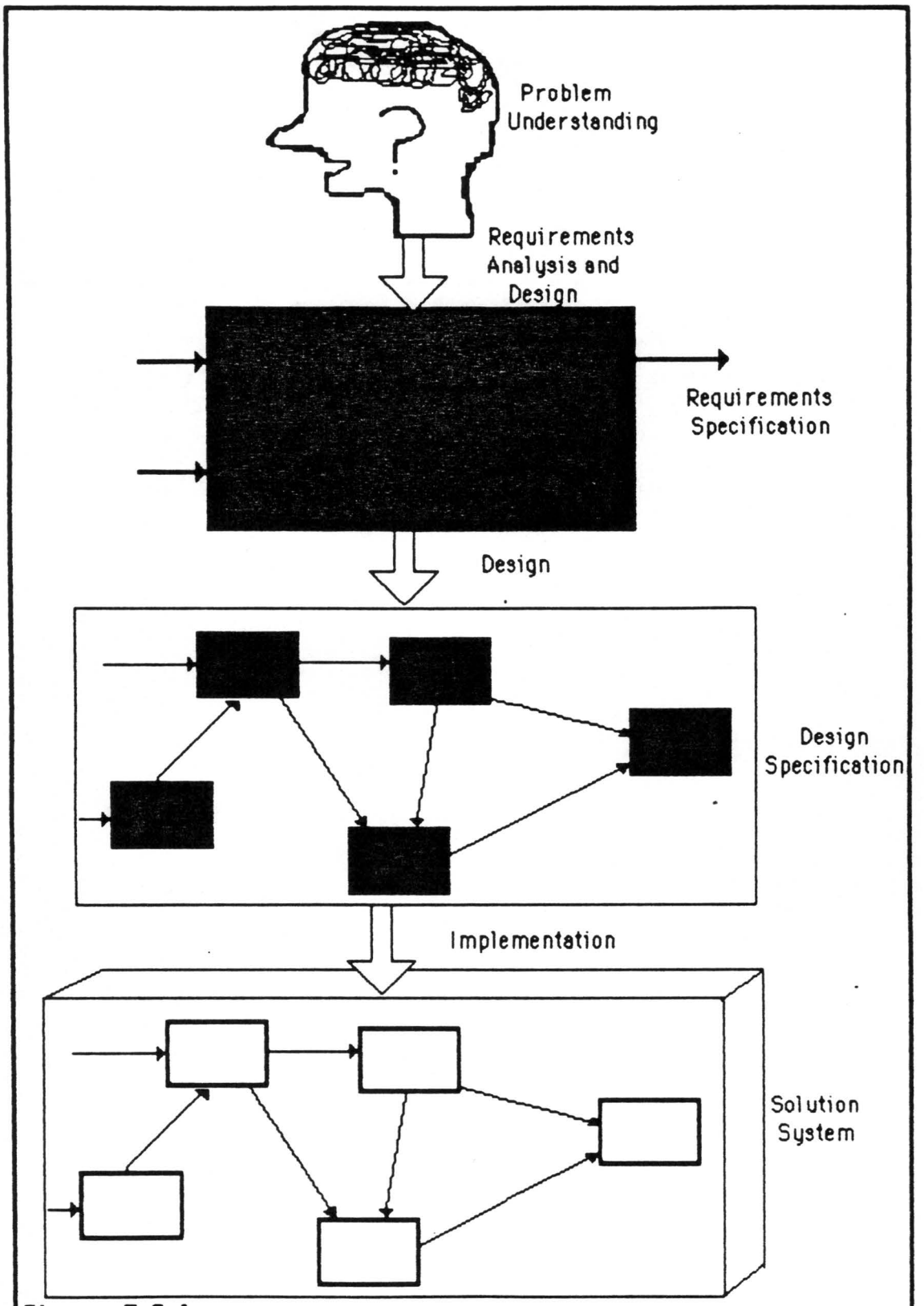


Figure 3.2.1
Conventional Approach to Software Development

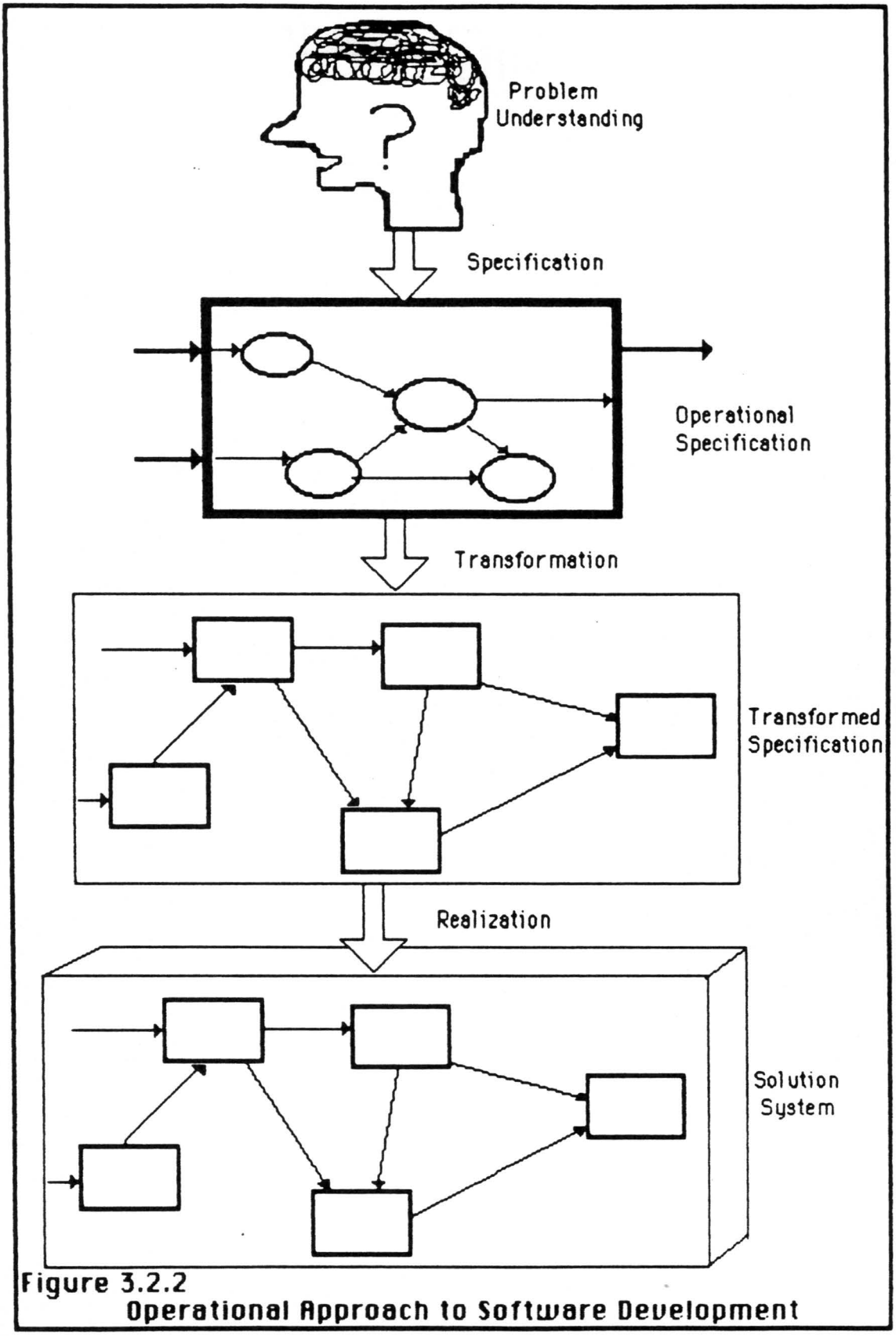


Figure 3.2.2
Operational Approach to Software Development

Some recent research efforts have sought executable requirements through operational specification ([ZAVEP81] and [BALZE82]). Through operational specification, executable specifications become prototypes, which are transformed into the final system. These executable specifications are typically written in a well-defined, formal language which can be interpreted to demonstrate the behavior of the system specified. Of particular interest is the PAISLey specification language ([ZAVEP80]) and its application for embedded systems (Computer systems which operate as part of some larger integrated network of machines) [ZAVEP81].

Embedded systems are extremely hard to test. Some example embedded systems which can have catastrophic results if tested are nuclear missle-guidance systems and patient-monitoring systems. One would like to avoid potential catastrophies during testing by verifying specifications prior to design. The operational approach allows designers to model not only the software, but also the environment in which the software is to function prior to design. Both data and control must be modelled in embedded systems, and for these systems, control is paramount and comes in a variety of forms. Embedded systems usually are modelled by a set of cooperating asynchronous processes and thus do not fit into the scenario approach to prototyping.

PAISLey has been used for executing specifications for a few small, embedded systems. Its major advantages are that it is a formal, internally consistent, strongly typed language allowing execution of specifications. Therefore, demonstrations to customers will not halt, deadlock, or go into undefined states. The PAISLey approach, however, offers no graceful transition to a production system, but offers a vehicle for user demonstrations, performance simulations, and prototype construction.

The PAISLey project did produce a significant result for operational specification techniques [ZAVEP81]: "A specification can be executed at **any** stage of development if a means for evaluating as-yet-primitive functions is supplied; two reasonable possiblities are

1. selecting a value at random from a declared range, or
2. displaying the function's values at a terminal and asking the analyst to supply a value."

This thesis attempts to resolve some of the problems left unsolved in both the scenario and operational specification techniques and to exploit features found to be very useful for rapid prototyping tools. These problems, features, and solutions are detailed in the next chapter.

4.0 ATTRIBUTES OF A NEW APPROACH TO PROTOTYPING

This thesis has presented in previous chapters a motivation for prototypes, the life-cycle paradigm, and previous work in rapid prototyping. This chapter compares previous approaches to prototyping and suggests attributes and features that should permeate future prototyping tools.

4.1 EVALUATION OF CURRENT DESIGNS

The scenario approach to rapid prototyping has lead to increases in programmer productivity because needs in interface design and system architecture could quickly be identified and fulfilled. Design of scenario tools can be divided into two basic classes: formal designs and informal designs. Formal designs emphasize designer manipulation of scenarios, while informal designs seek to provide a common framework in which both designers and users may communicate. Formal designs have the advantage that information derived from their expression may lead designers to discover inconsistencies in system design. Informal designs allow users to verify system completeness and correctness, and eliminate many misunderstandings that occur between designers and users. Some formal designs provide for dynamic displays, which are closer to a real system; whereas, informal designs offer

only static displays. Those efforts that have relied upon a consistent design methodology have succeeded where others have not and have also sought to gradually evolve prototypes into deliverable products. Use of a design methodology has also lead some researchers using the scenario approach to conclude that even in interactive systems, dialogue dominant control is an inadequate model (WASSE85] and [JACOB83]).

Operational specification efforts, rather than concentrating on interactive systems, have dealt almost exclusively with embedded systems. This approach seeks to quickly identify design flaws through executable specifications. This approach promotes automatic program transformation techniques for moving from specification to design to implementation. Being a more fluid approach, operational specification does suffer from the handicap that simulations remain simulations and do not evolve as the true system does. Hence, users have only an incomplete sense of how the completed system will behave. These simulations are good for determining correct requirements, but can not help users determine whether all functional and performance constraints have been included, nor can they help designers experiment dynamically with design alternatives. Operational specifications have not been used thus far for general system specification, interactive systems in particular.

Both scenario construction and operational specifications provide valuable information useful in the software life-cycle. The next section promotes the careful integration of these two approaches with a new life-cycle concept and suggests features that future prototyping tools should possess.

4.2 FUTURE PROTOTYPER FEATURES

Some authors have suggested that with prototyping as an option in the software life-cycle, current design techniques and even the life-cycle concept itself may be detrimental to the software industry [MCCRA82]. If such is the case, then prototypers should provide designers with more information than they currently do. Software has become such an expensive commodity that developers and customers need ways of estimating and controlling costs. Developers need ways of controlling man-machine resources, while customers need to know exactly how much a particular system feature will cost to add to an evolving system.

If prototypes become a prevalent design technique, software experimentation and conjecture will become common. In such an environment, users may insist on almost continual demonstation, gaining little progress toward a final system. Such activity will result in **thrashing** of system designs

[BOARH85]. To prevent thrashing, charging users for demonstrations has been suggested. This would eliminate thrashing, but would not help designers determine exactly when demonstration must end and product evolution should begin.

One solution would be to develop a means whereby project managers could inform users of how changes would impact a design, then both groups could prioritize these changes. This information would be available for users if designers could estimate the effort of changes, and thus the cost. Unfortunately, metrics for estimating maintenance effort are not sufficiently advanced to derive this information [HENRY85]. Future prototyping efforts should seek to collect metrics that could be used to derive this information.

Operational specification evolved as a specification technique that could represent design as an integral part of the specification process. Specification modifications occur in part as a result of design decisions. Reasons why certain design decisions occur are commonly lost after a system has been coded, creating many problems during maintenance. Most maintenance problems have been attributed to basic losses of information commonly known during specification and design. In a recent study conducted to determine features that should be included in professional programmer workstations, a panel determined a list of questions that are usually needed for

maintenance, but whose answers are usually lost. These questions are [GUTZS81]:

1. "Why a particular design decision was made.
2. How much individual developers spent on different phases of the project.
3. How many revisions were made to each module and why they were made.
4. How modules are organized into overall software structure.
5. What the sets of test data were and what the overall test plan was.
6. What the performance characteristics were for specific modules.
7. Which modules satisfied which specific system requirements."

Since prototypes help refine and verify design specifications, constructing them should provide useful answers to some, if not all, of these questions. **If these answers were recorded as part of the prototyping effort, reduction of the maintenance effort should result, and prototyping would become an integrated technique which designers use to structure the development of software systems.**

4.3 NEW TOOLS FOR THE FUTURE

The notion of integrated tools and techniques has been promoted as a means for increasing productivity. At a recent

conference investigating productivity, a panel identified six functional areas where tools are needed [REPOR81]:

- Maintenance of existing code.
- Requirements definition and analysis.
- Rapid and early prototyping.
- Management visibility.
- System generation using prefabricated models and reusable code.
- Distributed system definition.

The group recommended a horizontal integration of tools during single phases of the development and vertical integration of tool sets which support a given design methodology and use a common project data base. Tool capabilities included user adaptability, graphical and tabular I/O, and the ability to generate data about systems under development. For requirements definition, the following recommendations were made [REPOR81]:

- Requirements tools should aid analysts in structuring system requirements. Alternate views such as data flow versus control flow may need to be supported.
- Requirements tools should allow assessment of functional and performance capabilities and should insure consistency and completeness.
- Requirements tools should be interactive.
- Requirements tools should have an underlying formal model basis, but should also be understandable by customers.

- Requirements tools support determination of the impact new specifications have on designs and should allow for automatic documentation of this information.

Recommendations for maintenance tools included:

- Configuration management tools.
- Tools which support management of multiple versions of designs.
- Testing tools.
- Tools which can collect data necessary for maintenance.

Recommendations for rapid prototyping tools included:

- An executable requirements language should be supported.
- Prototypes are needed early in the life cycle at minimal cost.
- Tools should use standardized and reusable components to minimize costs of initial requirements definition.

Management visibility tools would allow managers to view the progress of an evolving system. These tools should track the history of components of a design from initial conception to component retirement. Such tools provide automatic mechanisms to track project status. Recommendations for these tools included structuring the design process so that every phase of the life cycle produces a set of machine readable/displayable deliverables.

Of all phases in the life cycle, maintenance, being the most expensive, has been deemed the most worthy of attention. Elimination of the great cost involved in software maintenance is one of the industry's toughest problems. Suggested solutions to the maintenance dilemma seek to resolve the problem by changing other phases in the life cycle, hoping that these small perturbations will "trickle" down as improvements to the maintenance phase. Attempts at eliminating, or even shrinking, this problem in this way have failed. It is as if there is a fundamental flaw in the life-cycle paradigm that prevents smaller maintenance phases. Admission that a flaw exists in any well-accepted paradigm invariably meets with disapproval, and in many cases uproar. For this reason a suggested new paradigm has been labelled a "radical new approach to software [BALZE83]."

A recent paper describing software technology for the 1990s identifies two flaws in the life cycle which have led to the maintanance problem [BALZE83]. The first problem is that the processes that translate requirements into code produce a vast amount of knowledge that goes unrecorded and is unavailable for maintenance. The second is that maintenance is performed on source code, not on requirements. Source code includes optimizations which programmers make mentally and which go unrecorded. This results in lost information. To achieve any improvement in the maintenance

phase, maintenance must be done on unoptimized requirements, and optimizations recorded as they are made. For this reason, an automation-based paradigm is suggested as a viable replacement for the current life-cycle paradigm. Figure 4.3.1 compares the current paradigm with the automation-based paradigm suggested in [BALZE83]. This approach would yield four important benefits which Balzer, Cheatham, and Green identify:

- No clerical errors are made in an automated environment.
- Developers are freed from optimizations, thus can do a better job considering optimization issues in the specification stages.
- Documentation will be more consistent and complete when requirements are kept up to date.
- The environment can generate reusable software, reducing coding and maintenance effort.

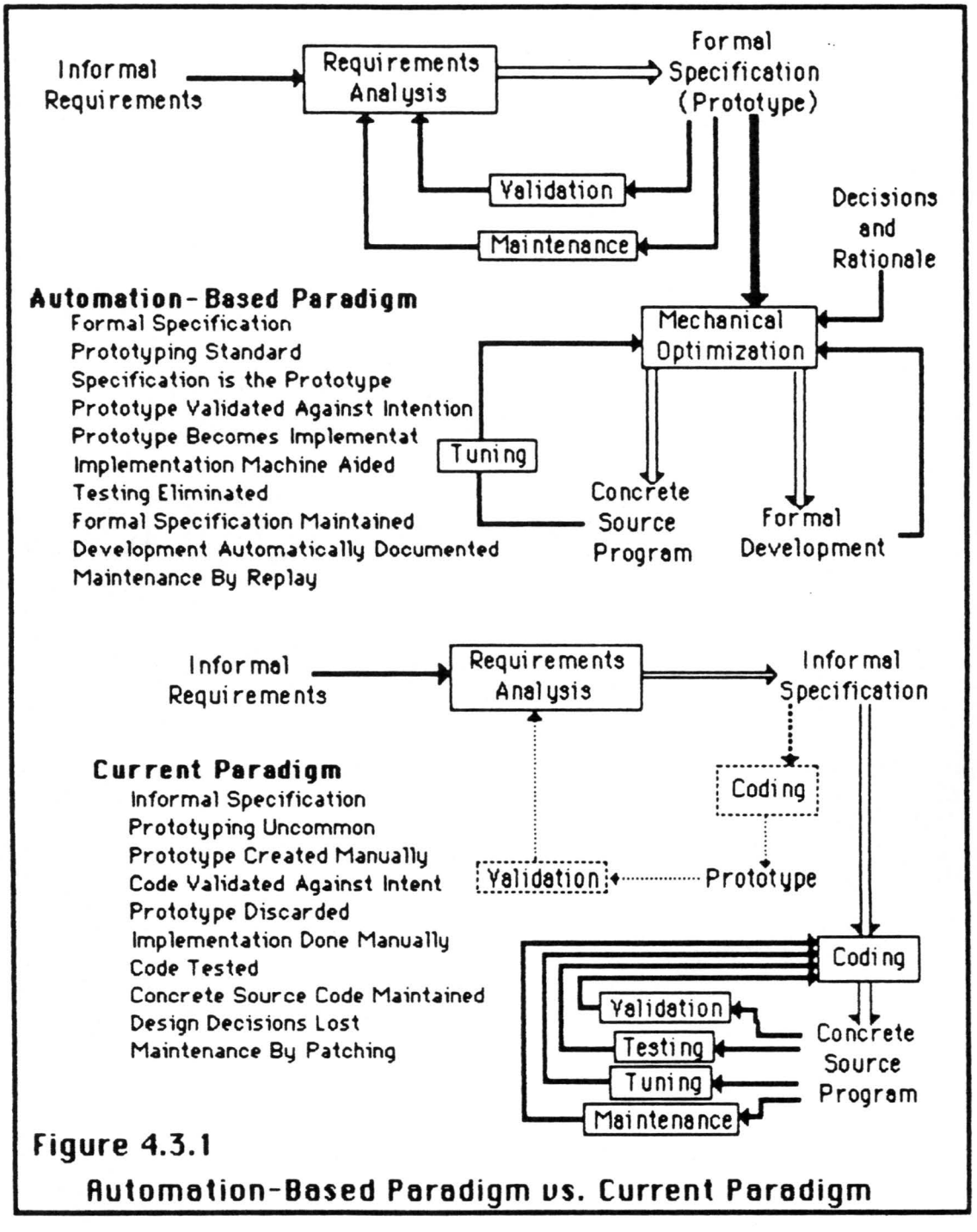


Figure 4.3.1

Automation-Based Paradigm vs. Current Paradigm

This approach depends upon a life-cycle paradigm in which specifications are the design, and this design gradually evolves into the product. Unfortunately, such a paradigm cannot exist without the necessary design methodology and support environment. **The research in this thesis presents a prototyping approach, based on a new design methodology (SUPERMAN [YUNTE85]) and support environment (DMS [HART84a]), that makes such a "radical new approach to software" possible and practical.**

5.0 SUPERVISORY METHODOLOGY AND NOTATION (SUPERMAN)

The need for a structured approach to software development that controls complexity in the design process is well known. A well disciplined approach using stringent standards is difficult to enforce when system developers are all from the same field, such as software engineering. When developers are from a variety of fields such as software engineering, psychology, human factors, and design automation, a simple, flexible, discipline is crucial if only for project communication. SUPERMAN has been developed as an all-encompassing design methodology to meet these needs and to provide a consistent, clear notation for specification of software systems [YUNTE85].

5.1 SUPERMAN VS. OTHER METHODOLOGIES

SUPERMAN has been created to simplify the conceptualization, design, construction, and modification of interactive systems, but can be used to develop any procedural system including completely manual systems, non-interactive systems, and embedded systems. Conventional methodologies have emphasized either data flow or control flow and typically require considerable training to use. Interactive system design methodologies have modelled either the compu-

tational component or the dialogue component as the dominant force of control in systems. SUPERMAN was designed to notationally balance data and control and to clearly distinguish the dialogue, computational, and control components of systems.

Research in cognitive psychology [PAIVI71] indicates that visual information conveys more total information to humans than purely verbal information. Implications of this research have led to a graphical system representation and notation in SUPERMAN. Existing methodologies which begin with a graphical system representation translate this into a purely textual representation that is harder both to understand and to maintain. Little research has investigated the approach of maintaining the graphical representation as the system representation and only translating the code automatically during the coding stage. Such an approach leads to maintenance on a graphical, presumably easier to understand system representation. In this way, SUPERMAN supports a graphical programming environment wherein all system conceptualization, design, implementation, and modification is done graphically.

From the dawn of interactive applications, people have realized that programmers are often not good interface designers. For this reason, SUPERMAN provides a task sepa-

ration which distinguishes dialogue design from computation and control design with roles. The **dialogue author**, a derived role [HART84a], is a human factors engineer who is proficient in the creation of good, consistent user interfaces and who designs dialogue components. Software engineers, proficient in the design of system architecture, design computational and control components. These two communicate through a common graphical notation understandable by both.

5.2 SUPERMAN NOTATION

In SUPERMAN, systems at their most abstract level of development consist of supervisory cells structured in a hierarchical representation. Systems are composed of concurrent supervisory structures. Structures are composed of Operational Procedure Diagrams (OPDs) and Supervised Flow Diagrams (SFDs). Human-computer systems can be viewed as either software systems, depicted by SFDs, which support human activity, or procedures, depicted by OPDs, which involve humans, computers, or both which cooperatively work together toward some management objective. This hierachical representation of systems and corresponding views is depicted in Figure 5.2.1.

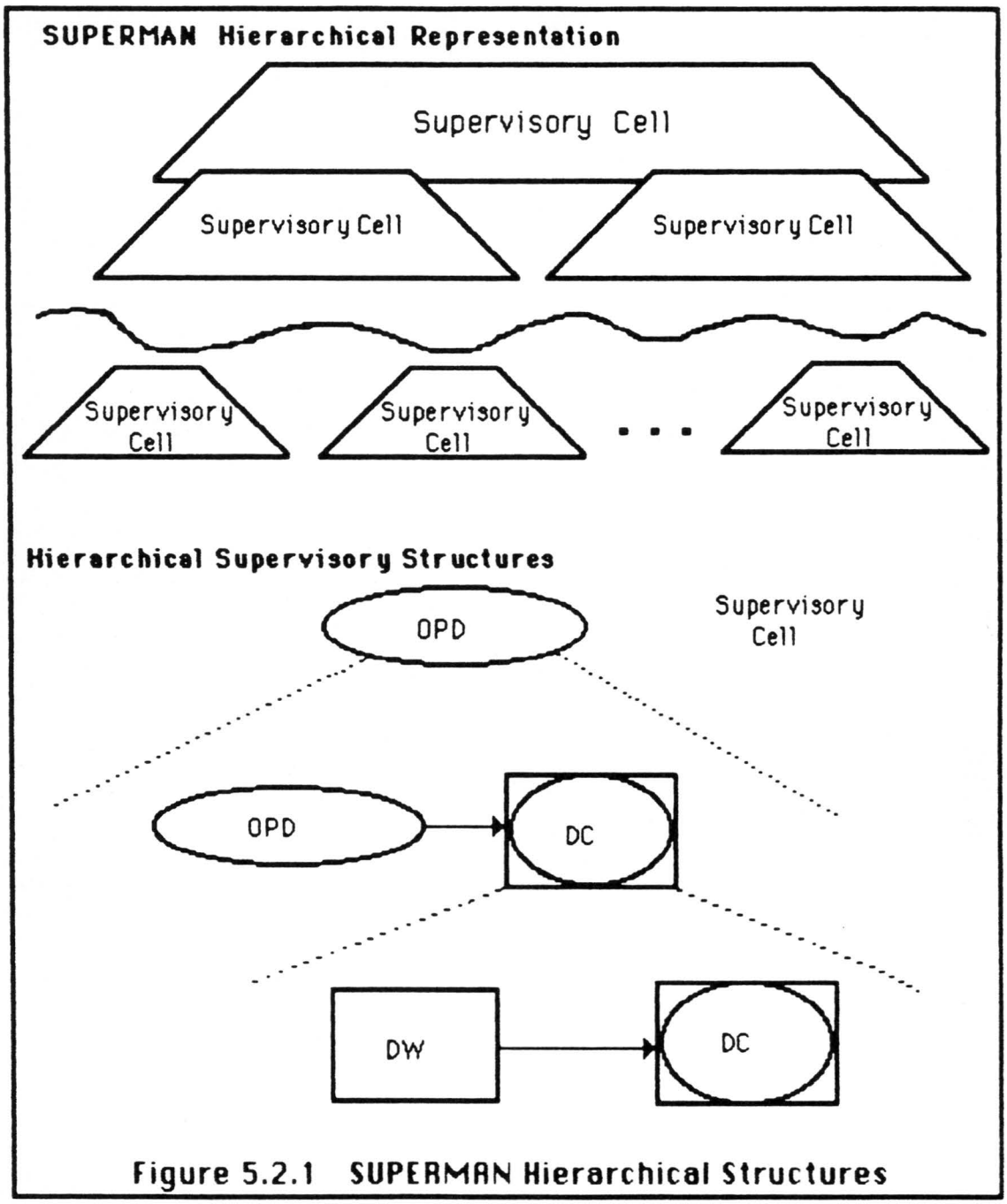


Figure 5.2.1 SUPERMAN Hierarchical Structures

OPDs describe, from both a managerial perspective and a user perspective, the operations and procedures that are necessary to meet desired goals and model the flow of managerial policy and goals through a system. OPDs are comprised of any combination of OPDs, SFDs, and manual functions. SFDs describe procedures in the software system and model the behavior of software in response to managerial policies. Manual functions describe corresponding functions that a user (or cooperating embedded piece of hardware) would perform when using the system and to model user behavior in response to software and policy. The remainder of this section will describe the syntax and semantics for SUPERMAN in the context of the foregoing discussion from a software system point of view. Alternate points of view follow directly.

Systems are hierarchically decomposed into supervisory cells. Wherein each supervisory cell defines "what" is to be done in a given function, the corresponding expansion of a supervisor cell defines "how" the function is to be performed. This expansion is known as the supervised flow diagram corresponding to its supervisory cell. As depicted in Figure 5.2.2, it is quite possible for functions in an SFD to be supervisory cells for other SFDs. Consequently, each supervisory cell may have two types of subfunctions in its SFD expansion: supervisory functions and worker functions. Where the expansion of a supervisory function is a corre-

sponding SFD, the expansion of a worker function is the executable source code of a high-level programming language such as FORTRAN. Like FORTRAN, the SFDs have their own syntax and semantics. As mentioned, graphic symbols comprise the SFD syntax, and the control constructs dictate its semantics. Figure 5.2.3 illustrates the graphical symbols included in the SFD language.

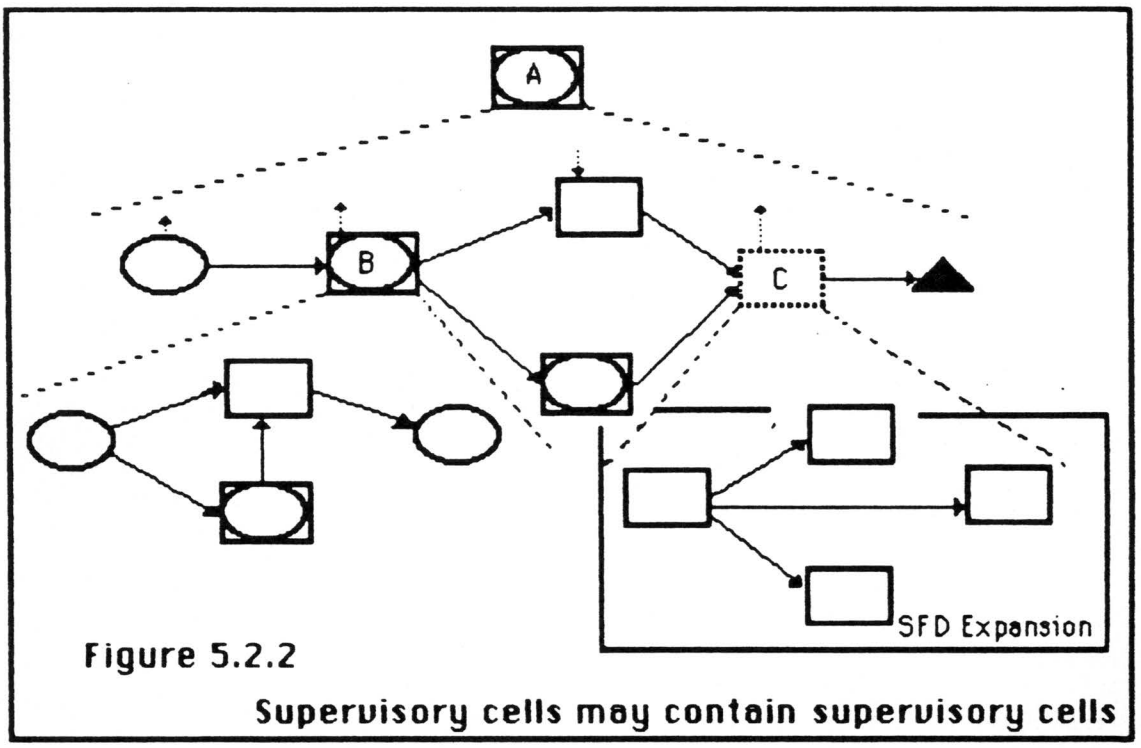


Figure 5.2.2

Supervisory cells may contain supervisory cells

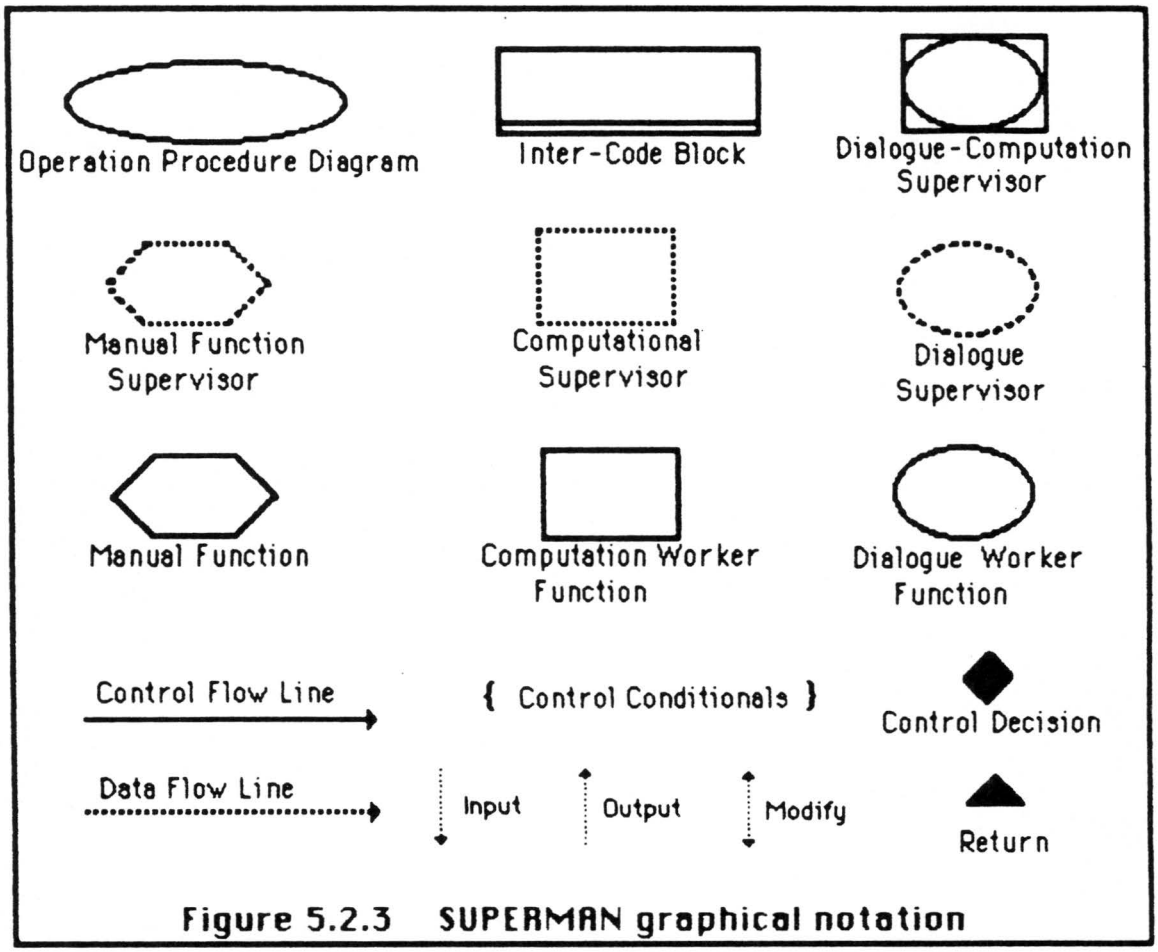


Figure 5.2.3 SUPERMAN graphical notation

The dialogue-computational node denotes a supervisory cell containing both dialogue and computational nodes in its SFD expansion. The dashed dialogue and computational nodes represent supervisory cells containing only dialogue or computational functions. **(The words node and function are used interchangably throughout this work.)** Intercode Blocks (ICBs) contain in-line code of some high-level language. ICB's represent the lowest level by which software systems may be decomposed, but are sometimes included in SFDs because the purpose they serve may not warrant creation of a full computational node. Control flow and data flow are represented by solid lined and dotted lines, respectively. Control conditionals, labelling alternative paths from a node, are contained within simple braces { }. Shaded triangles represent the logical end of control in a SFD expansion. They are called return symbols for their similarity to the RETURN statement in some high-level languages. Explicit conditionals in a control path are denoted by shaded diamonds. For full expressive power, four types of control constructs are defined. These are illustated in Figures 5.2.4-5.2.7. Appendix A contains the full notational description for SUPERMAN. Further information can be found in [YUNTE85]. The next chapter explains the integrated environment that supports this graphical notation and the SUPERMAN methodology.

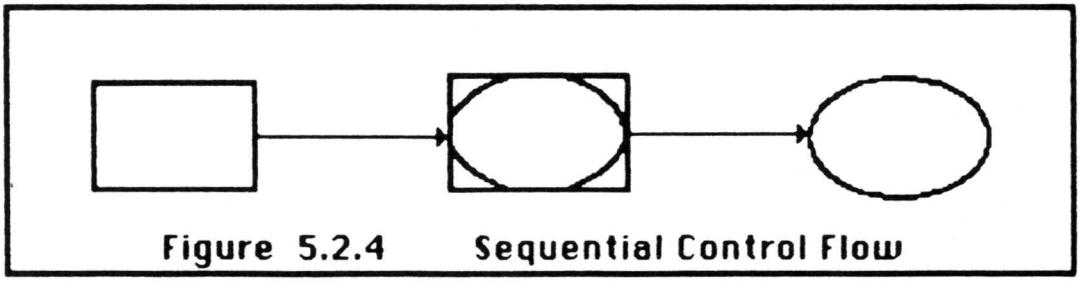


Figure 5.2.4 Sequential Control Flow

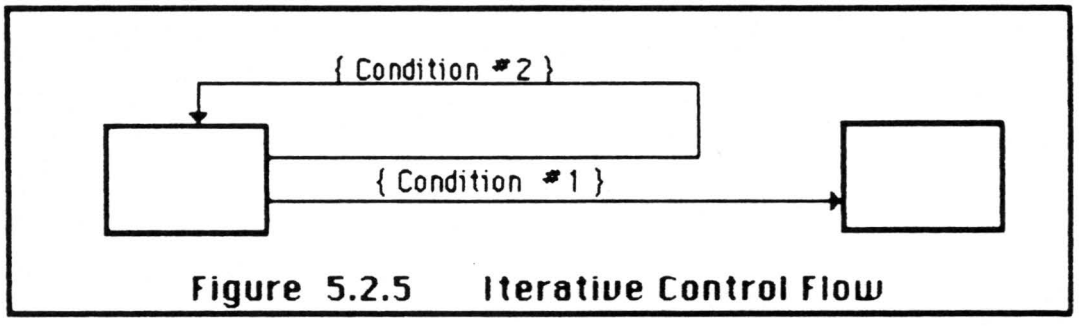


Figure 5.2.5 Iterative Control Flow

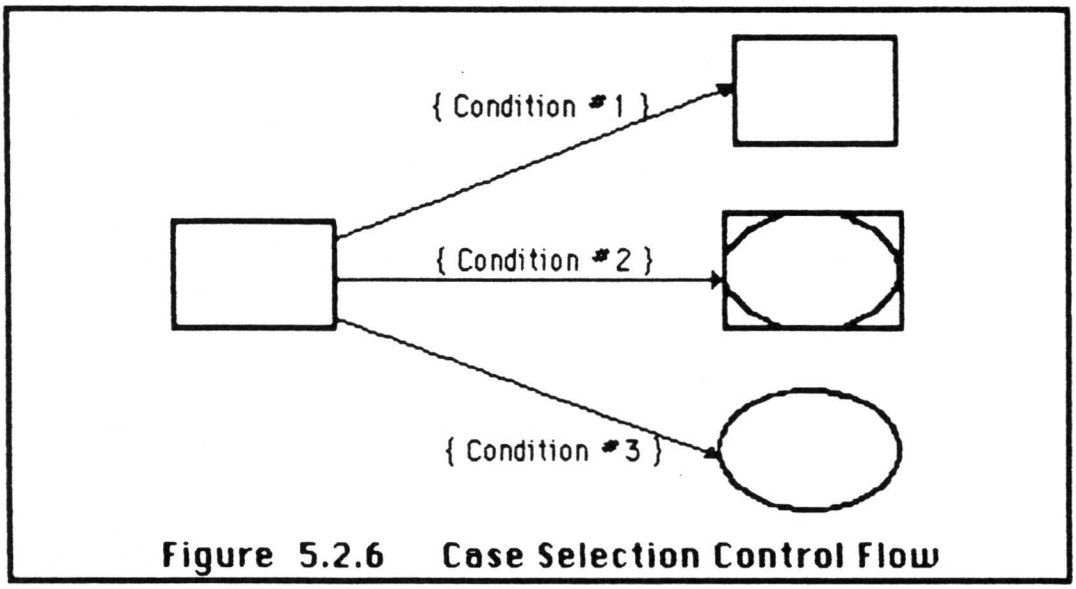


Figure 5.2.6 Case Selection Control Flow

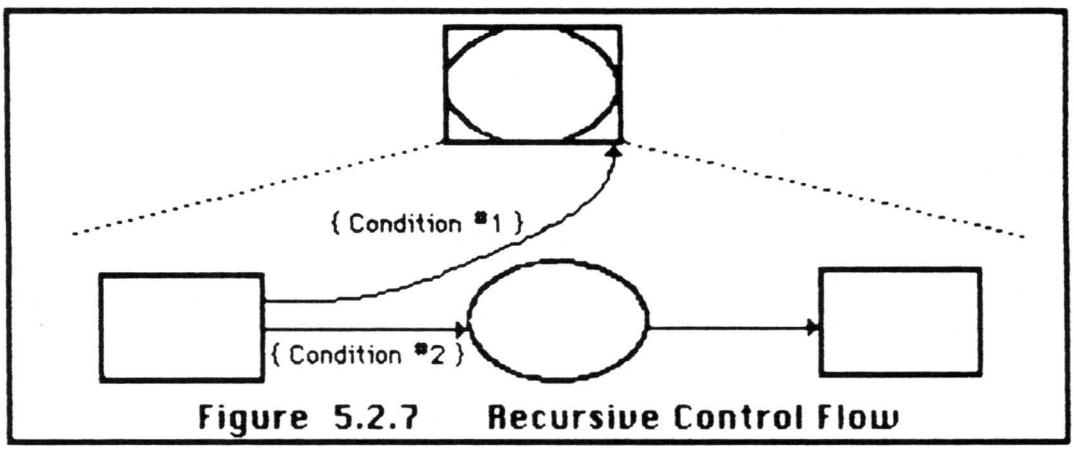


Figure 5.2.7 Recursive Control Flow

6.0 THE DIALOGUE MANAGEMENT SYSTEM (DMS)

SUPERMAN supports the separation of dialogue, computational, and control components of systems through its notation, while the Dialogue Management System (DMS) supports this separation during system development using SUPERMAN. This is accomplished through a new concept called **dialogue independence** ([EHRIC81]), which is "manifest at the highest level of DMS in three ways: by providing a holistic design methodology for the production of integrated systems, by providing parallel sets of automated tools with which the dialogue and computational components are implemented, and by providing separate execution environments for each of the components. [HARTH85]" In DMS, computational components are coded and compiled, while dialogue components remain data-driven and interpreted. This separation has a number of healthy benefits [HARTH85]:

- Prevents modification of either dialogue or computational components from propagating to the other. This isolates changes and enhances maintenance.
- Clarifies dialogue components. Because dialogue is data-driven and interpreted, rather than coded, it is easier to understand and change. DMS uses direct manipulation tools ("What you see is what you get") to design dialogue components, simplifying the process of change considerably [JOHNS85].
- Encourages the involvement of behavioral specialists in the software life cycle, because systems using this approach are easier to understand and change [JOHNS85].

In such a hybrid development environment, the notion of dialogue "modules" has been replaced by **dialogue transactions.** Dialogue transactions delineate the basic model of software systems, that is, applications typically being broken down into steps involving a dialogue input transaction, followed by some computational processing, followed by a dialogue output transaction [JOHNS85]. Each transaction may be subdivided further into dialogue interactions, but discussion of dialogue components at this level is beyond the scope of this thesis. Interested readers should consult [JOHNS85] for a full discussion of the dialogue transaction model and its use in DMS.

The development and implementation of DMS has been driven by numerous requirements [HARTH85]:

- Separate dialogue and computational development tools.
- Supportive of SUPERMAN.
- **Rapid demonstration of system behavior.**
- **Fast redesign cycle.**
- Human factorability (ability to quickly change and improve human-computer interfaces).
- Representing interfaces as languages.
- Direct manipulation interface design for tools.
- Highly integrated tool sets.
- **Metered system development.**
- Predictive metrics.

- Support for a large variety of input/output devices.
- Device independence.
- Good response time and performance.
- Multiple end-user interface designs reflecting different end-user roles.
- A high performance relational database system, the DMS database system (DDB).

To provide separate tool sets for each role identified in SUPERMAN, DMS has been built to support two development facilities: a dialogue development facility, used by a human factors engineer, playing the role of dialogue author, and a computational development facility, used by software engineers. Figure 6.2.1 (adapted from [EHRIC81]) illustrates this dual architecture.

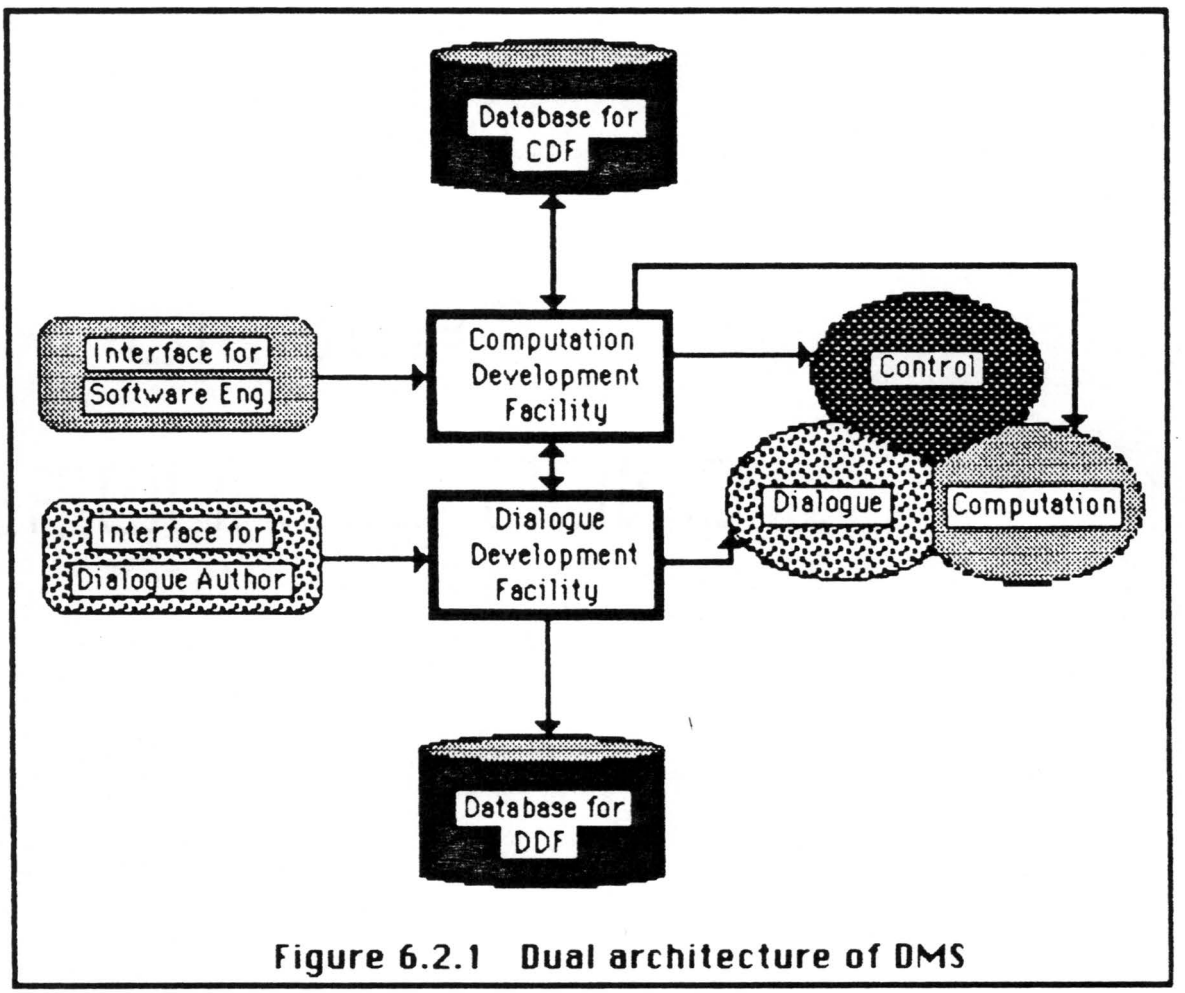


Figure 6.2.1 Dual architecture of DMS

The Dialogue Development Facility, commonly called the Author's Interactive Dialogue Environment (AIDE), consists of a set of very high-level, direct manipulation tools which "facilitate dialogue design, implementation, testing, and evaluation without writing source programs [JOHNS85]." Each tool supports a different class of interface design. These tools are depicted as the highest level of the AIDE architecture in Figure 6.2.2 (adapted from [JOHNS85]).

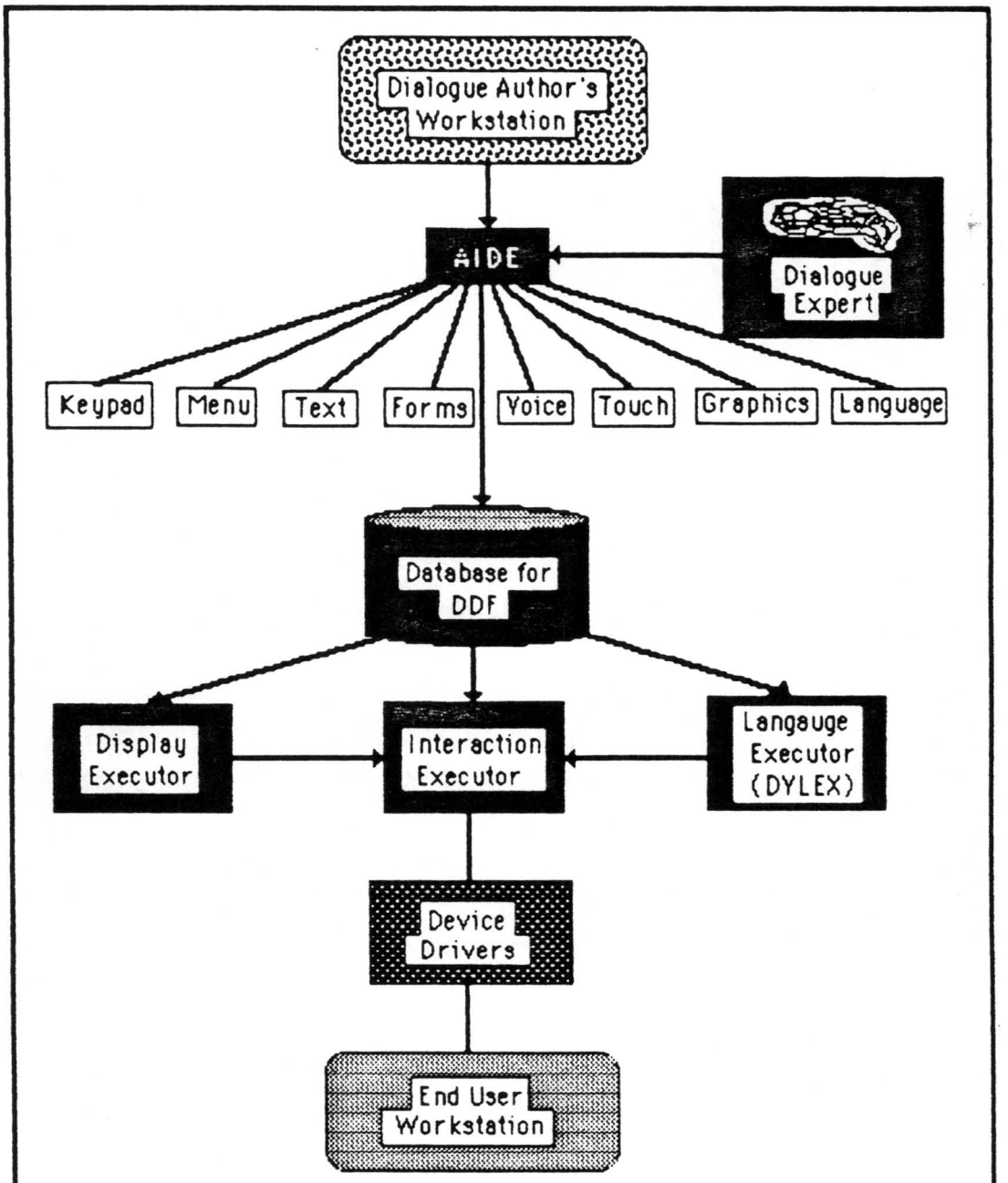


Figure 6.2.2 AIDE architecture

The Computational Development Facility supports the design, implementation, testing, and integration of the computational and control components of software systems. Since the CDF supports SUPERMAN, it uses the graphical notation in SUPERMAN as a Graphical Programming Language (GPL) and is supported by a GPL editor, a GPL automatic code generation tool, and a rapid prototyping and management support tool called the **Behavioral Demonstrator.** Figure 6.2.3 illustrates the architecture of the CDF environment.

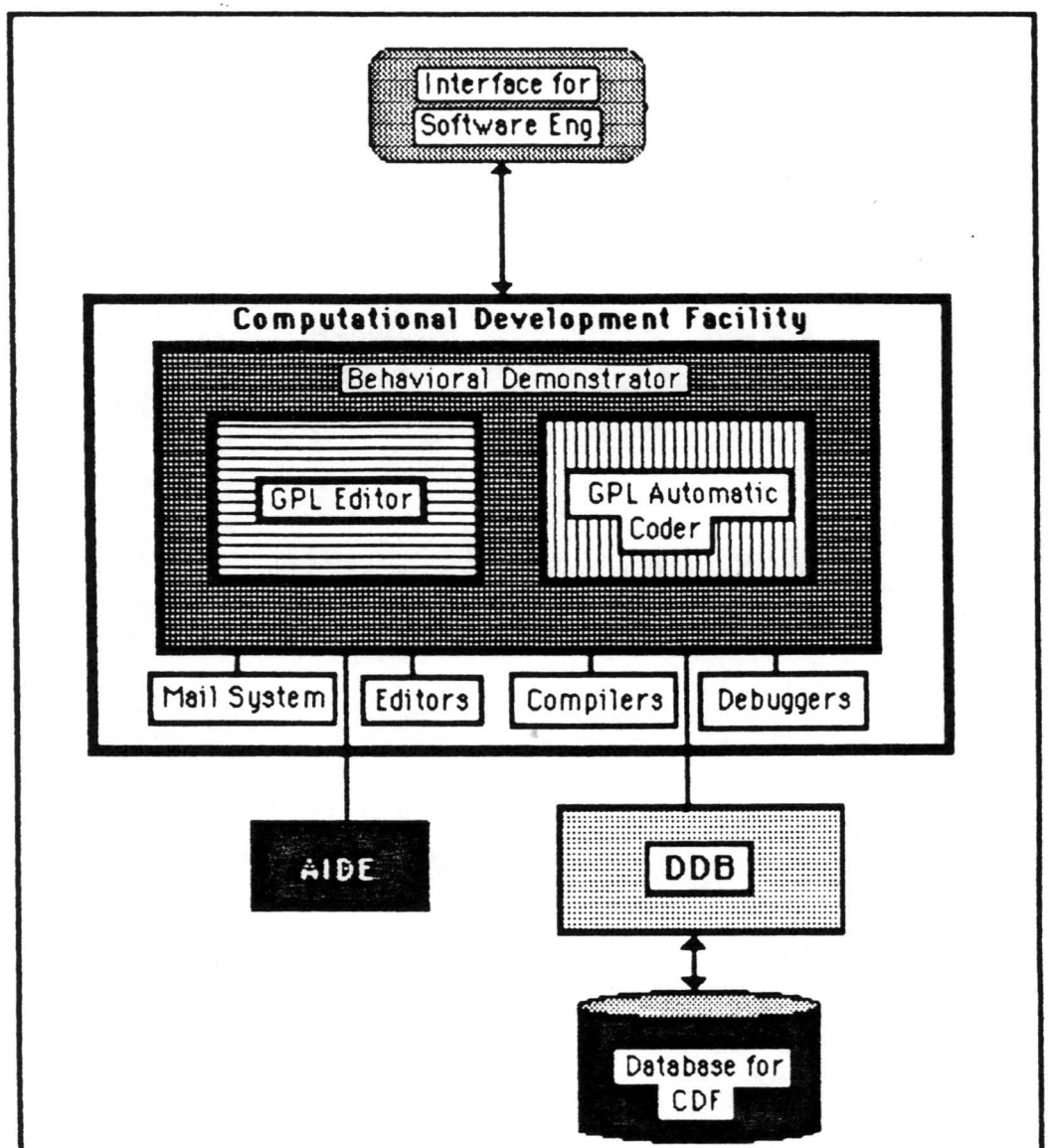


Figure 6.2.3 Computational Development Facility

DMS supports the separation of dialogue, computational, and control components during both design-time, behavioral demonstration, and run-time. During design-time, each component may be considered separate and uncoupled from the others. During rapid prototyping, or **behavioral demonstration**, dialogue is loosely coupled with the control component and "run" as a separate process from the computational component. During run-time, dialogue is separated, and the computational component is coupled with the control component. These different phases of development are illustrated in Figure 6.2.4.

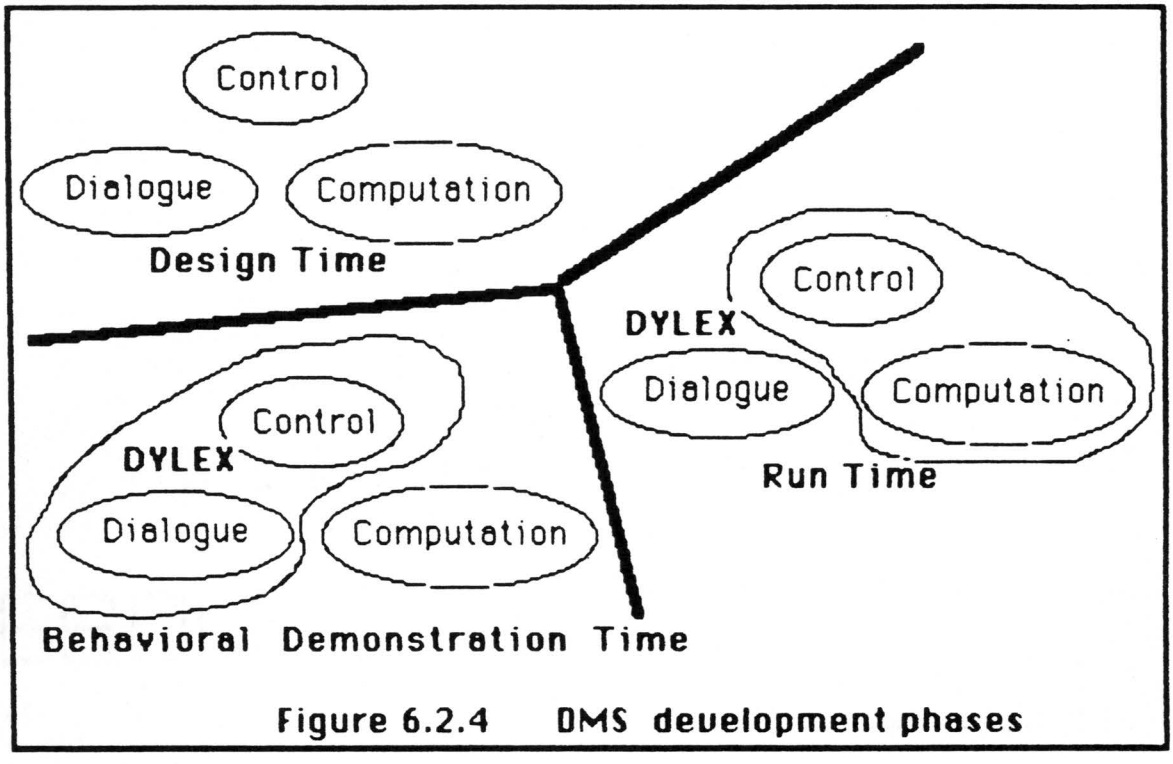


Figure 6.2.4 DMS development phases

7.0 THE BEHAVIORAL DEMONSTRATOR (BD)

This thesis has in previous chapters motivated prototyping as a necessary phase in the life-cycle, presented and evaluated previous work in rapid prototyping, promoted an automated paradigm which uses operational specifications as a replacement for the current life-cycle paradigm, and described SUPERMAN and DMS. This chapter details the history and development of a rapid prototyping tool, the behavioral demonstrator, which is the research contribution of this thesis. The behavioral demonstrator supports:

- Demonstration of system behavior at any stage of development **(behavioral demonstration NOT simulation)**
- Software cost estimation
- Automatic code generation
- Project-personnel management
- Interactive global control testing and refinement, independent of computational and dialogue components
- Data flow tracking
- Control flow tracking
- Interactive test case generation for modules
- Management of design decisions, project progress, and time constraints
- Top-down (as well as bottom-up) testing, refinement, and execution of system requirements and specifications.

7.1 BD PROTOTYPE (1983)

In 1982, behavioral demonstration was a new notion awaiting refinement. The idea was to use SUPERMAN's structuring capabilities as a structuring facility for requirements specifications. The BD was to be a system demonstrator, not a system simulator; simulations cannot evolve into the real system and must be discarded later. It was forseen to be the most important tool in all of DMS. Requirements for the behavioral demonstrator in 1982 included:

- Allows users to take systems "for a spin" early in the life cycle
- Interfaces with end-users to capture human factor suggesting for dialogue components
- Avoids mapping model of system to the real system
- Supports requirements specification evolution directly to the final system
- Tracks system design throughout the life cycle
- Supports the GPL editor directly.

In the Spring of 1983, work began on refining the requirements for the behavioral demonstrator so that a first version could be implemented. The concept of an automated life-cycle paradigm had not been anticipated, so the current life-cycle paradigm became a basis for the requirements which included [CALLA83]:

- The behavioral demonstrator will demonstrate the flow of data and control in systems.
- It will provide a communication forum for design teams during development.
- It will manage the progress of each component of systems.
- It will collect notes and progress reports for all functional components.
- It will enforce SUPERMAN.

Implementation of a behavioral demonstrator prototype began in May of 1983 and was completed shortly thereafter. The behavioral demonstrator was a push-down automaton [AHOAV78] which operated on graphical and textual data. Execution took place using a two screen architecture depicted in Figure 7.1.1. On a standard DEC VT100 terminal, the requirements specification for a node would appear, while on a DEC GIGI graphics terminal, the current SFD symbol would be highlighted. Users could traverse the **behavioral structure** (depicted in Figure 7.1.2) of systems by specifying the control path to take upon exiting individual modules. To trace data flow, data names would appear in a small window on the VT100. With this prototype, users could be given a sense of the system early in the life cycle, but this prototype could not truly demonstrate the dynamics of data flow, nor could it demonstrate modules that had already been completed. Requirements specifications were simply textual descriptions

having no structure and no means of discerning what nodes had been referenced in another node's requirements.

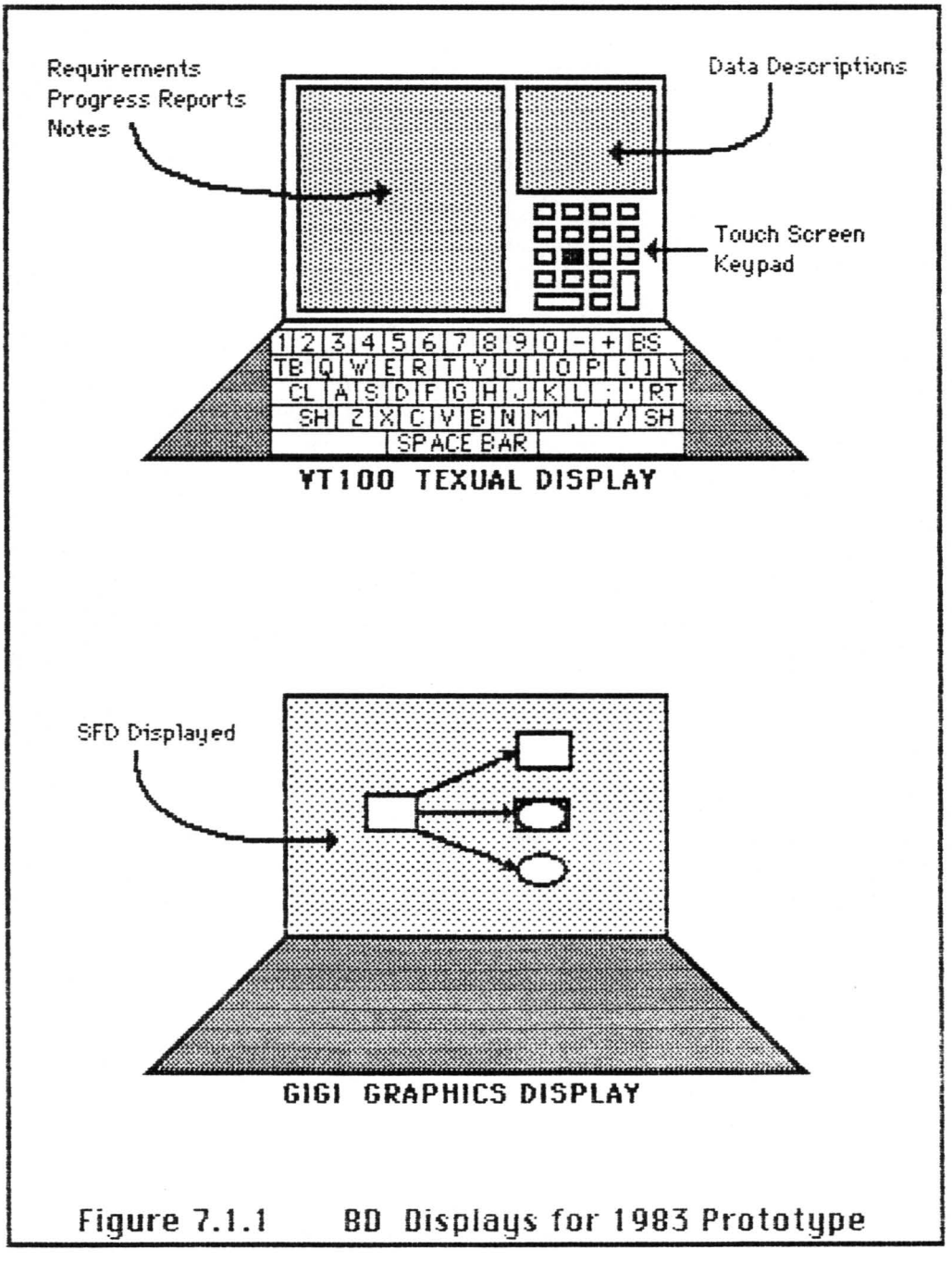


Figure 7.1.1 BD Displays for 1983 Prototype

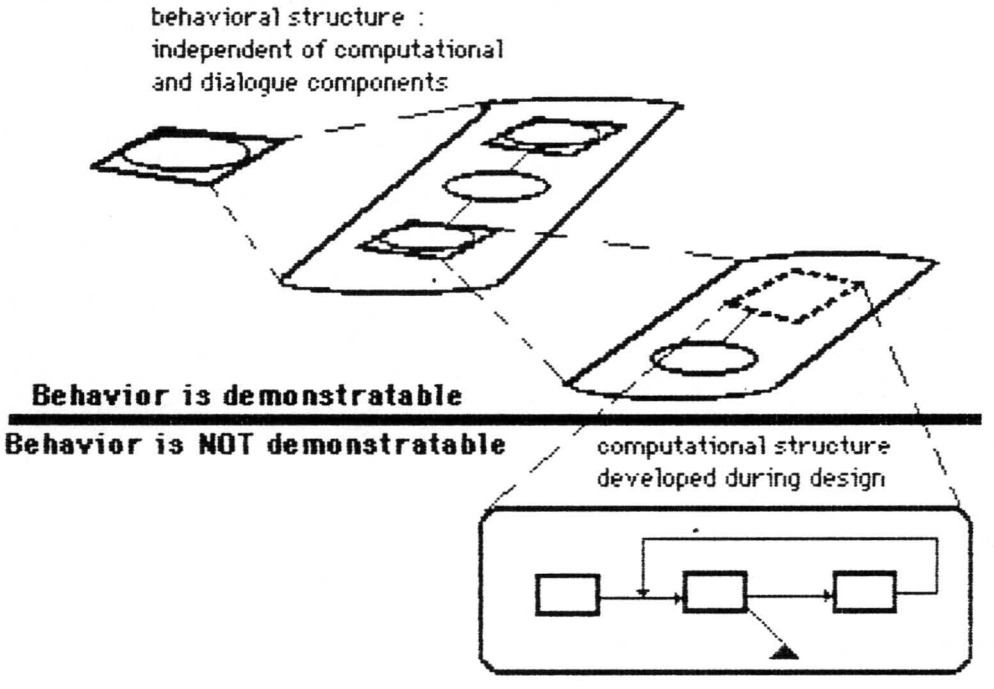


Figure 7.1.2 Behavioral Structure (1983).

Constructing the behavioral demonstrator prototype did indicate that operational specification was applicable to all system designs, not only embedded systems, and that behavioral demonstration should demonstrate every aspect of system designs during any phase of development. Also, it became clear that information necessary for system maintenance could be obtained and managed during system refinement and could be made available during behavioral demonstration. Restructuring of the behavioral demonstrator's logical data base would also allow software metrics, often obtainable only "after the fact," to be collected and maintained. Realization that many requirements for the behavioral demonstrator (see Chapter 4) were missing and others were incompletely specified only strengthened the notion that such a tool would be invaluable. Hence, work began on a new design that could achieve full behavioral demonstration.

7.2 BD (1985): A VIEW OF THE FUTURE

Chapter 4 presented attributes that were felt missing in the prototype constructed in 1983. Clearly, not all desirables anticipated for the future can be implemented in one tool with the current life-cycle paradigm, but with an automation-based paradigm and a new notion of exactly what requirements specifications are in the operational specification approach, many of them can become current technology.

Unfortunately, without a tool like the behavioral demonstrator or an operational specifications language it became difficult to refine, or even completely define requirements specifications for the behavioral demonstrator. Another means had to be developed for specifying enough of the behavioral demonstrator so that implementation could begin on a version of the tool that would allow future versions to be "boot-strapped" from older versions. Since the behavioral demonstrator is interactive, this method had to convey the user interface, system functionality, and system dynamics prior to implementation. The method decided upon was simply to construct paper scenarios, **behavioral templates,** that demonstrated dynamics but no environmental or performance constraints. A set of behavioral templates that were used to derive an operational specification for the behavioral demonstrator follow. These behavioral templates have been extracted from the design of a hypothetical, interactive airline reservation system.

As most system designs begin with a user wish list, the wish list and preliminary requirements for the airline reservation system follow:

1. The system must check the agent-id every time a person logs on and adjust agent-computer communication according to the agent's expertise level (eg., novice or expert).

2. The system must perform the following functions, each of which can be invoked independently by a terminal function button.

   a. Reservation

   b. Get-Payment

   c. Cancel-Reservation

   d. Show-User's-Reservation-Status

3. The system has two major logical record structures. These records are shown in Figure 7.2.1.

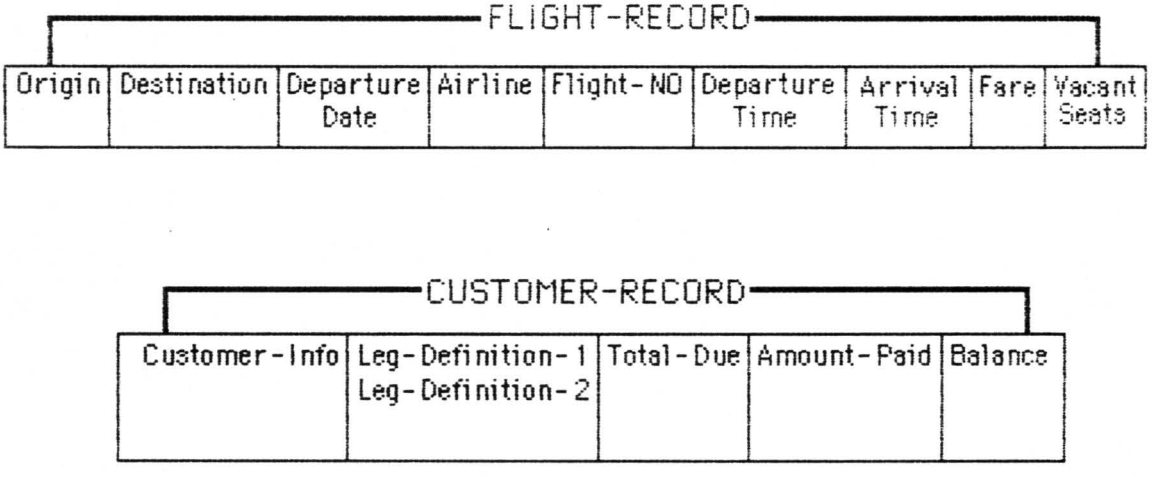


Figure 7.2.1 Airline System Logical Record Structures

In SUPERMAN, this wish list could be recorded as a supervisory structure of operational procedure diagrams and manual functions, which could later be refined into SFD's. The SFD specifications for the two top levels of the system are depicted in Figures 7.2.2 and 7.2.3. PERFORM AIRLINE RESERVATION meets requirements #1 and #2. As the specification slowly evolves into the true system design, the ability to demonstrate system behavior is needed. Suppose a user of the behavioral demonstrator wants to trace the control flow of the system. The scenario for this begins in Figure 7.2.4.

When the user selects the BD--> icon, behavioral demonstration begins by executing the first node in the SFD expansion for PERFORM AIRLINE RESERVATION. This is indicated by highlighting GET AGENT ID. Now since GET AGENT ID is itself a supervisor, PERFORM AIRLINE RESERVATION should be "stacked" and behavioral demonstration should continue with GET AGENT ID. This determines that the behavioral demonstrator should be implemented by a mechanism similar to a push-down automaton.

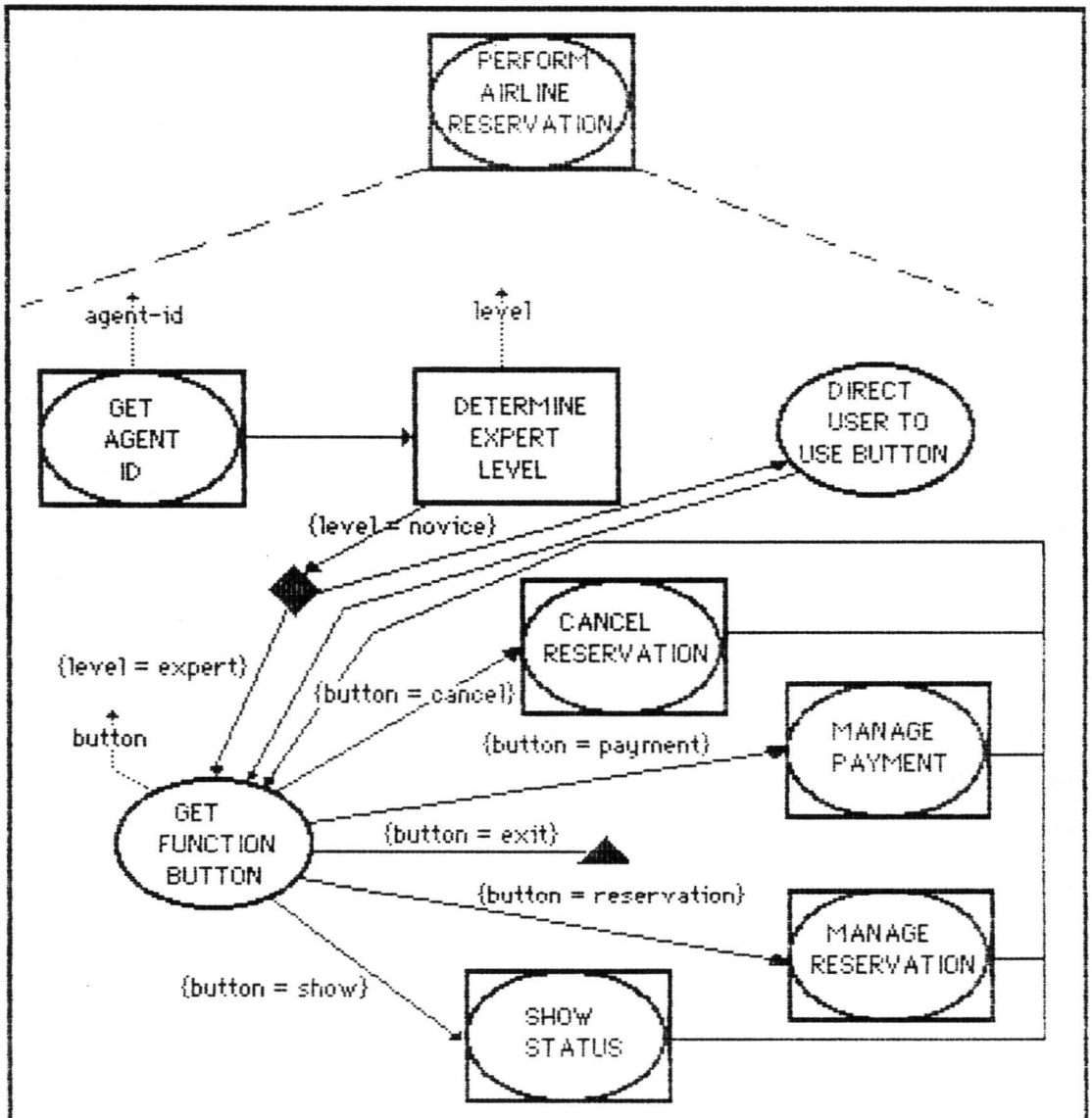


Figure 7.2.2 SFD for PERFORM-AIRLINE-RESERVATION

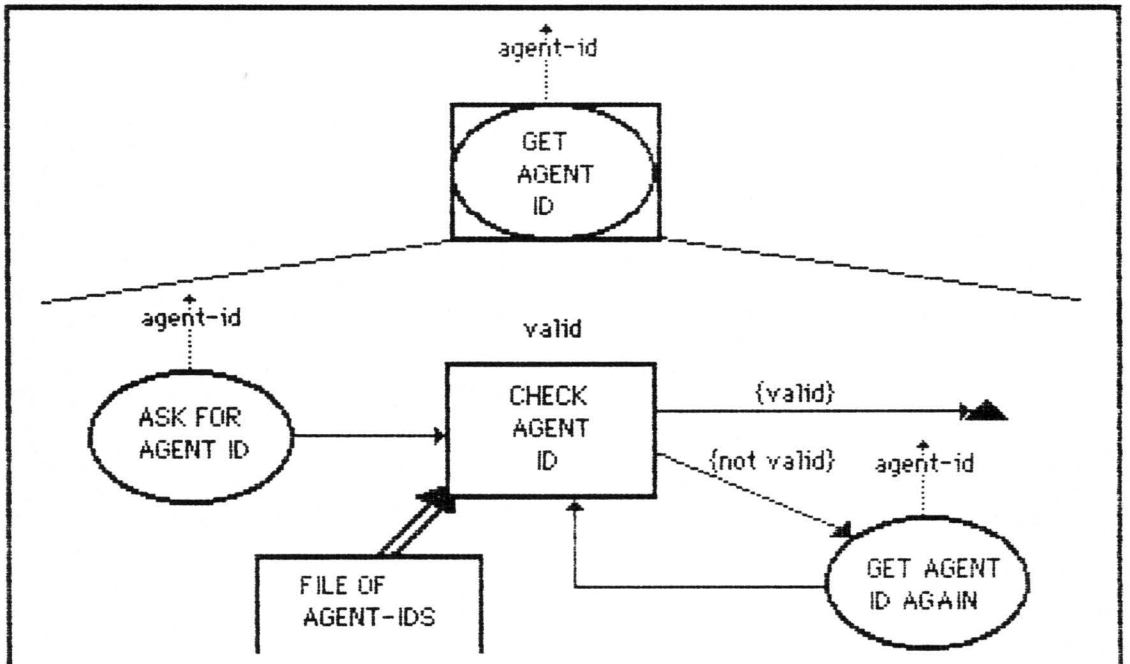
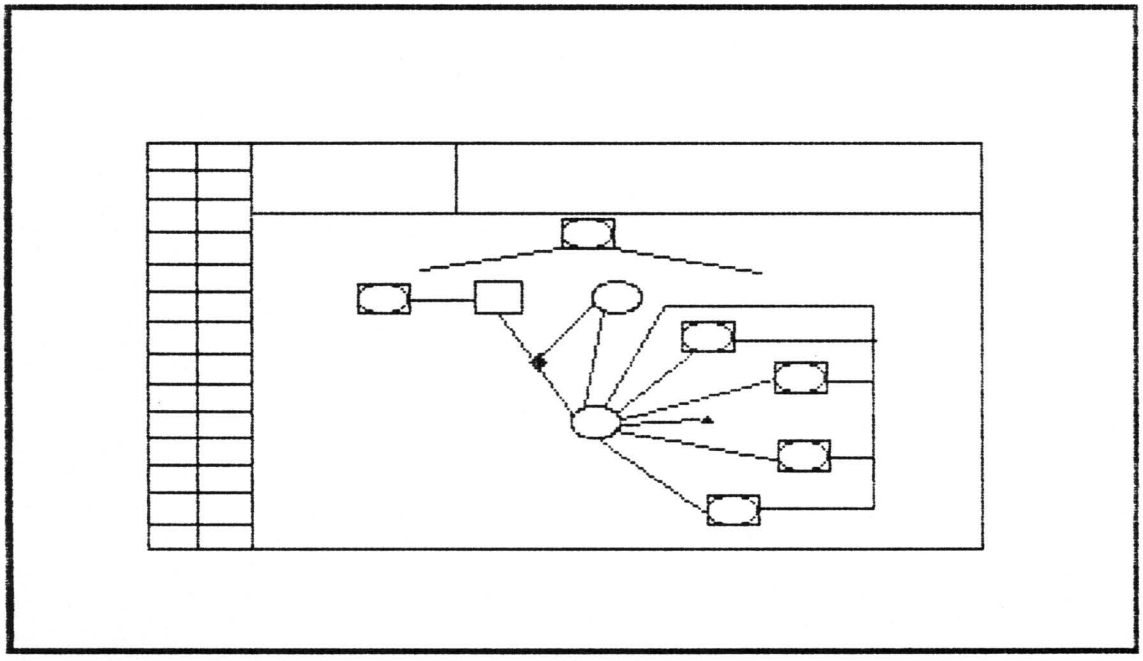


Figure 7.2.3 SFD for GET-AGENT-ID



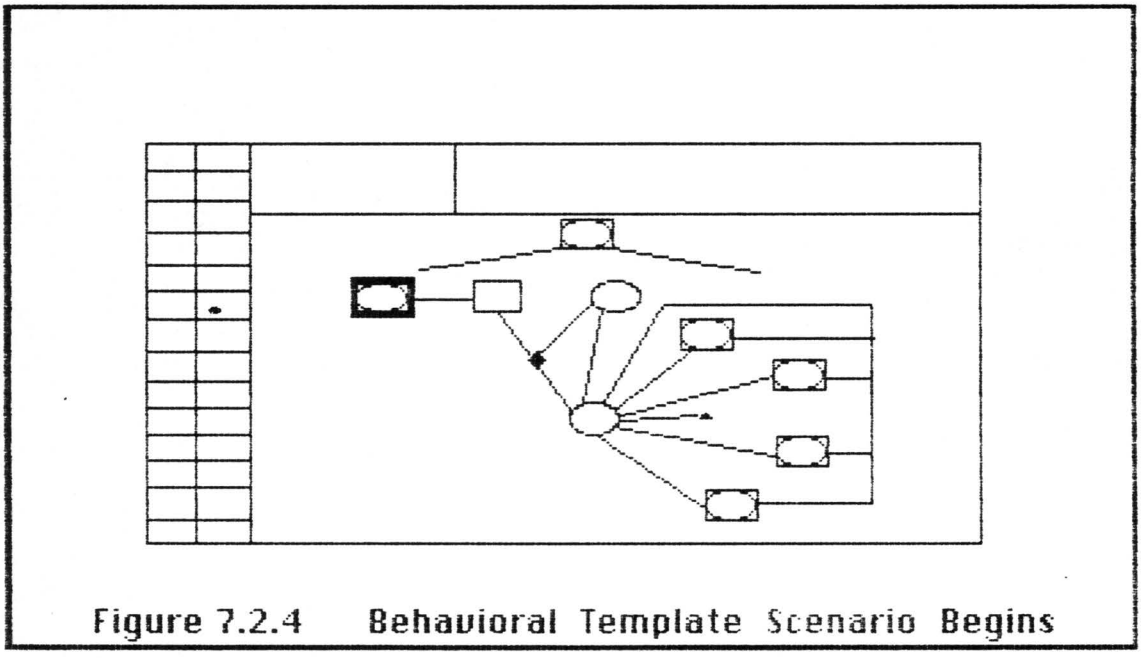
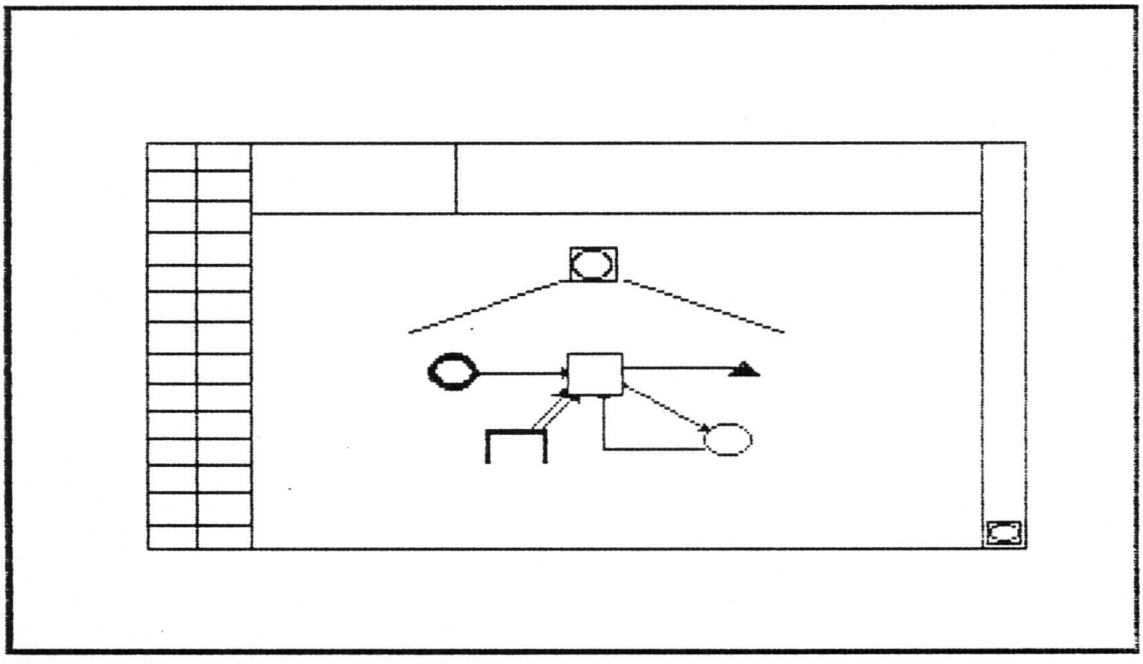


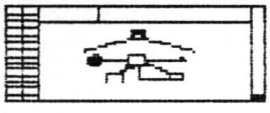
Figure 7.2.4 Behavioral Template Scenario Begins

Now the SFD expansion for GET AGENT ID should appear in the GPL window, and ASK FOR AGENT ID should be highlighted. Since ASK FOR AGENT ID is a dialogue worker function, it should be "executed." There are two cases for ASK FOR AGENT ID; either it has been completed and has been approved for system integration, or it is in a state of refinement. Both cases will next be considered.

In the case where ASK FOR AGENT ID needs refinement, all information currently known about this function should be made available to the user at a glance. This is accomplished by partitioning the behavioral demonstrator screen and windowing the details of ASK FOR AGENT ID in half of the screen. Notes can be added by selecting a notepad icon. This scenario is depicted in Figure 7.2.5.

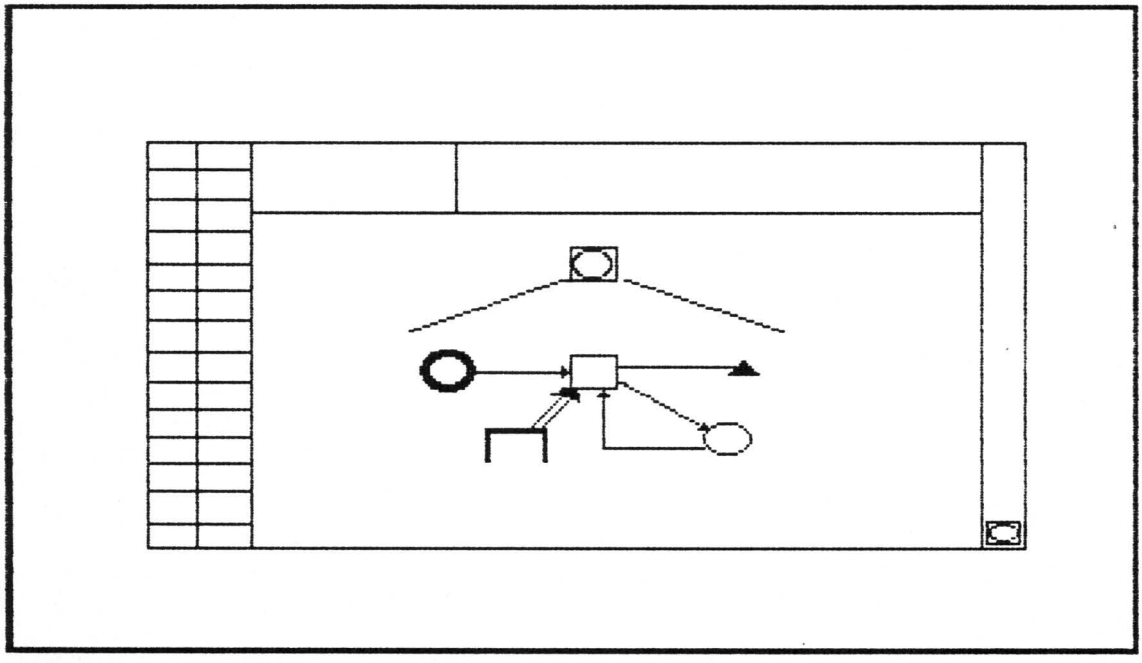
If ASK FOR AGENT ID, had been completely refined, the dialogue transaction should be executed by the transaction executor support tool in AIDE. The interaction would take place in the second screen partition. Since AIDE can be used to create a variety of user interfaces, this transaction could be a menu selection, a screen touch, voice input, a keyboard entry from the terminal, or one of many other variations. However the transaction had been designed, the user of the behavioral demonstrator will see the true behavior of the system as Figure 7.2.6 depicts.





Requirements and Specifications	Progress Report
Design Decisions	Timing Statistics

Figure 7.2.5 Behavioral demonstration for ASK FOR AGENT ID (Unrefined)



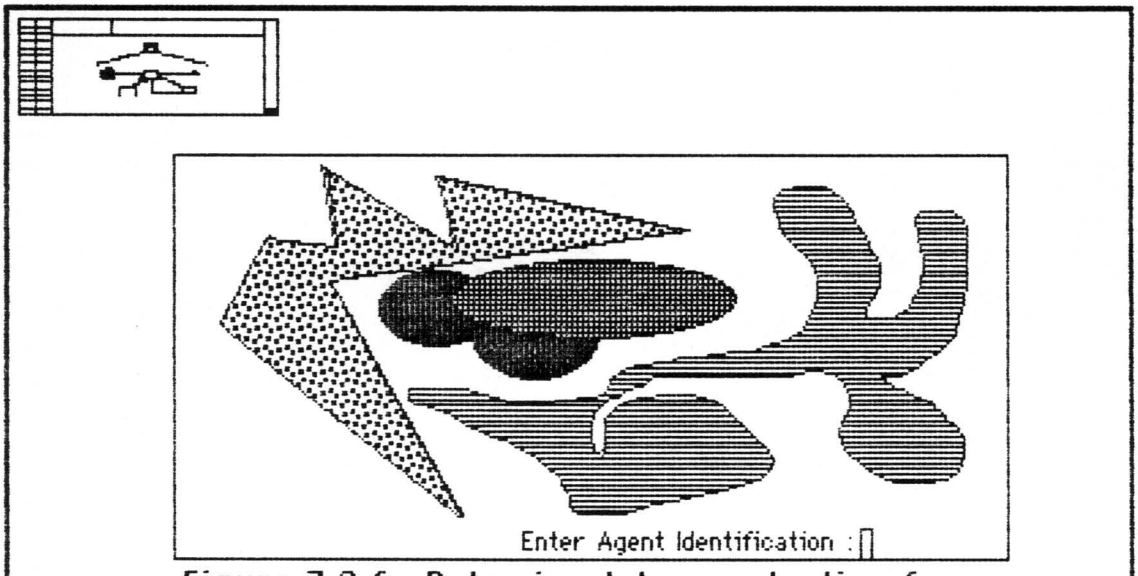


Figure 7.2.6 Behavioral demonstration for ASK FOR AGENT ID (Refined)

At this point demonstration should continue after an invocation of ASK FOR AGENT ID in GET AGENT ID, but it becomes clear that data input during the demonstration of ASK FOR AGENT ID should be available from this point forward in the demonstration. There are two possibilities for the variable AGENT-ID; either it has been declared using the graphical data description facilities of the GPL editor, or it has not. If it has been fully refined, the type for AGENT-ID is known and its value may be stored in a symbol table. This requires that the behavioral demonstrator will be more of an Augmented Transition Network, thus will have the power of a Turing Machine [WOODS70]. If it has not been refined, other components of the system may still require the data for behavioral demonstration to continue. The PAISLey project (described in Chapter 3) solved this problem by prompting the user for the necessary values. One problem with this approach is that in large systems users may not know a valid value for AGENT-ID. To solve this problem a means for establishing useful values for refined variables must be found. A value could be selected at random from a declared range, as done in PAISLey, but how is "random" defined for a data type like name? By extending the notion of data declarations to include data instances, this problem may be solved elegantly. This is analogous to Query-By-Example in the database literature [ZLOOF75].

To illustrate this concept, a few extended data definitions for PASCAL follow:

- Agent_id : name {'Tom Smith'};
- SSN : ssn_type {'344-85-4552'};
- Today : days {MONDAY};

If AGENT-ID had not been refined, its type may not be known. In this case the behavioral demonstrator could continue without the type, obtaining a value from the user, or a similar variable instance when AGENT-ID is used. After AGENT-ID has been instantiated, the screen could return to the previous state allowing the user to select the control path out of ASK FOR AGENT ID. Here there is only one path, and the behavioral demonstrator could choose it automatically.

Now CHECK AGENT ID is a computational function and can be demonstrated exactly like dialogue funtions if it too is unrefined. If, however, CHECK AGENT ID has been coded and approved for system integration, it too should execute. Herein lies one of the major problems of behavioral demonstration; how to bind environments dynamically to only partially completed code without interpreting computational functions or relying on incremental compilers. When the computational module executes, the data is windowed on the

behavioral demonstrator screen. Global data accessed by the module will also be displayed as shown in Figure 7.2.7.

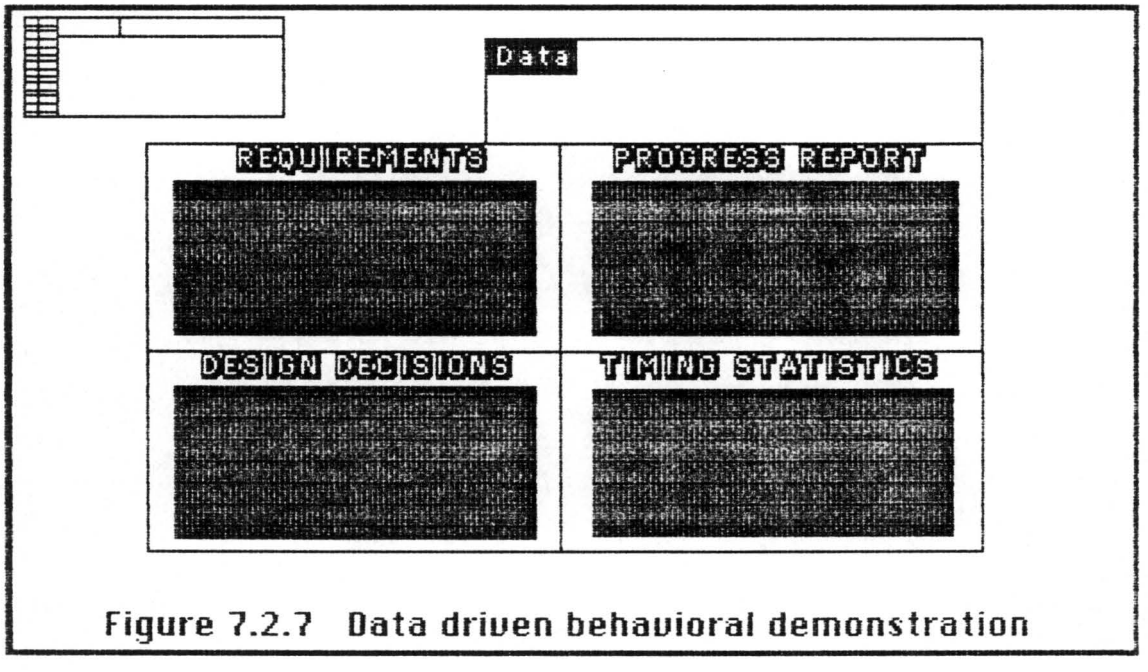


Figure 7.2.7 Data driven behavioral demonstration

Behavioral demonstration for data flow in systems is similar except that data values will determine navigation through the design. For exhaustive demonstration of small sections of a system, a hover capability has been defined, wherein data governs the flow of control within a constrained area of the design. Additionally, the behavioral demonstrator should be able to begin demonstration in any area of the design. These are but a few of the detailed specifications for the behavioral demonstrator, details of implementation have not been determined from using behavioral templates, but interesting problems that must be solved during implementation have surfaced:

- How to bind environments dynamically to only partially completed code.
- How to form example data from dynamic and linked structures.
- Deciding when to prompt for data and when to use declared examples.
- How to recover when computational modules abort.
- How to consistently display control based alternatives to the user.
- Representing environmental and incomplete data descriptions in a symbol table.
- Obtaining and using data values whose type may be unknown.
- Resolving conditionals whose predicates may be incomplete or undefined.
- Representing recursion's effects on data without its effect on control.

- How to represent concurrency during behavioral demonstration.

The next section details the design of the behavioral demonstrator, and how these problems, and others not anticipated, have been solved.

7.3 BD DESIGN

Intended "users" for the behavioral demonstrator include system analysts, programmers, project managers, human factors engineers, and end-users. To meet the requirement of a good "user" interface, the behavioral demonstrator was designed with a number of guidelines which Hanau and Lenorowitz (see Chapter 3) believe are important in a rapid prototyping environment. These guidelines are listed below [HANA80a]:

- System designers should be able to construct simulations without heavy dependence on persons with specialized knowledge.
- The environment should be a primary medium for communicating design ideas.
- The specification language in which system designs are expressed should be the same for demonstration purposes and should drive the simulation engine.
- The environments should encourage structure in designs and should provide a facility for investigating alternative designs.
- The simulation mechanism should support a variety of devices so that designers can experiment with the appear-

ance and behavioral characteristics of a number of devices.

- The environment should be general enough to ensure that the degree of achievable realism in a scenario is limited only by the completeness and accuracy of the specifications.
- Implementation of the production system should benefit directly from the efforts of early prototyping and scenario construction.

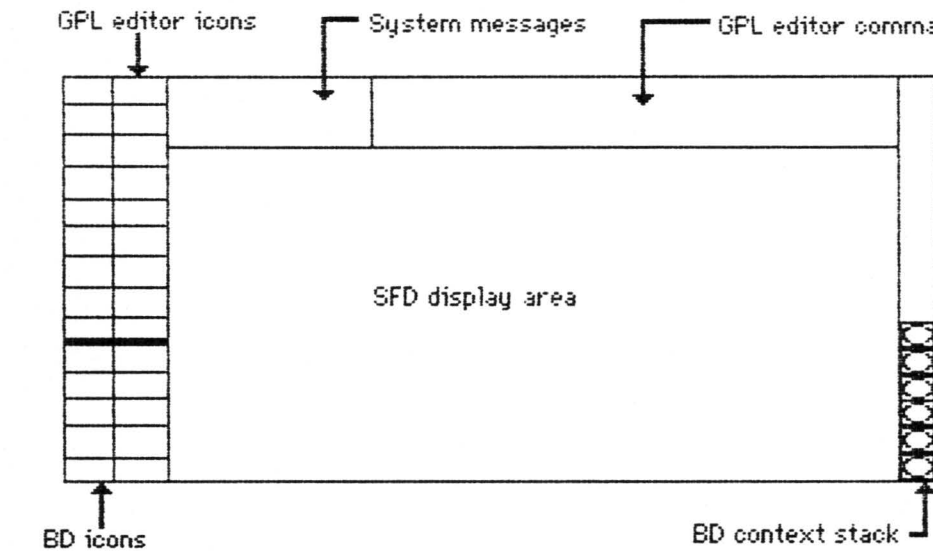
These and the future characteristics for prototypers described in chapter 4 provided a foundation upon which the behavioral demonstrator has been built.

7.3.1 Human-Interface

The behavioral templates of section 7.2 provided an incomplete operational specification for the BD (ie. Refined functions are to executed during behavioral demonstration, while unrefined functions are to have design information displayed as the "execution."). The requirements for the BD mandated a good human-interface which will now be detailed.

The BD supervises the behavior of the GPL editor, and as such provides the environment in which the editor must operate. For this reason, the editor bit-mapped display screen is divided into a number of regions, some supporting the editor, others supporting the BD. The extreme left margin of the screen is divided into two sets of icons which are

used to select various direct manipulation commands. The top set is used to perform editing, while the bottom set is used for selecting BD operations. The top of the editor screen is partitioned into two sections. One section displays system messages, and the other section displays additional editing commands. The rest of the screen is used to display and edit SFDs. All BD actions are selected by moving or selecting BD icons. These icons, their actions, and the screen are shown in Figure 7.3.1.



BD Tracing the data flow during behavioral demonstration.

Hovering over a single module for completeness and correctness tests.

BD Tracing control flow through system.

Recording and retrieving time stamps for modules.

Storing and retrieving design decisions, progress reports, and time-tables.

Recording notes for designers during behavioral demonstration.

Recording reminders during behavioral demonstration.

Zoom icon for expanding all information on a module.

Unzoom icon for hiding details of modules.

Figure 7.3.1 Behavioral demonstrator screen and icons

Control based behavioral demonstration allows developers to select the path taken through the behavioral structure. This mode is chosen by simply moving the control icon into the SFD window of the editor. Behavioral demonstration begins with the supervisor currently being edited. After demonstration of a function, developers are prompted to select the path to take from the function for the demonstration to continue. This is depicted in Figure 7.3.2.

Data based behavioral demonstration, selected by moving the data BD icon into the SFD window, allows users to enter data for variables and uses the data values to determine paths through the behavioral structure. Hovering is a data based behavioral demonstration that constrains demonstration to a limited number of functions in an SFD expansion. This is selected by first selecting a region in which to hover, then "flying" the helicopter icon into the selected region. This is depicted in Figure 7.3.3.

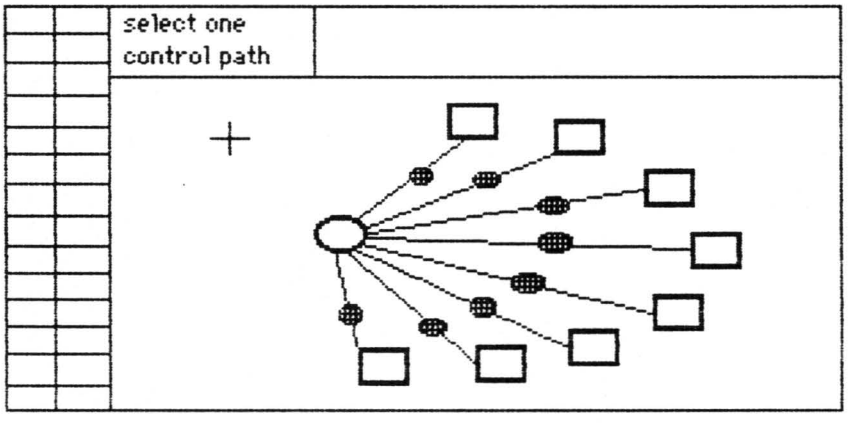
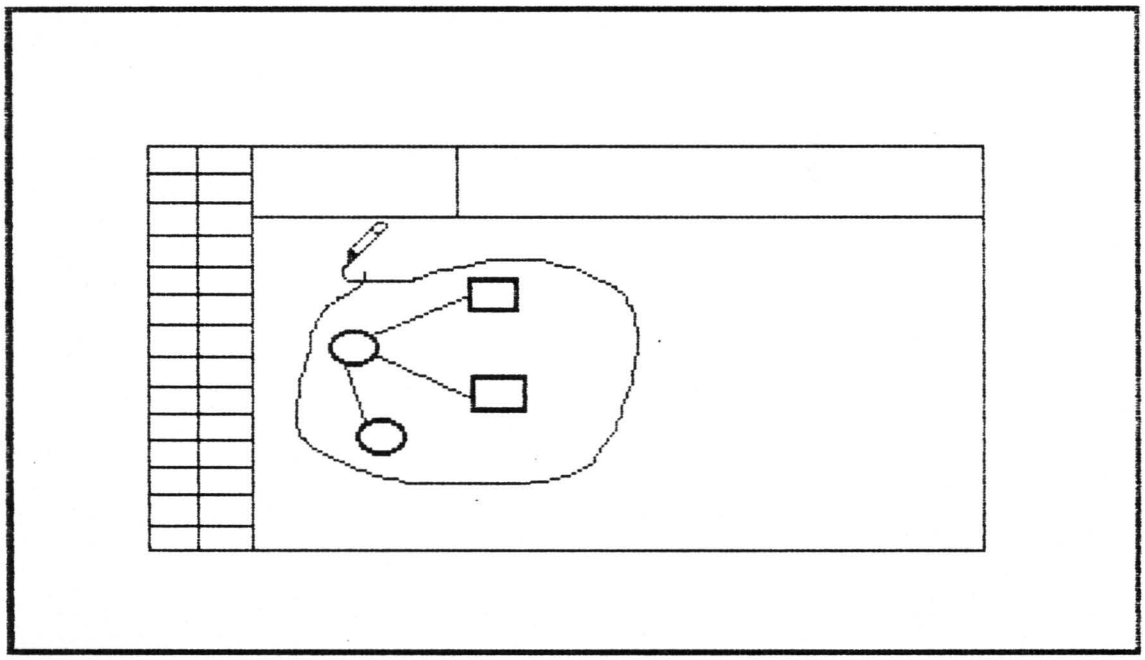


Figure 7.3.2 Control path selection



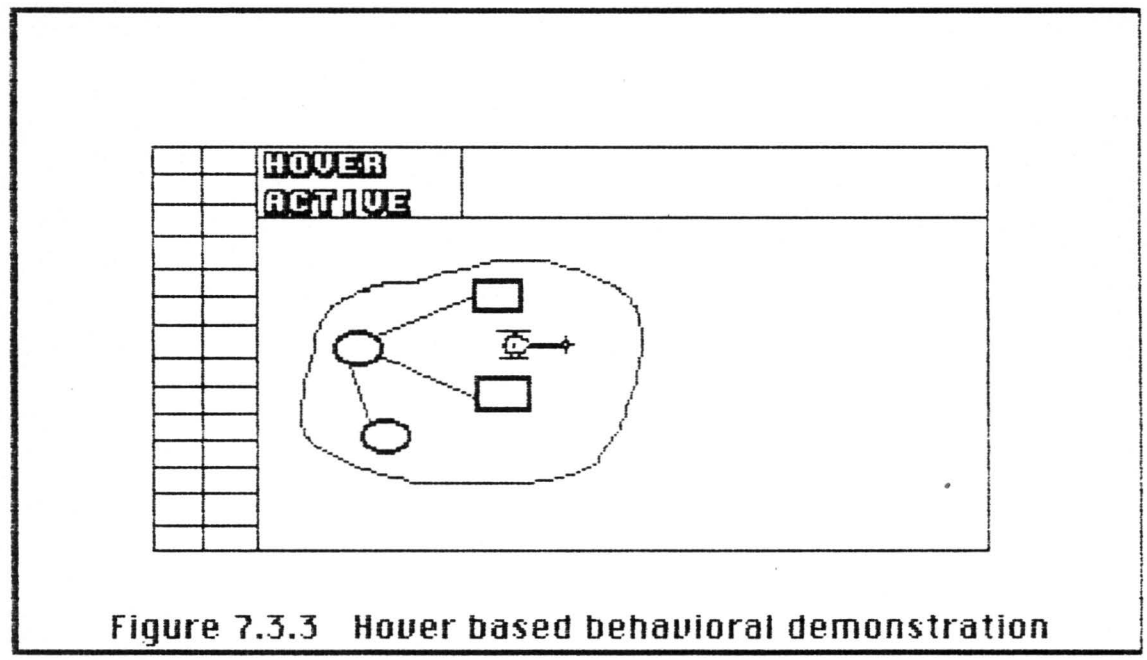


Figure 7.3.3 Hover based behavioral demonstration

Zoom may be selected for displaying information maintained about functions. The notepad icon allows developers to make notes and attach them to particular functions. The reminder icon allows developers to record notions that need to be consistently flashed at some point in the development of a particular function.

During any mode of behavioral demonstration, the screen maintains a stack which displays the level of behavioral structure that has been traversed. When a function is demonstrated, the editor screen is shrunk and a window opened for demonstration. Information displayed when unrefined functions are "executed" must convey all needed information about the function. The structure and content of this information is described below.

While unconstrained natural language designs may be useful for humans, they remain nearly impossible to process by machines. To constrain what information could be displayed about functions it became necessary to define information that is useful during the development of software systems. Some research efforts ([ROMAN84] and [LINGE80]) have proposed taxonomies of requirements and specifications and have proposed certain classification criteria which are considered critical to system development. These criteria, and those listed in Chapter 4 have been refined into a set

of six basic classes of information which, when ranked by perceived importance, include:

1. Requirements/Specifications
2. Project Status
3. Design Decisions
4. Critical Time Statistics
5. Notes and Comments
6. Development Reminders

The first four classes are considered **major** classes, the remaining two, **minor** classes. Each class can be considered to be a separate, yet related information space, where each space has been partitioned into a set of dimensions forming the basis of that space. During behavioral demonstration, the major classes appear in a window on the screen. This window is partitioned into four areas, one for each information space. Within each space, developers may choose which dimension they wish to access by using pull-down menus. The partitioned screen and dimension selection for requirements is shown in Figures 7.3.4-7.3.6. The dimensions for design decisions and project status are shown in Figure 7.3.7. Appendix B contains a complete description of each dimension.

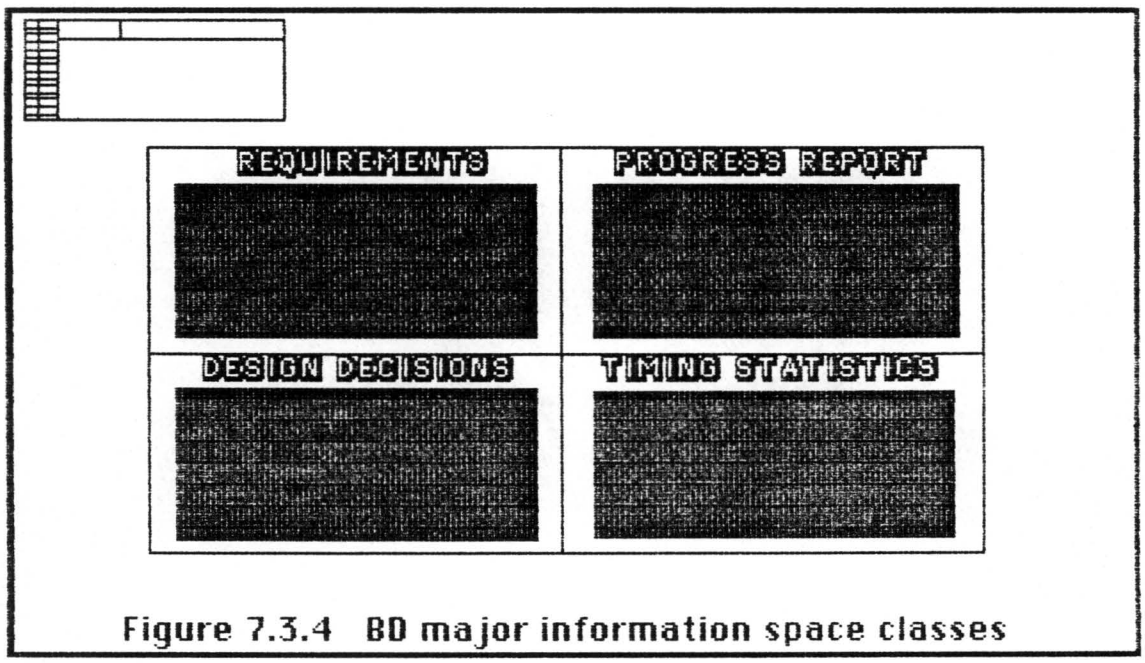


Figure 7.3.4 BD major information space classes

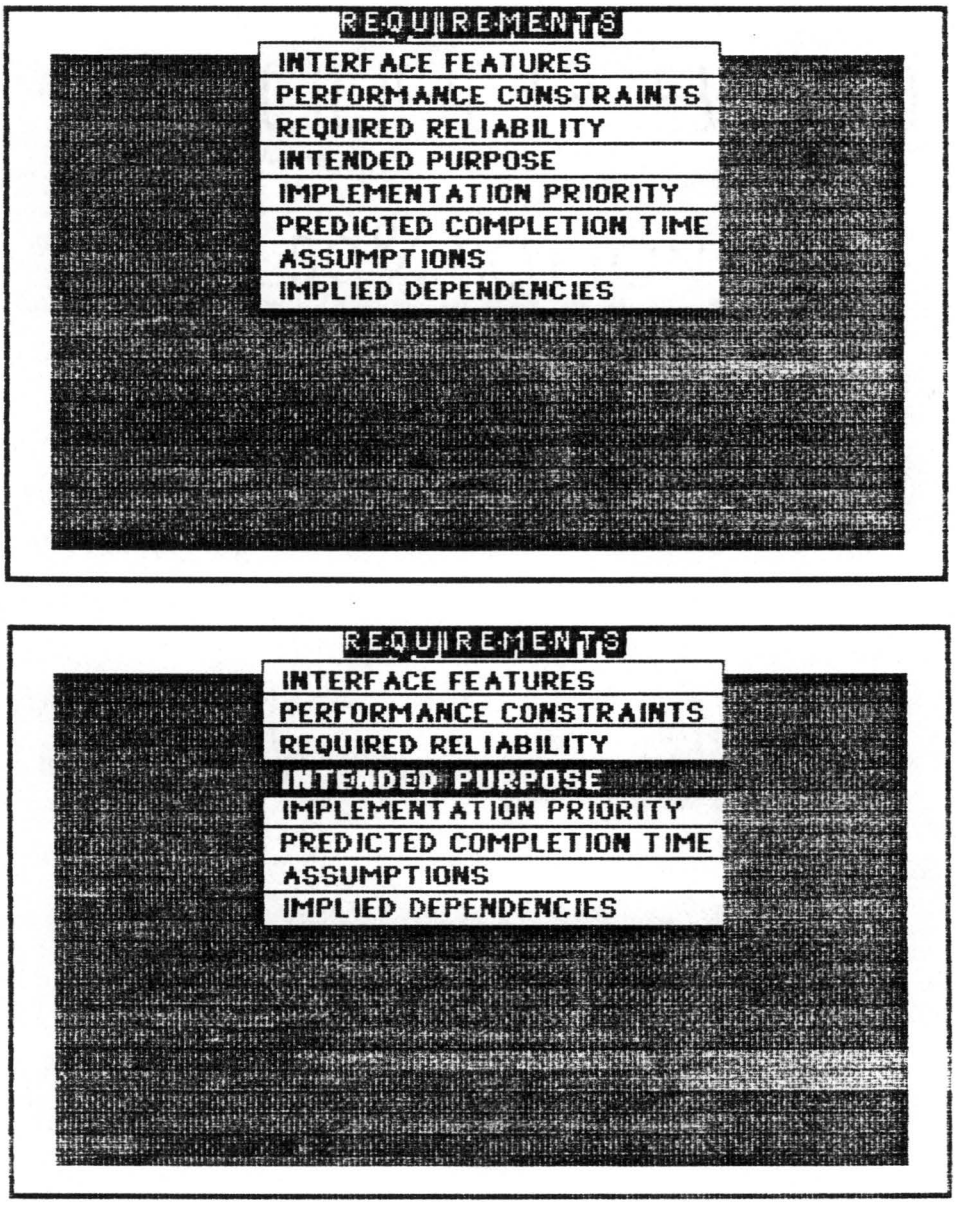


Figure 7.3.5 Dimensions are menu selected

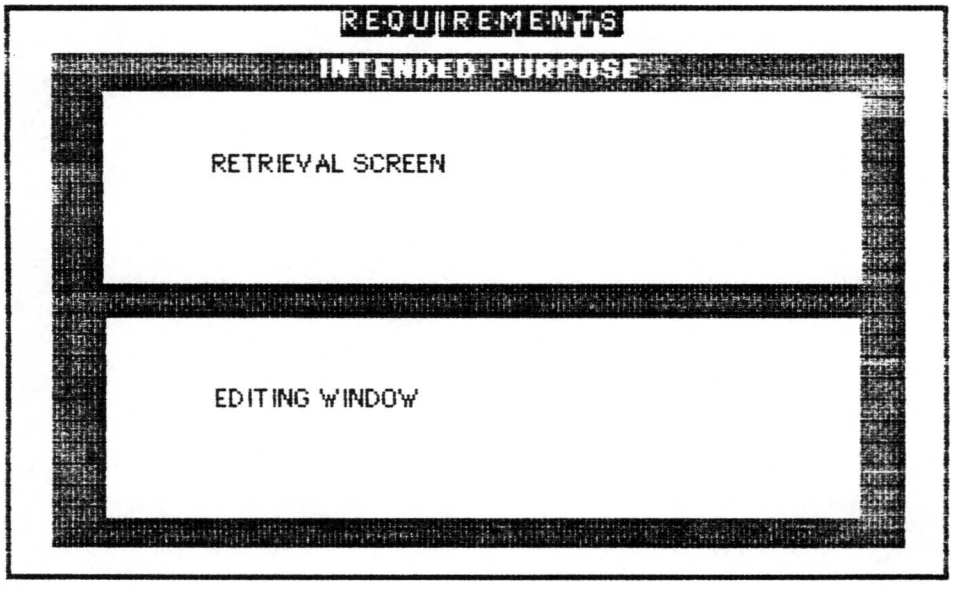


Figure 7.3.6 Dimensions are accessed using windows

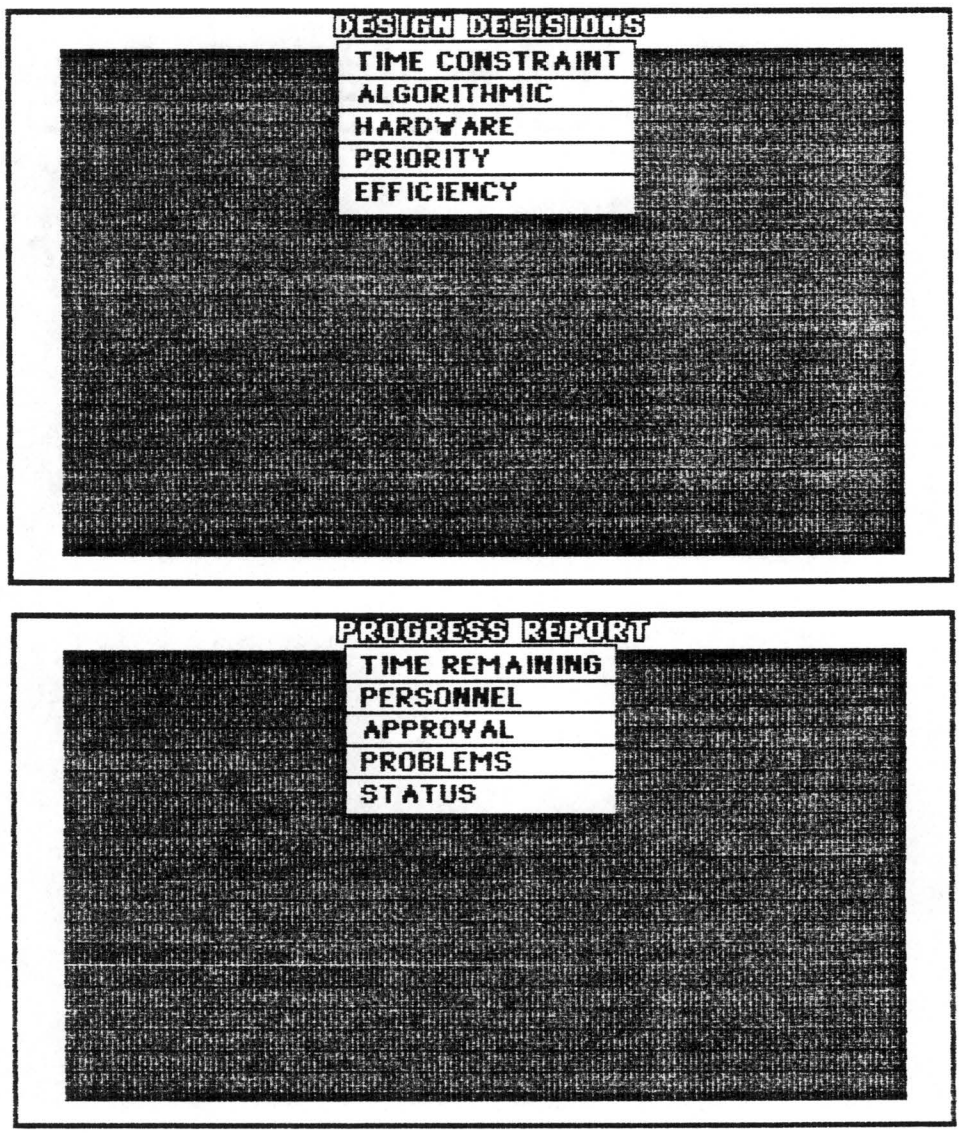


Figure 7.3.7 Dimensions for Design Decisions and Project Status

The last major-class information space, critical time statistics, may well be one of the most important for determining actual and predicted software cost and complexity of system designs [HENRY85]. This information space is a contribution of this thesis, and its dimensions have been defined in the context of this research. This information space will now be described.

As mentioned in Chapter 4, time-cost estimates are an important factor commonly used to determine software costs [MOHAN81]. Unfortunately, such estimates are often unavailable in any form other than a total accummulated time. Much finer granularities of time, if made available, could be used for determining software cost as well as software complexity measures [HENRY85]. The critical time statistics emerged as an information space used to collect and convey these finer granularities. Seven dimensions have been defined and form the basis for this space. These dimensions and their definitions follow:

Exposure reflects the time that developers have been exposed to a particular function in the SFD expansions of all other functions. Since function C may be referenced by functions A and B, any access by designers to functions A and B will add time measurements to the exposure of function C.

BD as a time statistic reflects the time that developers have used the BD to observe the behavior of a particular function.

Access reflects the time that developers have accessed the SFD for a particular function.

Transformation is the time spent by developers transforming the system into target versions for end-users. This statistic presumes the GPL coder tool is used to make the transformation. Also, it assumes that the first transformation marks the end of development and the beginning of maintenance and refinement for a system.

Refinement is the time spent modifying (editing) a function after the system has been transformed for the first time.

Design reflects the time developers have spent designing a particular function. Design begins when the first SFD symbol appears in the SFD expansion for a supervisor, when the first line of code is written for a computational worker, when the first interaction is developed for a dialogue worker, or

when the first bit in any information space owned by a function appears.

Execution reflects the time that a particular function takes to execute. In particular, this refers to computational functions, as similar metrics are collected in AIDE for dialogue functions.

For each time dimension, six statistics are gathered and made available to developers in CPU seconds and wall-clock time; they are:

- Total accumulated time
- Low time recorded
- High time recorded
- Mean time assuming a normal distribution
- Variance assuming a normal distribution
- Standard deviation assuming a normal distribution

These time statistics may be used with function fan-in and fan-out values (Fan-in is the number of references to a particular function, whereas fan-out are the functions referenced by a particular function.) to provide valuable structuring metrics for determining complexity in software systems ([HENRY79], [HENRY81], and [HENRY85]). In particular, Exposure is analogous to fan-in and Access is analogous

to fan-out and may be used to determine functions that are critical to system development. Refinement, Transformation, and BD may be used to evaluate the SUPERMAN methodology, the Computational Development Facility, and the Behavioral Demonstrator if similar statistics were available in other development environments using alternative development methodologies.

During behavioral demonstration, dimensions for the critical time statistics are selected by selecting the clock icon which labels the dimension desired. When a dimension has been selected, a bell-shaped curve, labelled with clock icons in the appropriate places, appears in a pop-up window. Individual statistics may be examined by selecting appropriate icons. The user interface for critical time statistics is shown in Figure 7.3.8.

The minor class of information spaces may be selected at any time during behavioral demonstration by selecting the appropriate icon for notes or reminders. These states are distinguished from those states in the major class through the user interface. Notes are placed on a notepad that is a large version of the notepad icon and reminders are entered similarly. Functions with nonempty minor spaces will be marked by notepad and/or reminder icons during editing sessions. These windows are illustrated in Figure 7.3.9.

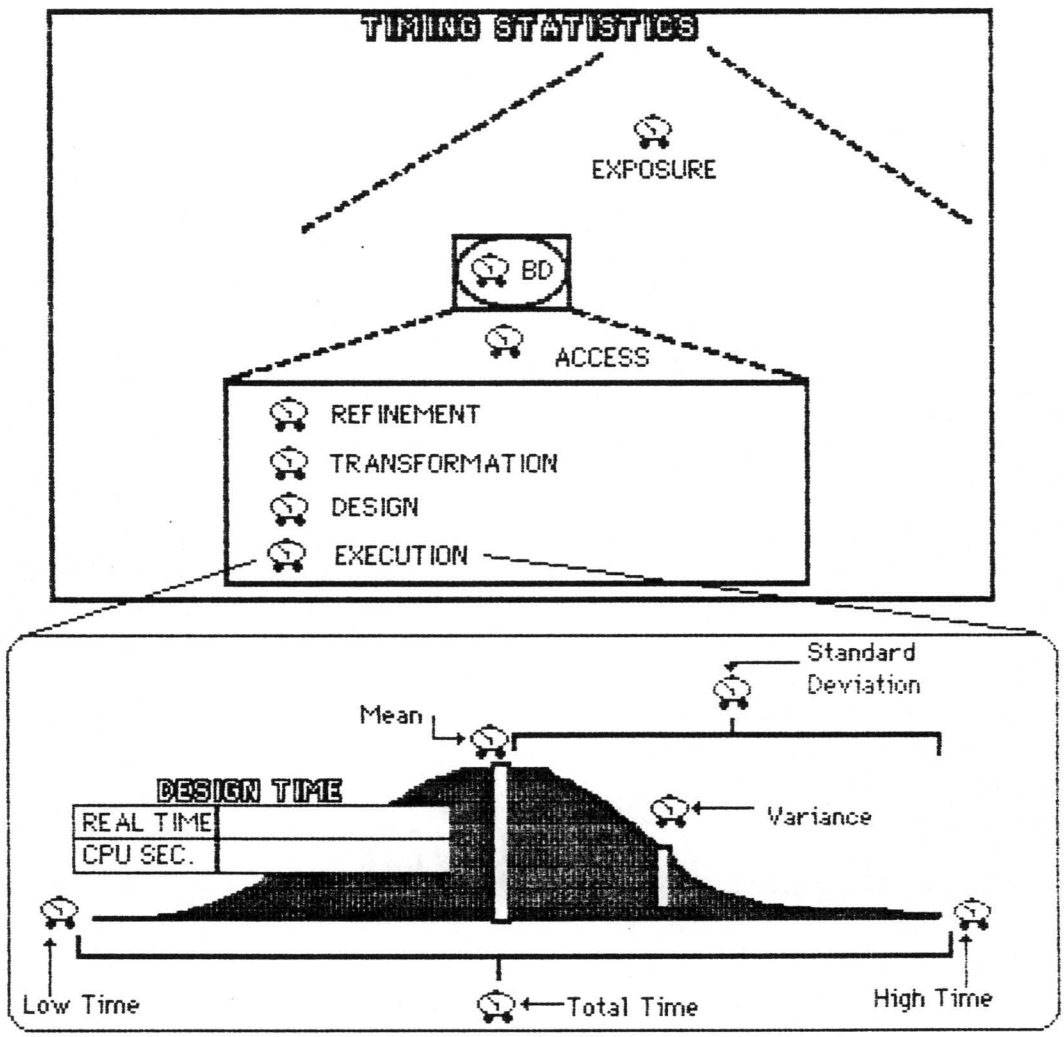


Figure 7.3.8 User interface for critical time statistics

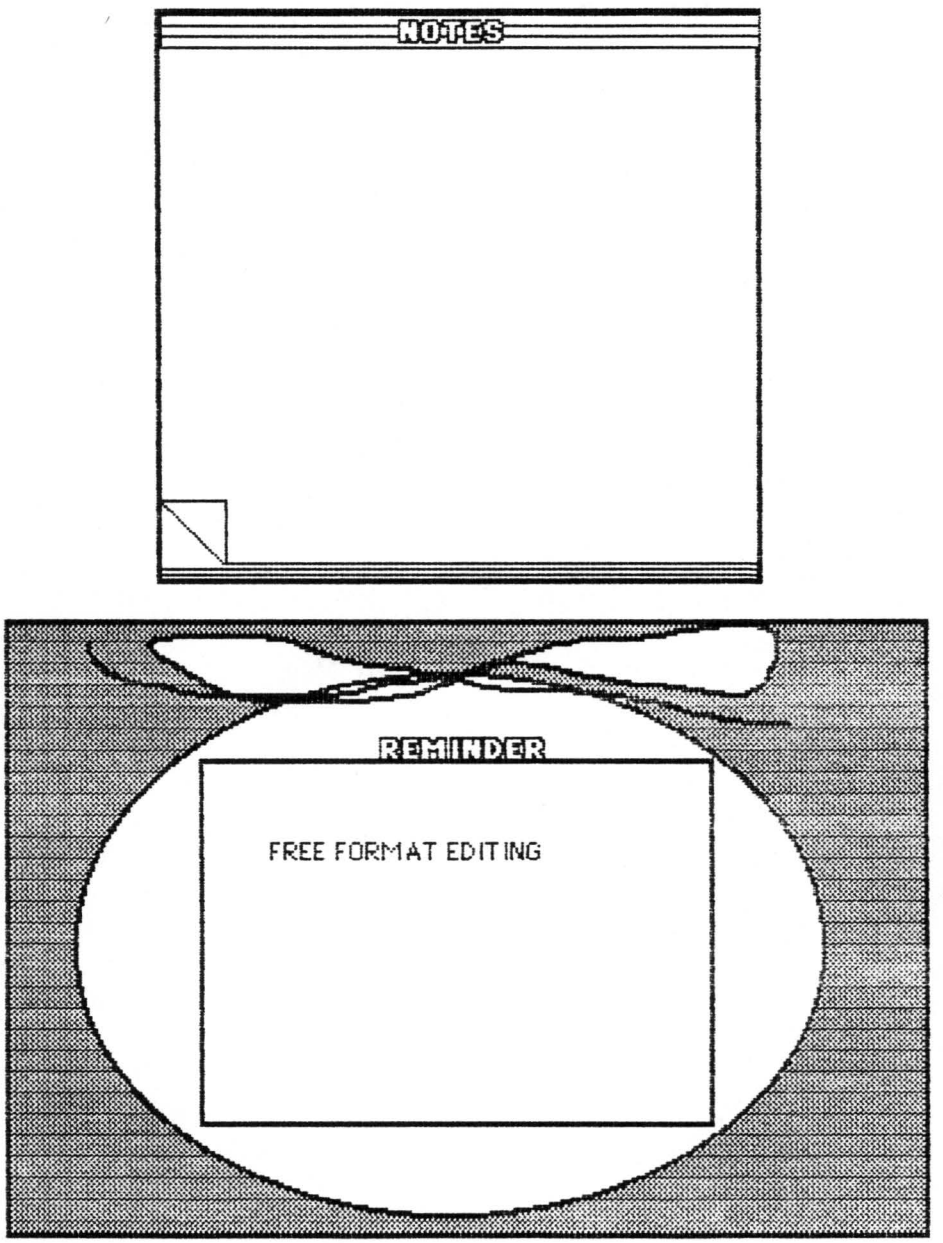


Figure 7.3.9 Minor information spaces' windows

7.3.2 Architecture

This section details the architecture and internal workings of the BD. This architecture has resulted from a number of important design decisions which will be discussed in section 7.3.3.

The BD accesses SFD definitions which are created by the GPL editor and stored in the CDF relational database. These data definitions contain all the information that is maintained by the CDF during system development, thus supports very complicated queries issued to the DBMS about system structure.

The BD is driven by an Augmented Transition Network (ATN) interpreter which is comprised of a finite state machine, a symbol table, and a context switching stack (see [AHOAV78]). During behavioral demonstration, the BD moves from state to state as it interprets the information contained in SFDs. State transitions may modify entries in the symbol table, or may cause the current context to be saved so that another state transition diagram can be executed. These state transitions are governed by the syntax and semantics for GPL, but may be interrupted by the existence of certain data contained in the information spaces for each function. These ATN's are shown in Figures 7.3.10-7.3.12.

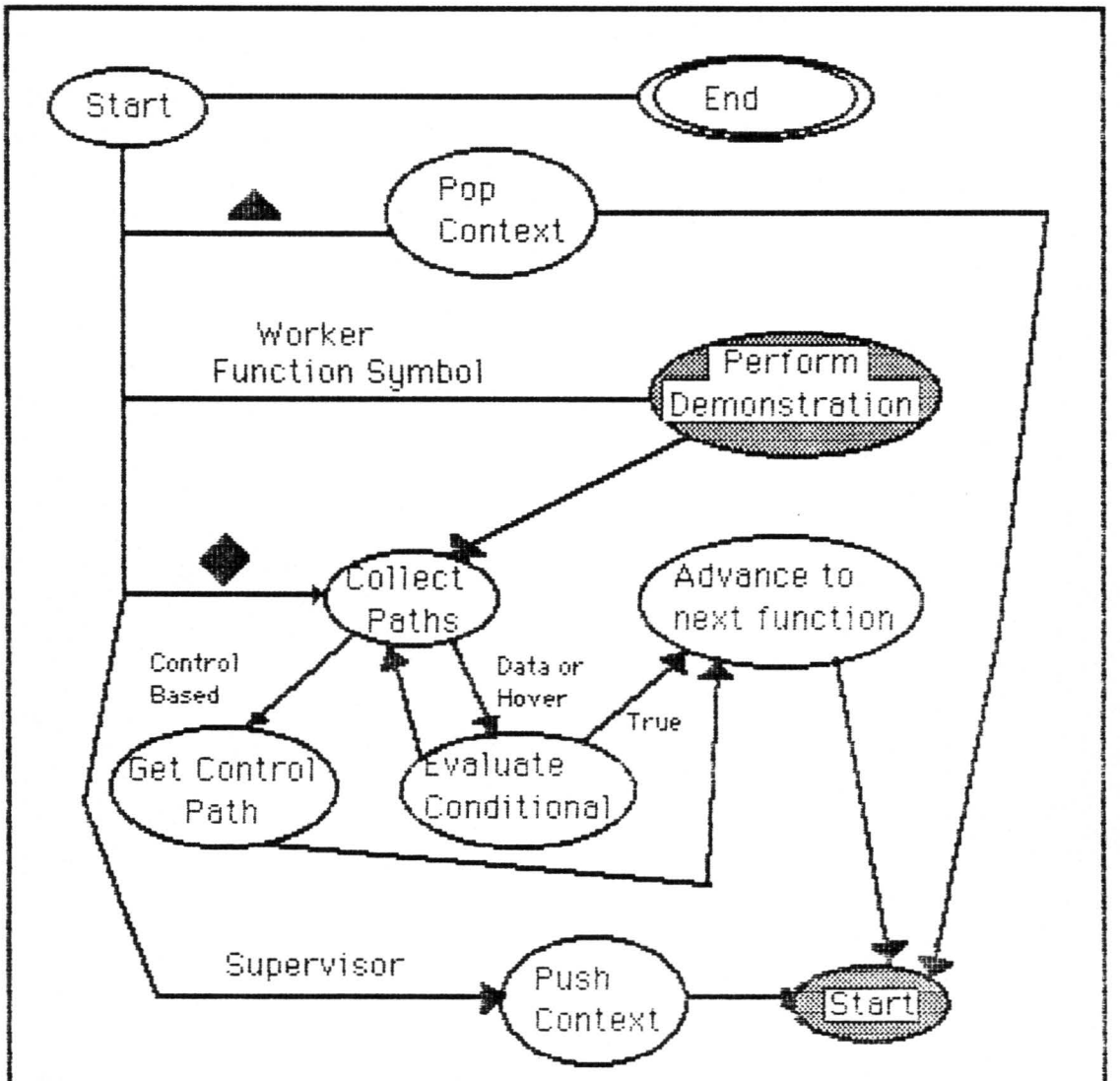


Figure 7.3.10 Top-most ATN for behavioral demonstration

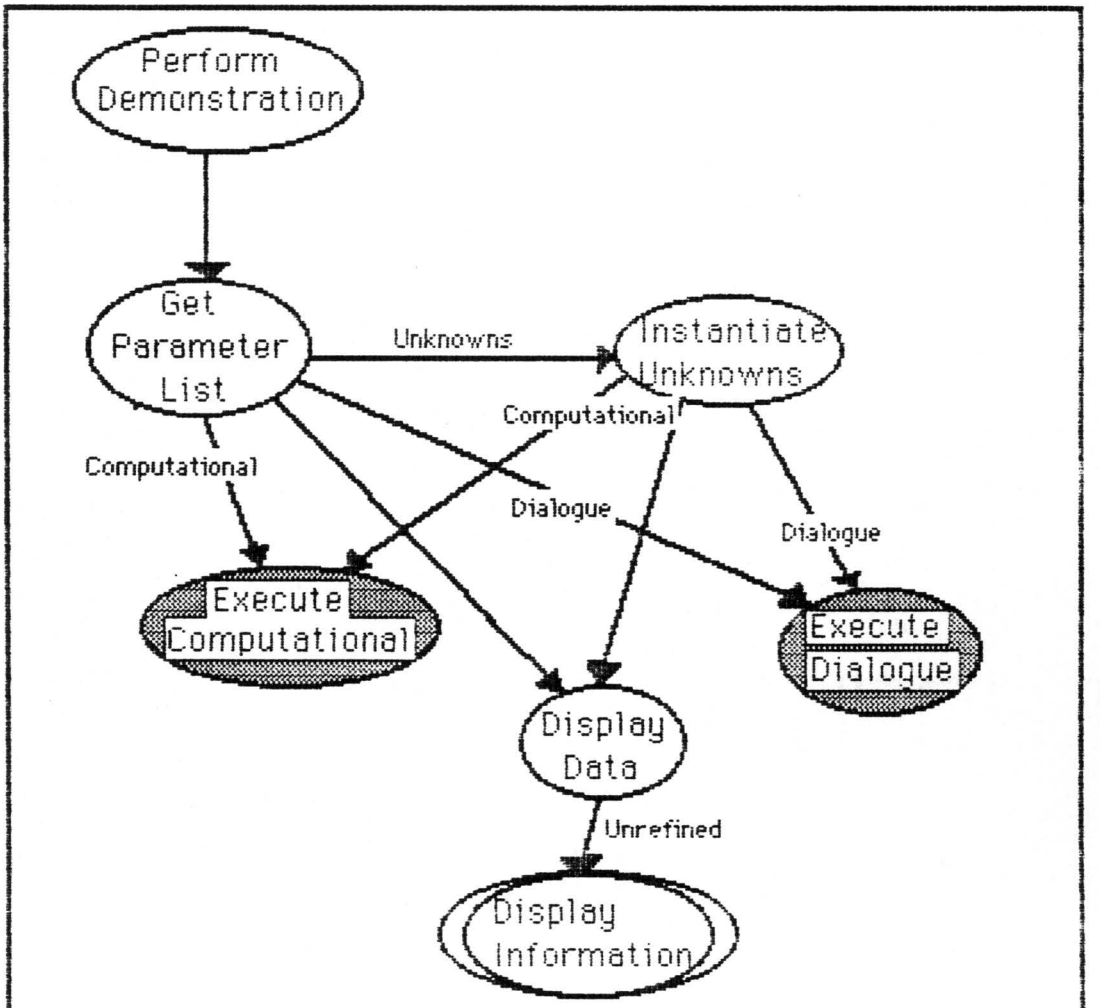


Figure 7.3.11 Undetailed ATN for Perform Demonstration

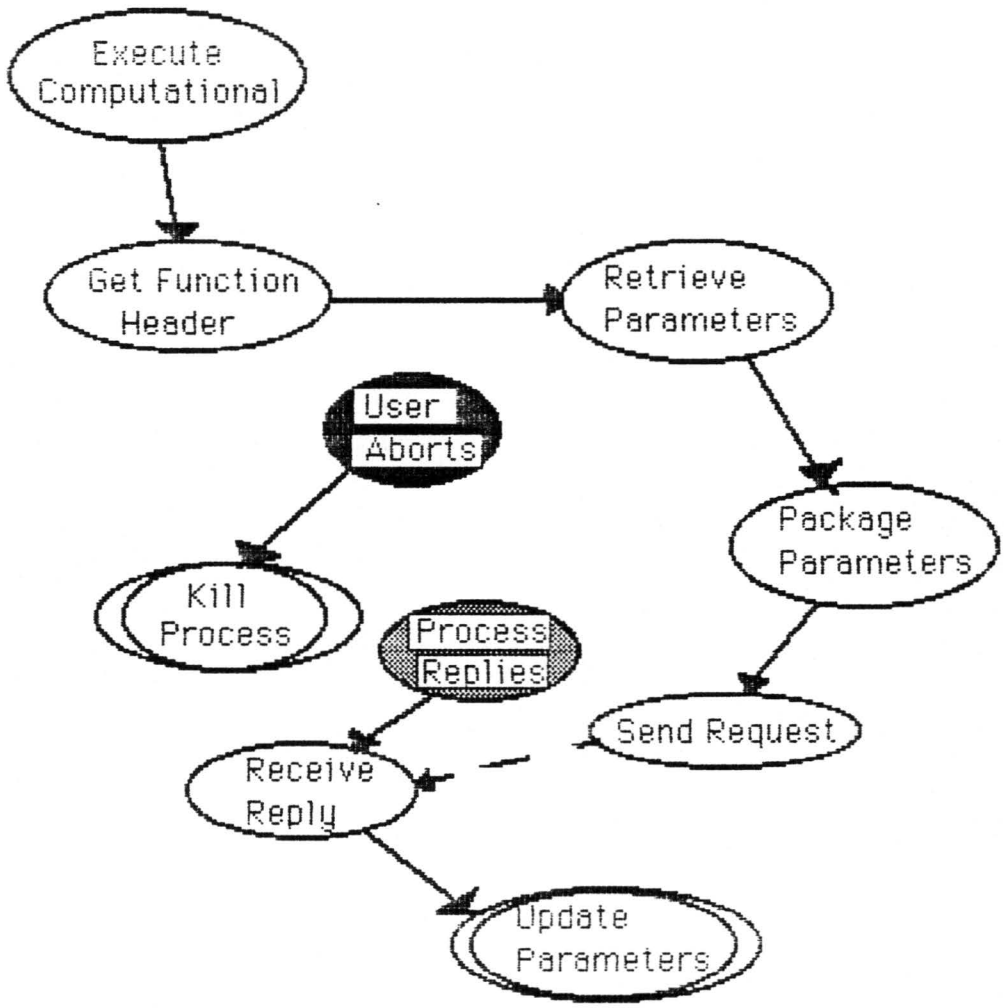


Figure 7.3.12 Simplified ATN for Execute Computational

Prior to behavioral demonstration, the BD must build an environment for refined computational functions and isolate this environment from the demonstrator so that its execution may be controlled and contained. To build and isolate the environment, the BD generates the code for binding together separate computational functions into a single program that runs in a separate process from the BD. This architecture for the BD is shown in Figure 7.3.13.

The generated program that forms the environment for the computational components can be divided into three sections: a type declaration section, a static variable initialization section, and a main loop section. The type declaration section simply makes type declarations that parameters passed in and out of the computational functions need. This is done by iteratively forming the closure over previous type declarations. The first set is defined to be all types composed exclusively of primitive types (eg. integer). The second set is defined to be all types composed exclusively of types defined in the first set. Subsequent sets continue building on the previous set until all type declarations have been made. This declaration scheme works in every case except for pointer types and self-referential structures. This problem is resolved by defining a pointer to any type to be a primitive type.

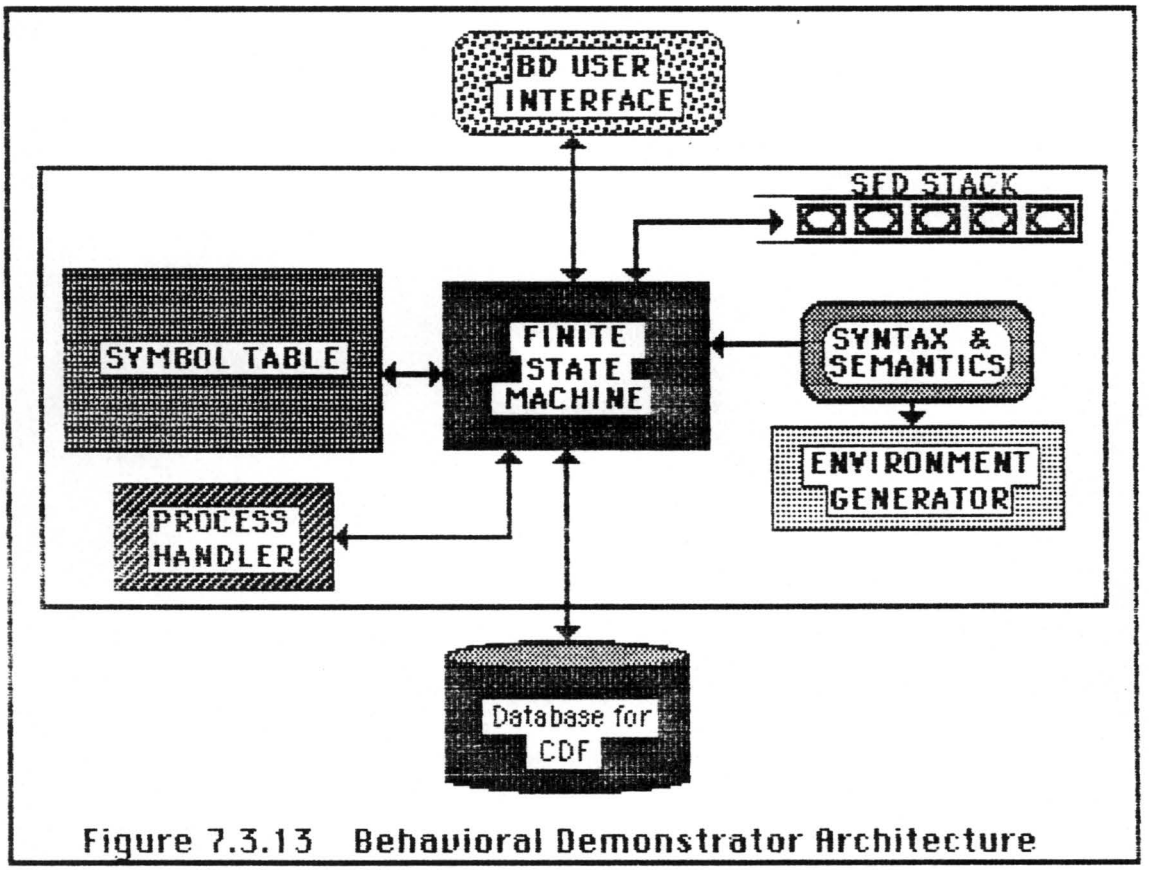
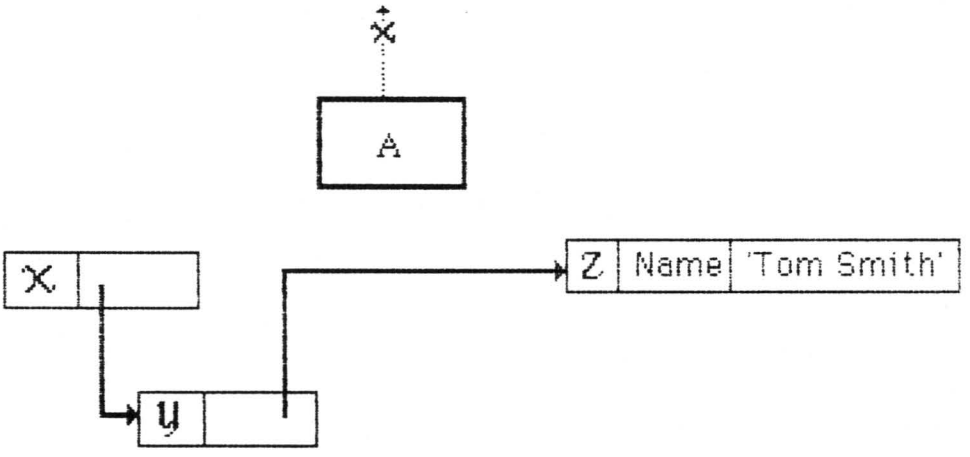


Figure 7.3.13 Behavioral Demonstrator Architecture

Pointers in the parameter lists for computational modules formed an interesting problem that had to be overcome. Suppose that a computational module A receives a parameter X, which is in reality the address of a parameter Y, which in turn contains the address of some data value. This can be depicted by X->Y->Z, where Z contains the actual information being passed to module A. Module A is linked to the environment process and will receive actual parameters from that process, but the content of this information will be passed to the environment process from the behavioral demonstrator. Since the address space for the environment process is isolated from that of the behavioral demonstrator, neither X nor Y would have meaning when passed to the environment process. This problem is solved by having the behavioral demonstrator generate type definitions in the environment process that will be used in the main loop section to build the appropriate "package" around information being passed to modules. These definitions are made by iteratively "unwinding" the type definitions for the formal parameters of computational modules and generating corresponding type definitions in the environment process to "rewind" actual parameters passed to these modules from the behavioral demonstrator. This general parameter packaging scheme is depicted in Figure 7.3.14.



Declaration section

typedef Name Z_1
typedef Z_1 Y_1 pointer
typedef Y_1 X_1 pointer

variable X X_1
variable Y Y_1
variable Z Z_1

Main loop section

```
Receive(Request)
if (Request.service = 'Perform A')
  then begin
    Move(Request.Z, Z)
    Y = address(Z)
    X = address(Y)
    A(X)
    Move(Z, Request.Z)
  end
Send(Request)
```

Figure 7.3.14 Parameter packaging scheme of BD

The static initialization section is composed of a variable declaration section and a variable initialization section. Variables used throughout the environment process are declared in the declaration section. Those variables which are not dynamic (ie. do not require pointer references and dynamic memory allocation) are initialized to their example values (determined from the extended type definitions) by standard assignment statements in the initialization section. Discussion of why dynamic variables are not initialized will be deferred until the next section.

Since the order of invocation of any function is not known until behavioral demonstration, the main control component for the environment process must rely on the BD to inform it of which procedures to invoke and with which data values. This is accomplished by making the main control for the environment process similar to a monitor which performs services on behalf of the BD. The BD sends a message to the environment process. The environment process decodes the message, invokes the necessary function with default values, or values supplied by the BD, and send the results back to the BD. This organization is depicted in Figure 7.3.15. The environment process can perform a number of services on behalf of the behavioral demonstrator; these are:

- Invoke functions with parameters supplied by the behavioral demonstrator.

- Supply values to the behavioral demonstrator for use in conditional expression evaluations.
- Perform conditional expression evaluations of dynamic data.
- Act as a distributed symbol table for the behavioral demonstrator.

Most of these services were invented to support dynamic variables, which are generally accessed through pointer variables that point to areas in memory dynamically allocated for data storage. These pointer variables have no meaning outside of the address space in which they were created, thus can not be used by the behavioral demonstrator. To solve this problem, the behavioral demonstrator uses the environment process as an extended symbol table that stores and manipulates dynamic variables on behalf of the behavioral demonstrator. This scheme presumes that the environmental process will not deadlock, loop indefinitely, or generate run-time errors. A contrived example illustrates how such events might occur and how the BD will respond.

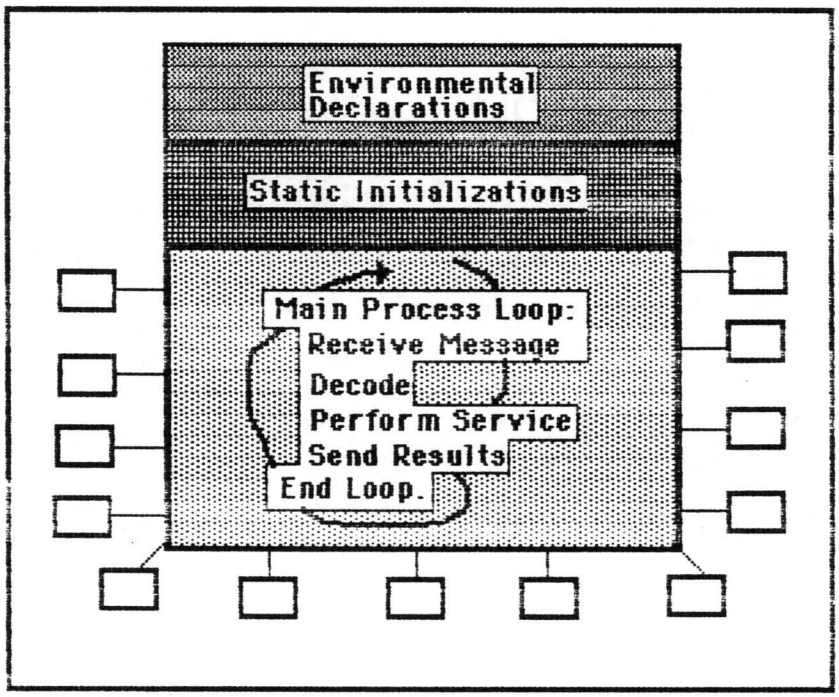


Figure 7.3.15 BD generated environment process

Suppose that there exists three modules A, B, and C, all of which access a data structure G. It is sufficient to assume that module A initializes G to some starting state. Suppose futher that module B transforms some information in G or constrains it in some way and that module C accesses this information, but assumes that it has been transformed by module B. Behavioral demonstration could occur before module B but after module C have been approved for integration. In this case, module C would probably abort, loop indefinitely, or cause the environment process to deadlock. This could happen for only two reasons: either module C simply has an implied dependency on module B, or module C requires refinement to handle rare events. If module C aborts, the BD would be informed and would determine the reason for the fault. If the implied dependencies dimension of the requirements space for module C is empty, then the BD will record the fault in the project status space and will continue behavioral demonstration by re-spawning the environment process. If the implied dependency has been recorded, the error is expected and behavioral demonstration continues. Deadlock and infinite loops can be recognized by users of the BD by recognizing when execution of a module "locks-up." For such situations, the BD user simply selects to stop behavioral demonstration of that function by depressing control C (in UNIX). This will cause the BD to kill the environment process and re-spawn it so that behavioral demonstration can

continue. This re-spawning entails re-transmission of all previous requests to the new environment process to insure that the dynamic portion of the symbol table is as it was prior to when the environment process aborted. Hence, not only does the behavioral demonstrator build, compile, and spawn the environment process, but also records all transmissions to and from the process for check-pointing purposes.

The symbol table used in the BD is different from most symbol tables in not its organization, but its content. The symbol table is organized as a list of threaded binary trees. Identifiers are first hashed to determine which binary tree to search, then a standard inorder tree search used to retrieve entries. A threaded tree is necessary for sequentially accessing all members in the table during type declarations and static initializations for the environment process. During recursive execution, each procedure has a binary tree containing all the symbol table entries for each re-entry. In general, symbol tables contain defined names, associated call names (sometimes called aliases), and attributes for identifiers. Although other information is necessary for correct execution and identifier use, it is seldom contained in symbol tables. This information is depicted in Figure 7.3.16 (adapted from [LEEJA85]). For interpreters most of this additional information is carried, but seldom is the actual value seen by humans contained in the symbol

table. Since not all variables may be declared prior to behavioral demonstration, the perceived value as seen by the human may need to be stored until a proper interpretation of the value can be determined in the context of the variable's first use.

For example, consider a variable X, produced by a module A, has not been fully defined, but it is known that X will contain a field Y that will later be tested in a condition that appears in the supervisor for module A. A conditional such as { X.Y <= 15 } would indicate that X is a record and contains at least one field, namely Y, whose type is known to be a subset of the integers. This information is used to update the symbol table entries for the variable X as it is gathered. Although it is not a specific requirement of the BD, these inconsistencies in how X is used can be pin-pointed and conveyed to appropriate personnel by adding points to the appropriate information spaces.

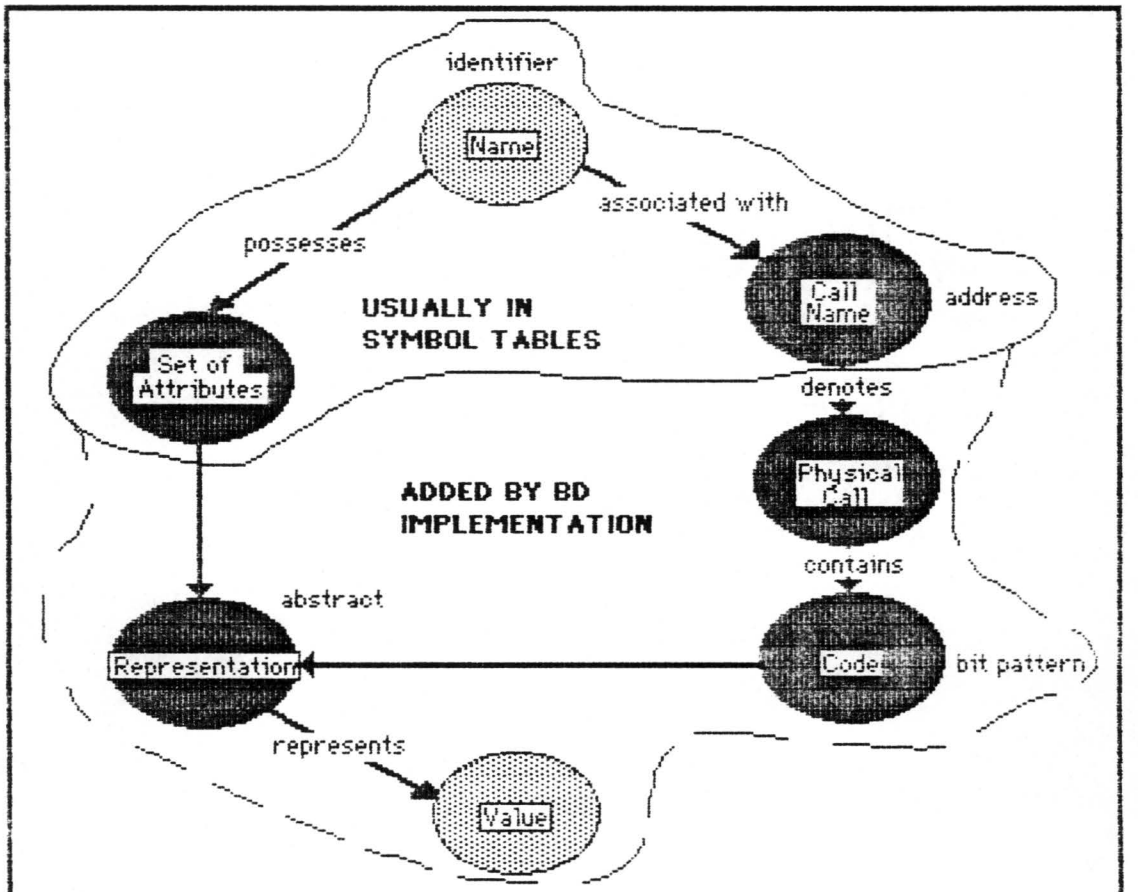


Figure 7.3.16 Extended Symbol Table Information in BD

The final design issue of interest is how points in one information space may affect points in others. Since it is recognized that design of one module might lead to the creation of a number of other modules, information spaces should be viewed as small collections of points, any one of which can be expanded into a completely new information space. This supports the notion in SUPERMAN that a supervisor tells "what" is to be done, while its SFD expansion tells "how" it is to be done. In the very early stages of development "what and how" for a supervisor is recorded in the requirements/specification information space. As the SFD for this supervisor is designed, the movement of the "how" for this supervisor's information space to that of supervised modules is supported by the information space window manager of the BD. This dissemination of information from space to space is depicted in Figure 7.3.17.

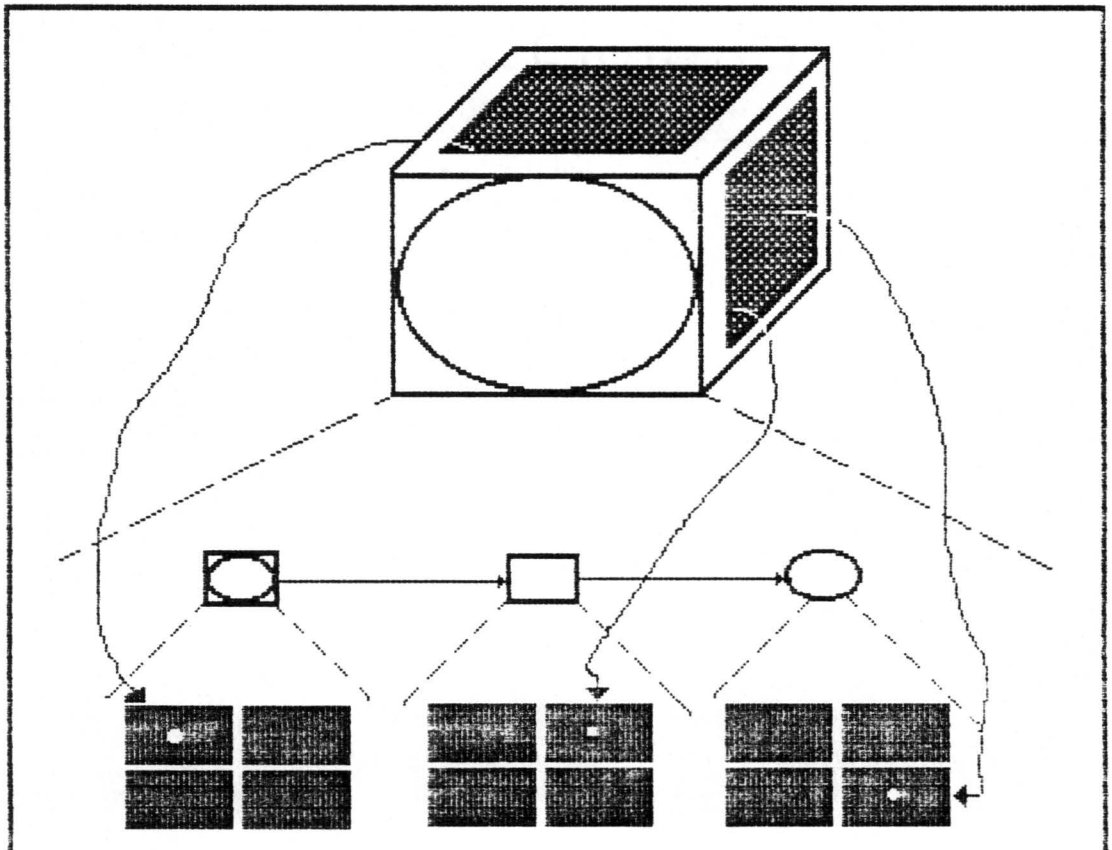


Figure 7.3.17 Space to space information dissemination

7.3.3 Design Decisions Affecting Design

This section describes the design decisions, alluded to in previous sections, that have governed the BD design. These design decisions include:

- Only simple variable assignments will be supported in ICB's.
- A color graphics bit-mapped display will be used as the target machine.
- Only the "C" programming language will be generated by the behavioral demonstrator.
- Only the normal distribution will be provided for critical time statistics.
- Only static data will be initialized.
- Concurrency will be demonstrated as sequential executions of alternate paths of control.

Assignments will be supported in ICBs because they are fundamental for certain control constructs, WHILE for instance. Other statements and constructs are not supported because they may not have general applicability for all target languages the BD may support in the future. Another reason other ICB constructs will not be supported is that parsing of ICBs is not an important contribution of this thesis and has only been included for completeness. The parser for the ICBs and conditionals that label control lines will be implemented using LEX and YACC (tools in the UNIX

system, see [LESKM83] and [JOHNS83]), so changes in this constraint will affect the BD design only slightly.

A color bit-mapped display (a Silicon Graphics IRIS 2400) has been chosen as the primary development workstation for the entire DMS project, thus must be used for the BD implementation. This mandate has carefully guided the BD architecture and interface design. A primary reason for choosing a bit-mapped display was to provide a workstation capable of supporting many user interfaces that could be designed using AIDE.

Similarly, the "C" language ([RITCH83]) has been selected as the implementation language for all DMS tools, thus must be used to implement the BD. This language has been chosen as the target language for the BD and the Computational Development Facility so that DMS can be boot-strapped as a first large-scale development effort.

One contribution of this thesis has been the definition of the critical time statistics. Only after a large experimental development effort using the BD and these statistics can a true distribution for each statistic be determined. For this reason, only the normal distribution is being supported.

The reason that the BD will initialize only static variables, as opposed to dynamic variables which utilize pointers and dynamically allocated memory is that in general, static initialization of dynamic variables cannot be done. This is true because the contents of dynamic variables depend not only on the syntactic and semantic form of the variable declarations, but also on the pragmatics of the variables. These pragmatics are imposed by the functions which operate on the dynamic data structures. If axiomatic specifications were used to describe the intended behavior of functions these pragmatics would be explicit, but without a formal specification technique, this problem can be shown to be equivalent to the Halting Problem, thus is unsolvable.

In SUPERMAN, concurrency is represented by a series of concurrent paths that emerge from a circular concurrency construct and later converge to one another. This construct appears in Appendix A.

Scanning the entire supervisory structure to determine which computational components were to be included in which concurrent processes requires the following algorithm:

1. For each set of concurrent paths, define the top-most path as the "default path," taken by the supervisor.

2. Begin with the system supervisor and determine the default path taken by this function. All functions appearing along this path and all functions recursively appearing on the default path of supervisors in the SFD for the system supervisor should be added to the first process generated by the BD.

3. Iteratively examine each of the remaining concurrent paths in the system supervisor as done in step 2. Place each concurrent path in a separate process. Nested concurrent paths are treated like separate functions (functions appearing on the default path should be added to the current process being generated).

4. Recursively expand supervisory functions from the system supervisor using the remaining concurrent paths not traversed in each function. Mark each function as having been examined so as to avoid loops and knots in the algorithm.

This algorithm is similar to the one that must be used by the GPL coder to automatically generate the target system. Both algorithms are based on previous work done on flowgraph traversal and automatic program restructuring (see [BAKER77]). Since true concurrency would only serve to confuse end-users who could be using the BD, behavioral demonstration gains nothing by generating all concurrent processes that would run in the final system. To this end, the BD places all approved modules in a single environment process,

and uses the context switching capabilities provided by the context switching stack to demonstrate concurrency.

8.0 CONCLUSIONS

Behavioral Demonstration and the Behavioral Demonstrator have made made it possible to demonstrate system behavior at **any** time during system development and has solved some of the major problems encountered in previous prototyping efforts. It has been shown that without a formal, axiomatic specification technique, dynamic data structures cannot be instantiated with static data without some pragmatic assumptions being made. If a satisfactory specification technique could be found, it would suffer from not being understandable by end-users, hence would lose its usefulness for demonstrating system behavior to users. This rapid prototyping technique supports a graceful evolution from system requirements to system implementation through the following automation-based life-cycle paradigm:

1. Operational specification is made at a very high level using Operational Procedure Diagrams.
2. Through behavioral demonstration and iterative refinement this executable specification evolves into design.
3. Behavioral demonstration is used to validate both the specification and design matches developers' intensions and customers' needs.
4. Bottom-up and top-down design is used to refine all functions until they have been approved for integration.
5. The GPL coder is used to automate production of the final system.

6. The graphical design and information space is used to automatically document the final product.

7. Maintenance occurs through refinement and iterative behavioral demonstration of the graphical design.

Using behavioral demonstration, system prototypes are not simulations, but are true systems, slowly evolving into final products. System develeloped using this life-cycle paradigm will have an abundance of information captured and maintained in the information spaces defined in this thesis. This information may be used to pinpoint potential trouble areas in designs, to determine true programmer productivity measures, to estimate software cost and software quality, and most importantly, to predict the impact and cost of user-suggested system enhancements. Exactly how these information spaces will be used has yet to be seen.

This work has taken place as part of an on-going research program at Virginia Tech. Implementation details for the final production version of the behavioral demonstrator are expected to be published in early 1986 as part of a series of technical reports. The next section suggests how the research in this thesis may be evaluated, and describes future work inspired by this work.

9.0 FUTURE WORK

The most important future activity that this work inspires is the implementation of the complete Computational Development Facility portion of the Dialogue Management System. The work also inspires experimentation in software metrics using the information collected and maintained by the behavioral demonstrator. It should inspire research by human factors engineers to determine the usefulness of behavioral demonstration in the design of complex systems. This work should be evaluated in an industry setting against alternative software design techniques using different methodologies. Research should continue to determine what the "true" information spaces for software systems really are, and what are the "true" dimensions which form the bases for these spaces. Research should seek to develop a "universal language" for Intercode Blocks in Supervised Flow Diagrams that embodies reusable code which is understandable by all persons exposed to system designs. One promising approach is to use a macroprocessor based universal language that would support "frame-based programming" (see [LANER81], [EMBLE78], [BARCU81], [CARDE82], and [GROUS84]) and multilingual translation.

BIBLIOGRAPHY

[AHOAV78] Aho, A. V. and Ullman, J. D., Principles of Compiler Design, Addison-Wesley Publishing, Reading, Mass., March 1978.

[BAKER77] Baker, B. S., "An Algorithm for Structuring Flowgraphs," Journal of the ACM, Vol. 24, #1, ACM Jan. 1977, pp. 98-120.

[BALZE82] Balzer, R. M., Goldman, N. M., and Wile, D.S., "Operational Specification as the Basis for Rapid Prototyping," ACM SIGSOFT Software Engineering Notes #7, ACM December 1982, pp. 3-16.

[BALZE83] Balzer, R., Cheatham, T. E., and Green, C., "Software Technology in the 1990's Using a New Paradigm," IEEE Computer, IEEE November 1983, pp. 39-45.

[BARCU84] Barcucci, E. and Pelacani, G., "A Software Development System Based on a Macroprocessor," Software-Practice and Experience, Vol. 14, #6, June 1984, pp. 519-531.

[BOARH85] Boar, B. H., Application Prototyping: A Project Management Perspective, American Management Association, New York, 1985.

[BOEHM84] Boehm, B. W., Gray, T. E., and Sewaldt, T., "Prototyping vs. Specifying: A Multi-Project Experiment," Proceedings of the 7th International Conference on Software Engineering, IEEE 1984, pp. 473-484.

[BOEHM75] Boehm, B. W., McClean, R. K., and Urfrig, D. B., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," IEEE Transactions on Software Engineering, IEEE March 1975, pp. 125-131.

[BOEHM78] Beohm, B., "Software Engineering - R and D Trends and Defense Needs," in Research Directions in Software Technology, MIT Press 1978.

[BROOK75] Brooks, F. P. Jr., "The Mythical Man-Month: Essays on Software Engineering," Addison-Wesley Publishing, Reading, MA. 1975.

[CALLA83] Callan, J. E. III, "The Behavioral Demonstrator: A Requirements Specification Executor," Departments of Computer Science and Industrial Engineering and Operations Research Technical Report CSIE-83-14, VPI&SU May 1983.

[CARDE82] Cardenas, A. F. and Grafton, W. P., "Challenges and Requirements for New Application Generators," AFIPS Conference Proceedings, National Computer Conference 1982, pp. 341-349.

[COXBJ83] Cox, Brad J., "The Message/Object Programming Model," SOFTFAIR Proceedings, IEEE July 1983, pp. 51-60.

[DIJKS72] Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, Dahl, O. J., Dijkstra, and Hoare, C. A., New York: Academic Press, 1972.

[DODDW80] Dodd, W. P., "Prototype Programs," IEEE Computer, IEEE Feb. 1980,

[EHRIC81] Ehrich, R. W. and Hartson, H. R., "DMS -- An Environment for Dialogue Management," COMPCON81 Proceedings, Washington, DC, Sept. 1981.

[EMBLE78] Embley, D. W., "Forms-Based Automatic Program Generation," Journal of the ACM, ACM Dec. 1978, pp. 972.

[GOMAA81] Gomaa, Hassan and Scott, Douglas B. H., "Prototyping as a Tool in the Specification of User Requirements," Proceedings of the 5th International Conference on Software Engineering, IEEE March 1981, pp. 333-342.

[GROUS84] Grouse, P. J., "FlowGuide - A Programmer's Work Station," AFIPS Conference Proceedings, National Conputer Conference 1984, pp. 329-341.

[GUTZS81] Gutz, S., Wasserman, A. I., and Spier, M. J., "Personal Development Systems for the Professional Programmer," IEEE Computer, IEEE April 1981, pp. 45-53.

[HANA80a] Hanau, P. R., and Lenorowitz, D. R., "A Prototyping and Simulation Approach to Interactive Computer

System Design," Proceedings of the ACM Conference on Design Automation 1980.

[HANA80b] Hanau, P. R., and Lenorowitz, D. R., "Prototyping and Simulation Tools for User/Computer Dialogue Design," Proceedings of ACM SIGGRAPH 1980.

[HARTH85] Hartson, H. R., (ed.), Advances in Human-Computer Interaction, Ablex Publishing 1985.

[HART84a] Hartson, H. R., Johnson, D. H., and Ehrich, R. W., "A Human-Computer Dialogue Management System," Proceedings of the First IFIP Conference on Human-Computer Interaction (Interact '84), London, Sept. 1984.

[HART84b] Hartson, H. R., and Johnson, D. H., "Dialogue Management: New Concepts in Human-Computer Interface Development," paper submitted to ACM Computing Surveys March 1984.

[HENRY79] Henry, S. M., "Information Flow Metrics for the Evaluation of Operating Systems' Structure," Ph.D. Thesis, Computer Science Department, Iowa State University, Ames, Iowa, 1979.

[HENRY85] Henry, S. M., personal communication to the author.

[HENRY81] Henry, S. M. and Kafura, D., "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, IEEE Sept. 1981, pp. 510-518.

[HOLTA79] Holt, A., "Net Models of Organizational Systems - In Theory and Practice," Ansatze zur Organisationstheorie, Rechnergestuetzter Informationssysteme, Report #111, Petri, C. A., (ed.), P. Oldenbourg, Muenchen Germany, 1979.

[JOHNS85] Johnson, D. H., "The Structure and Development of Human-Computer Interfaces," Ph.D. Thesis, Computer Science Department, Virginia Tech, Blacksburg, Virginia, 1985.

[JOHNS83] Johnson, S. C., "YACC: Yet Another Compiler-Compiler," in UNIX Programmer's Manual Vol. 2, Bell Telephone Laboratories, Murray Hill, New Jersey, 1983, pp. 353-387.

[LANER81] Lanergan, R. G. and Dugan, D. K., "Software Engineering with Reusable Designs and Code," IEEE

Transactions on Software Engineering, 1981, pp. 296-303.

[LEEJA85] Lee, J. A. N., Lecture presented to CS5612 students during Spring 1985.

[LENOR77] Lenorowitz, D. R. and Ramsey, H. R., "A Dialogue Simulation Tool for Use in the Design of Interactive Computer Systems," Proceedings of Human Factors Society, Santa Monica: CA. 1977, pp. 95-99.

[LESKM83] Lesk, M. E. and Schmidt, E., "LEX - A Lexical Analyzer Generator," in UNIX Programmer's Manual Vol. 2, Bell Telephone Laboratories, Murray Hill, New Jersey, 1983, pp. 388-400.

[LINGE80] Linger, R. C., "The Management of Software Engineering Part III: Software Design Practices," IBM Systems Journal Vol. 19 #4, IBM April 1980, pp. 432-450.

[MASON81] Mason, R. E. A. and Carey, T. T., "Productivity Experiences with a Scenario Tool," COMPCON Proceedings, IEEE Sept. 1981, pp. 106-120.

[MCCRA82] McCracken, D. D. and Jackson, M. A., "Life Cycle Concept Considered Harmful," ACM SIGSOFT Software Engineering Notes, ACM April 1982, pp. 29-32.

[MOHAN81] Mohanty, Siba N., "Software Cost Estimation: Present and Future," Software-Practice and Experience Vol 11, 1981, pp. 103-121.

[REPOR81] "Report of the Tools Working Group of the Engineering Productivity Workshop," COMPCON Proceedings, IEEE Sept. 1981, pp. 353-359.

[RITCH83] Ritchie, D. M., "The C Programming Language - Reference Manual," in UNIX Programmer's Manual Vol. 2, Bell Telephone Laboratories, Murray Hill, New Jersey, 1983, pp. 247-277.

[ROMAN84] Roman, G. C.,"A Taxonomy of Current Issues in Requirements Engineering," Department of Computer Science Technical Report WUCS-84-8, Washington University Sept. 1984.

[ROSSD76] Ross, D., "Homilies for Humble Standards," Communications of the ACM Vol 19 #11, ACM November 1976, pp. 595-564.

[ROSSD77] Ross, D. and Schoman, W., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, IEEE Jan., 1977.

[SWART82] Swartout, William and Balzer, Robert., "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM Vol 25 #7, ACM July 1982, pp. 438-440.

[WASSE80] Wasserman, A. I., "Information System Design Methodology," Journal for the American Society for Information Science, 1980.

[WASSE77] Wasserman, A. I.,"On the Meaning of Discipline in Software Design and Development," Tutorial on Software Design Techniques, IEEE 1977, pp. 37-44.

[WASSE85] Wasserman, Anthony I. and Shewmake, David T., "The Role of Prototypes in the User Software Engineering (USE) Methodology," to appear in Hartson, H. R., (ed.), Advances in Human-Computer Interaction, Ablex Publishing 1985.

[WINST75] Winston, R. W., "Software Requirements Analysis," in Practical Strategies for Developing Large Software Systems, Horowitz, E. (ed.), Addison Wesley, 1975.

[WOODS70] Woods, W. A., "Transition Network Grammars for Natural Language Analysis," Communications of the ACM Vol. 13 #10, ACM Oct. 1970, pp. 591-606.

[YUNTE85] Yunten, T., "A SUPERvisory Methodology And Notation for Human-Computer System Development," to appear in Hartson, H. R., (ed.), Advances in Human-Computer Interaction, Ablex Publishing 1985.

[ZAVEP81] Zave, Pamela and Yeh, Raymond T., "Executable Requirements for Embedded Systems," Proceedings of the 5th International Conference on Software Engineering, IEEE 1981, pp. 295-304.

[ZAVEP80] Zave, P.,"The Operational Approach to Requirements Specification for Embedded Systems," University of Maryland Computer Science Technical Report TR-976, Dec. 1980.

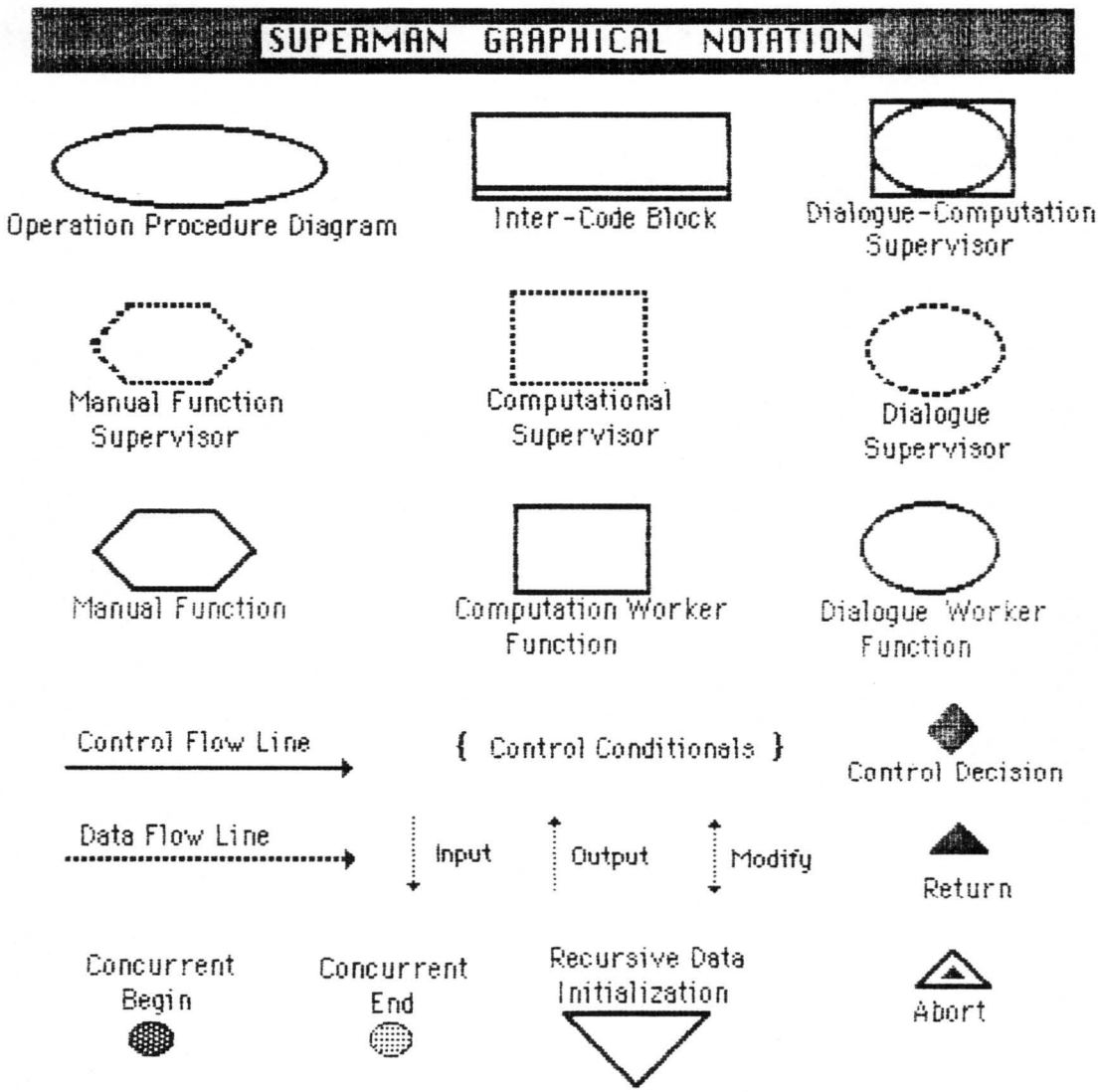
[ZAVEP84] Zave, Pamela, "The Operational Versus the Conventional Approach to Software Development," Communi-

cations of the ACM Vol 27 #2, ACM Feb. 1984, pp. 104-118.

[ZLOOF75] Zloof, M. M., "Query By Example: The Invocation and Definition of Tables and Forms," Proceedings of the First International Conference on Very Large Data Bases, Sept. 1975.

APPENDIX A

This appendix describes the syntax and symantics for SUPERMAN pictorially through the following diagrams.


SUPERMAN GRAPHICAL NOTATION
Operation Procedure Diagram
Inter-Code Block
Dialogue-Computation Supervisor
Manual Function Supervisor
Computational Supervisor
Dialogue Supervisor
Manual Function
Computation Worker Function
Dialogue Worker Function
Control Flow Line
{ Control Conditionals }
Control Decision
Data Flow Line
Input
Output
Modify
Return
Concurrent Begin
Concurrent End
Recursive Data Initialization
Abort

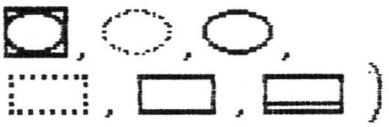
SYNTAX FOR SUPERMAN NOTATION

SYSTEM → CONCURRENT STRUCTURES | STRUCTURE

STRUCTURE →  | 

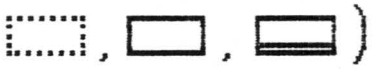
 → OPD OF ANY COMBINATION OF ()

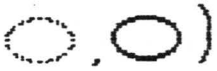
EXCEPT ANY COMBINATION OF ONLY ()

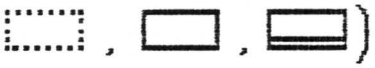
 → SFD OF ANY COMBINATION OF ()

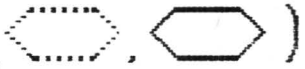
EXCEPT ANY COMBINATION OF ONLY ()

OR

ANY COMBINATION OF ONLY ()

 → SFD OF ANY COMBINATION OF ()

 → SFD OF ANY COMBINATION OF ()

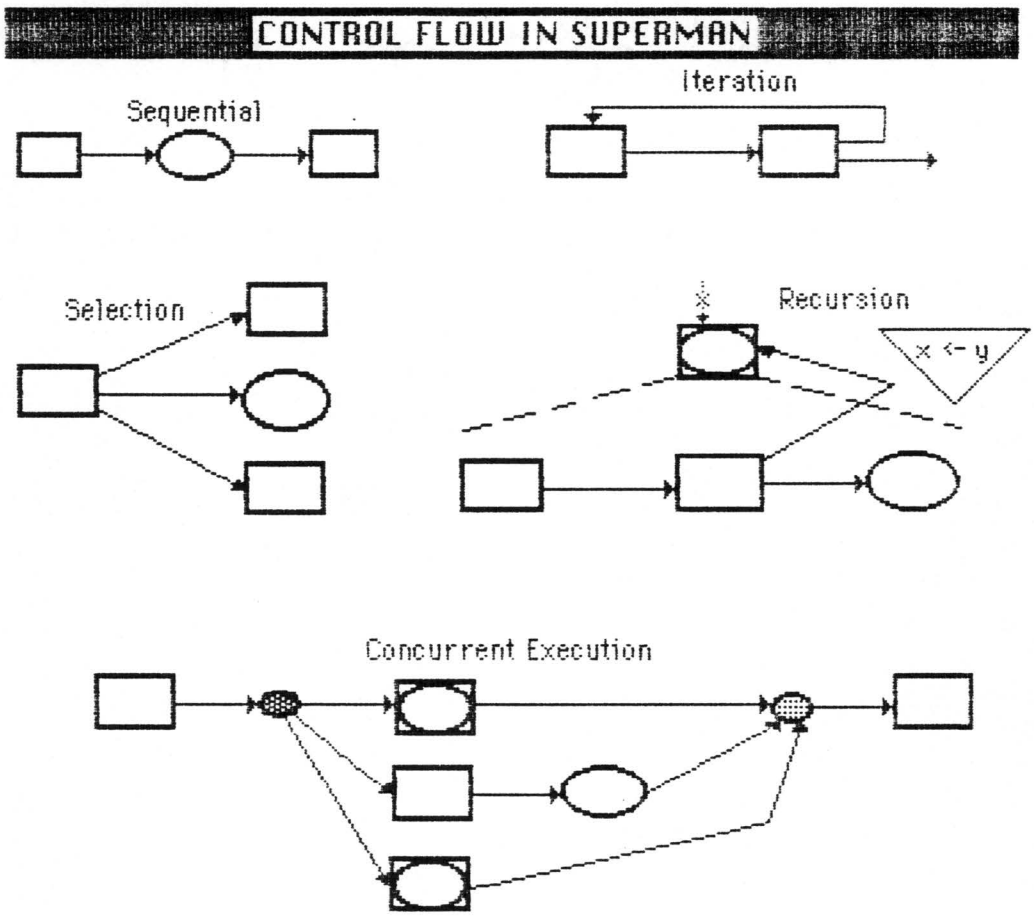
 → OPD OF ANY COMBINATION OF ()

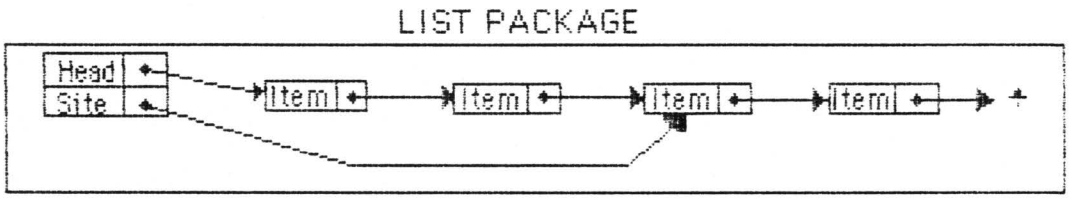
 → SFD OF ()

 → CODE FOR DIALOGUE FUNCTON

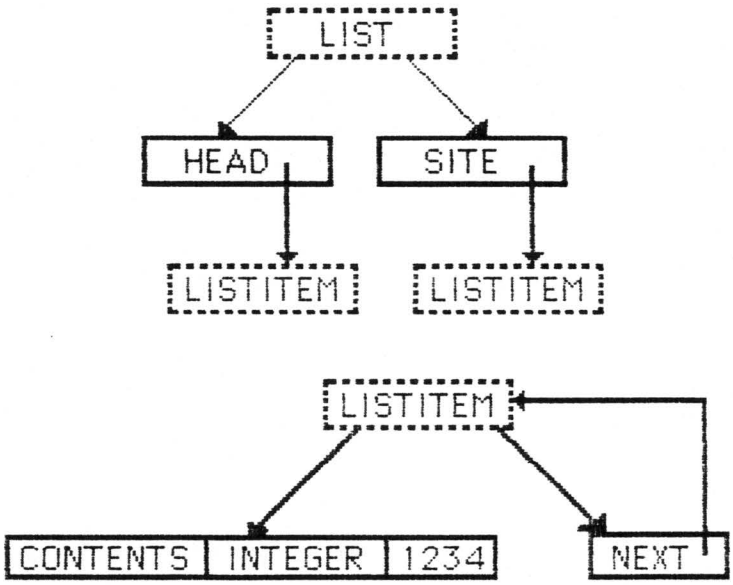
 → CODE FOR INTERNAL_CODE_BLOCK

 → PROCEDURE FOR MANUAL FUNCTION


CONTROL FLOW IN SUPERMAN
Sequential
Iteration
Selection
Recursion
x <- y
Concurrent Execution



GRAPHICAL DATA DEFINITIONS IN SUPERMAN



APPENDIX B

INFORMATION SPACE DESCRIPTION

This appendix describes each dimension defined in this thesis for all six information spaces.

Requirements/Specifications

Interface Features contains the desired features that are to be included in the human-interface (for dialogue components), machine-interface (for embedded systems), or intermodule interface (for computational components).

Performance Constraints contain imposed constraints which indicate the performance (real-time) that is expected of modules.

Required Reliability indicates the minimal reliability that should accepted from a module before it may be approved. This would be of particular use in distributed software development.

Intended Purpose conveys in a clear manner the user requirements that this module is to support.

Implementation Priority is used to determine critical paths in large software projects and to place priorities on modules that may be used to guide the development effort.

Predicted Completion Time is entered once into the module's information space when detailed design begins. Once entered, this information is frozen. This information can be used later to determine expected time overruns and may be used to validate software complexity metrics.

Assumptions record any assumptions that people responsible for the module's design are making. This can later be used to formalize testing procedures for the module.

Implied Dependencies record any intermodule and intramodule dependencies that could affect the performance and reliability for the module.

Project Status

Time Remaining is used to record the expected time before module completion as anticipated by the design team. Associated with each update in this dimension is a comment indicating why the entry differs from the previous expected time entry.

Personnel is used to manage which people are assigned to particular modules. This information is used by the behavioral demonstrator for distributing mail to appropriate personnel when design decisions are made which may affect this module.

Approval is used to record when a module is approved and by whom. Also maintained by this space is a list of personnel who should receive notices when project status changes in any way.

Problems is a space to record problems which are currently affecting the development of modules.

Status maintains a history of a module's development.

Design Decisions

Time Constraint is a space in which managers can record information related to a module's progress. This information supports the control of large projects which have imposed time constraints and contain modules that are critical to a project's success.

Algorithmic is a space for recording design decisions which affect the particular algorithm chosen to solve problems addressed in the module.

Hardware is a space for recording decisions based on hardware constraints.

Priority is a management space used for controlling critical paths in a project's development. Managers use this information to control manpower and to funnel it to critical places in projects.

Efficiency is a space to record design tradeoffs based on efficiency issues. These tradeoffs may later be used to locate areas where tuning would be useful.

Critical Time Statistics

Exposure reflects the time that developers have been exposed to a particular function in the SFD expansions of all other functions. Since function C may be referenced by functions A and B, any access by designers to functions A and B will add time measurements to the exposure of function C.

BD as a time statistic reflects the time that developers have used the BD to observe the behavior of a particular function.

Access reflects the time that developers have accessed the SFD for a particular function.

Transformation is the time spent by developers transforming the system into target versions for end-users. This statistic presumes the GPL coder tool is used to make the transformation. Also, it assumes that the first transformation marks the end of development and the beginning of maintenance and refinement for a system.

Refinement is the time spent modifying (editing) a function after the system has been transformed for the first time.

Design reflects the time developers have spent designing a particular function. Design begins when the first SFD symbol appears in the SFD expansion for a supervisor, when the first line of code is written for a computational worker, when the first interaction is developed for a dialogue worker, or when the first bit in any information space owned by a function appears.

Execution reflects the time that a particular function takes to execute. In particular, this refers to computational functions, as similar metrics are collected in AIDE for dialogue functions.

Notes and Comments

Enhancement is used to convey to designers suggested enhancements.

Question is used to pose to designers a question that may affect or impact other portions of a project.

Correction is used to record where corrections should be made in a design. Corrections have a higher priority for implementation than enhancements.

Messages are used to convey comments to designers, and may be used by management to issue positive feedback.

Development Reminders

This space is completely free format, and may be used any way that designers see fit.

The vita has been removed from the scanned document