THE DEVELOPMENT OF A GRAPHICAL NOTATION FOR THE FORMAL SPECIFICATION OF SOFTWARE

THESIS

Gene A. Place
Captain, USAF

AFIT/GCS/ENG/90D-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/90D-10

THE DEVELOPMENT OF A GRAPHICAL NOTATION FOR THE FORMAL SPECIFICATION OF SOFTWARE

THESIS

Gene A. Place
Captain, USAF

AFIT/GCS/ENG/90D-10

AFIT/GCS/ENG/90D-10

THE DEVELOPMENT OF A GRAPHICAL NOTATION FOR THE FORMAL SPECIFICATION OF SOFTWARE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

Gene A. Place, B.S.
Captain, USAF

December, 1990

Approved for public release; distribution unlimited

Acknowledgments

Many individuals have contributed to the final results of this research. First, many thanks go to my thesis advisor, Dr (Maj) Paul Bailor, for providing his boundless enthusiasm when it was needed, and for restricting his enthusiasm when it wasn't. Thanks also go to my committee members, Dr (Maj) Patricia Lawlis and Dr (Maj) David Umphress who took care of the "odd man out". Most importantly, sincerely thanks go to my wife, Theresa, whose patience and support for this project were extraordinary.

Gene A. Place

Table of Contents

Page

List of Figures

Figure Page

Figure Page

AFIT/GCS/ENG/90D-10

Abstract

The program transformation lifecycle model, proposed by Balzer in 1984, may hold the key to the dramatic leap in software engineer productivity necessitated by the tremendous growth in the world's demand for software. This leap in productivity is made possible by the program transformation lifecycle's reliance on formal specifications rather than programs as the primary vehicle for the creation and maintenance of software systems. Because formal specifications are concerned only with system behaviors and not implementation details, formal specifications tend to present critical system characteristics much more concisely than programs. The major disadvantage of using formal specifications in a software development process, however, is that, because of their highly mathematical nature, formal specifications tend to be very difficult to create, understand, and maintain for the average software engineer or programmer.

This ~~research~~ thesis develops a graphical formal specification language based on the Refine wide spectrum language using a graph based iconic representation to present formal specifications in a format that is much easier to create and manipulate than the equivalent textual formal specifications. The development of this graphical formal specification language proceeds in two steps: the Refine language is first decomposed into its primitive data types and operations, then iconic representations are developed for each of the primitive data types and operations that have been identified. In addition to the development of a graphical formal specification language, this research also proposes a number of heuristics to convert existing graphical program design notations, such as data flow diagrams, state

(KR)

transition diagrams, and entity relationship diagrams, into graphical formal specification syntax.

The architecture for a graphical formal specification environment is proposed in this research that would support the creation and revision of formal specifications using only their graphical representations. Such a graphical formal specification system is implementable using current graphical workstation environments, such as Sunview or X Windows. In addition, although this graphical language is based on the Refine language, the decomposition methods proposed in this research should facilitate the creation of graphical formal specification languages based on any well defined textual formal specification language.

THE DEVELOPMENT OF A GRAPHICAL NOTATION FOR THE FORMAL SPECIFICATION OF SOFTWARE

I. Introduction

The waterfall lifecycle model, used to help manage the majority of Department of Defense software development programs (22) and many commercial software development efforts, is rapidly losing its usefulness as a software development tool. One reason is that the waterfall model is unable to cope with the growing gap between the world's demand for software and the number of people equipped to develop that software (Fig. 1.1)(17). Unless a new software lifecycle model is developed to make programmers more productive, the software community will not be able to develop the software needed to drive the world's new industrial, information, and military computer systems.

Another growing deficiency of the waterfall model is the length of time required to develop software systems under the model. A commercial software package typically takes 2-3 years to develop; real-time command and control systems take even longer, 5-7 years or more. In today's rapidly changing business environment, a software developer bringing a package to market may find an environment totally different from the one for which the software was originally intended, rendering the software obsolete before it is even used. This time lag in fielding software systems is even more critical in a military environment where it is crucial to keep pace with a rapidly changing threat.

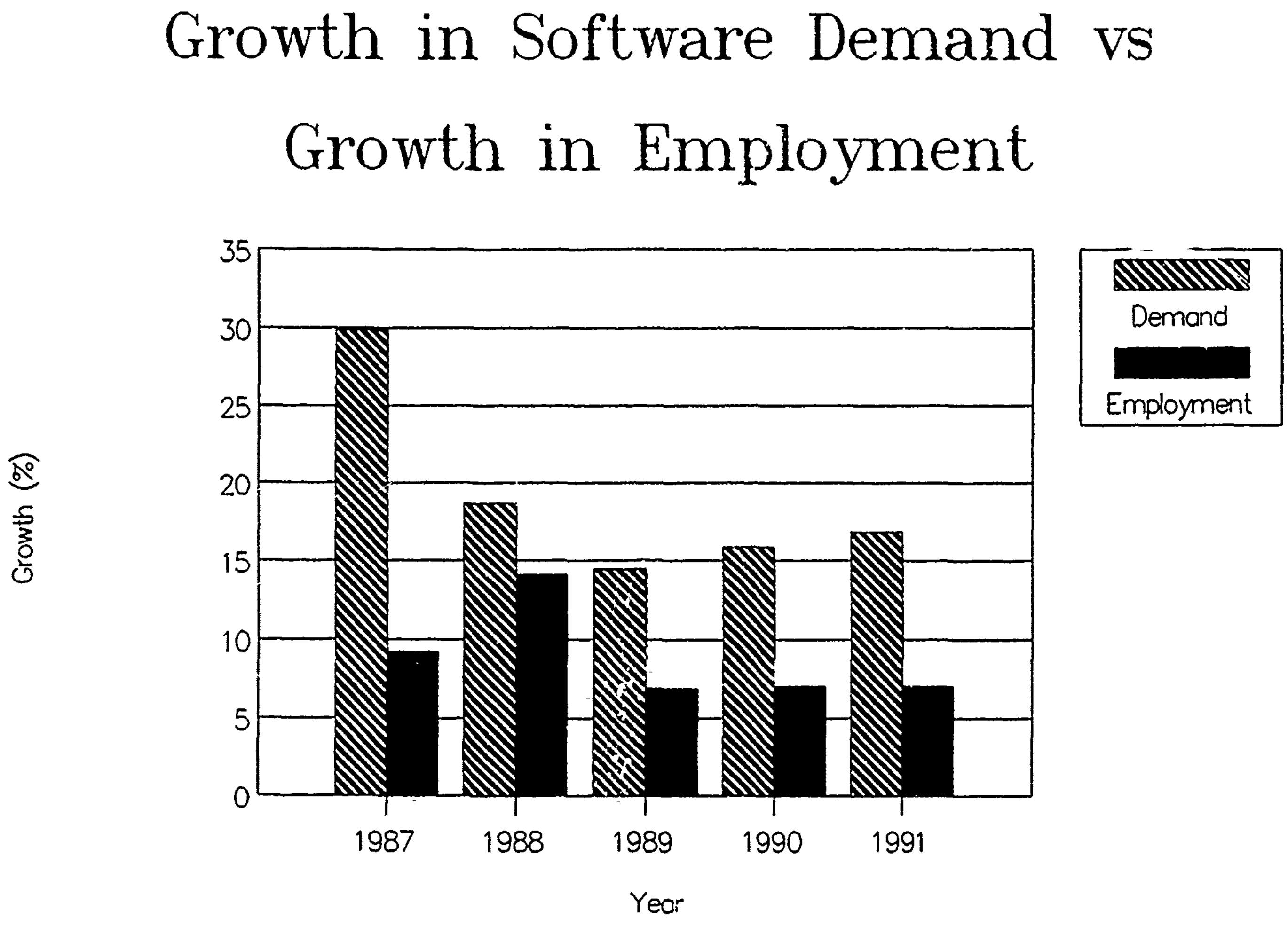


Figure 1.1. Growth in Demand for Software vs Growth in Employment

As if these reasons were not enough to justify the need for a new lifecycle model, Barry Boehm makes another argument in terms of real money. "By 1995," Boehm states, " a 20% increase in programmer productivity will result in approximately $90 billion in savings for the software industry" (8:43).

Boehm points out one way to increase programmer productivity: reduce the number of steps required to develop a program (8:45). The waterfall lifecycle model (Fig. 1.2) requires six separate steps (eight if an optional prototyping phase is performed) to develop a computer program (2:40). In 1985, Robert Balzer proposed the program transformation lifecycle model (Fig. 1.3) as an alternative to the waterfall model (2). Balzer's transformation model uses only six steps to produce a final software product, two fewer steps than the lifecycle model, which by Boehms's argument should both increase programmer productivity and reduce the development time. The transformation model achieves this two step reduction by basing software development on the concept of formal specification of software.

Formal specification is a highly precise, mathematical description of a software system's behavior. Besides reducing the number of steps required to develop software, formal specifications have several other advantages over traditional development approaches. First, formal specifications allow the developer to mathematically verify that a program does exactly what it is supposed to do, nothing more, nothing less. In traditional development methods, the only method of validating the correctness of a program is through exhaustive testing. Exhaustive testing, however, is all but impossible for any substantial program; for example, a small program with only 50 IF/THEN statements that is run on a machine capable of executing 1 million instructions per second (ips) would take just over

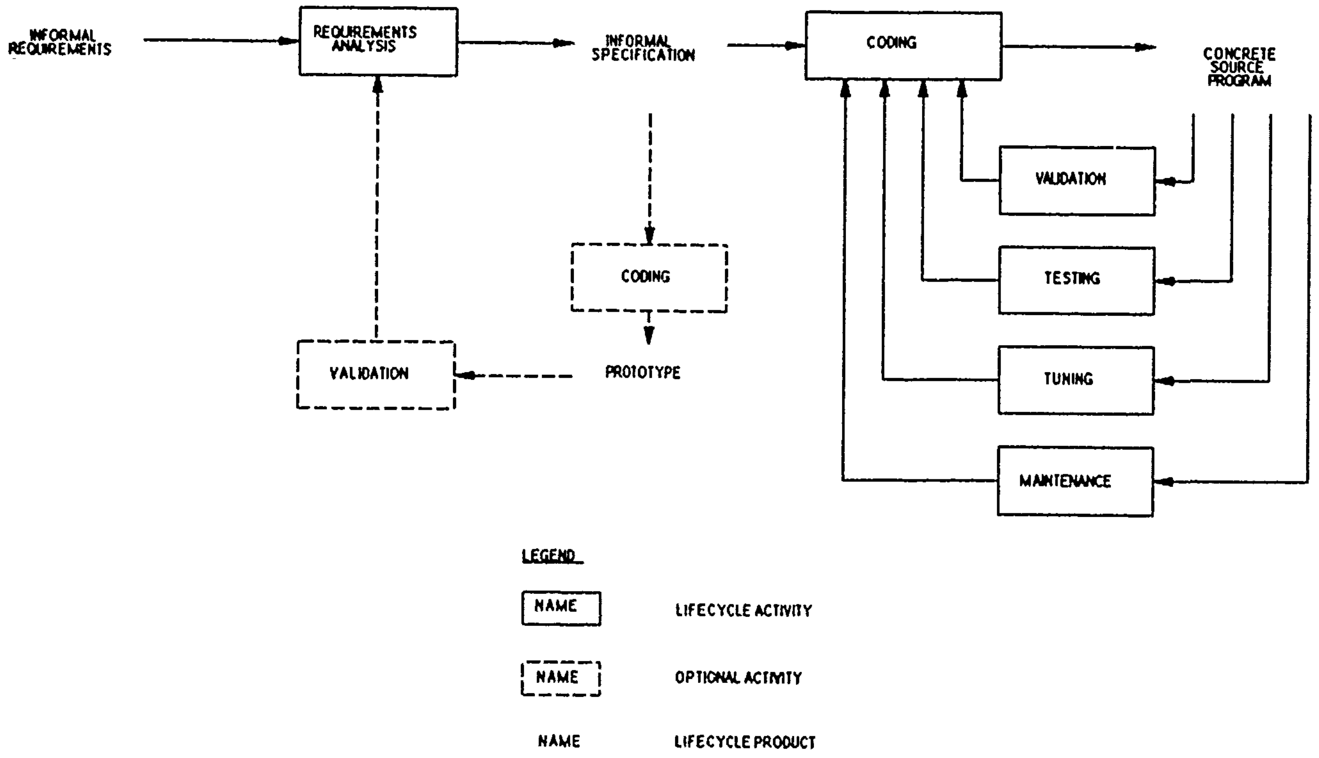
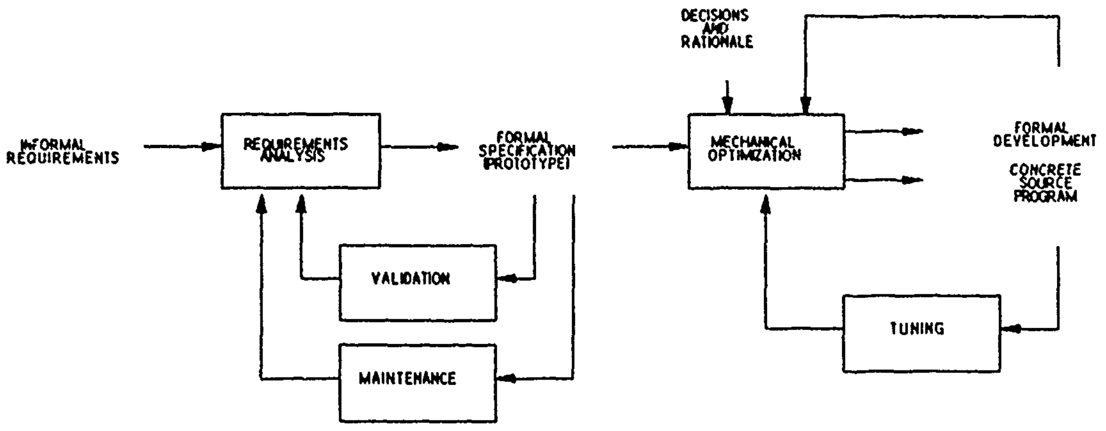


Figure 1.2. Waterfall Software Lifecycle Model



ADVANTAGES:
MATHEMATICALLY CORRECT PROGRAMS
NO VERIFICATION NECESSARY
REDUCED TESTING REQUIREMEN
MAINTENANCE PERFORMED ON SPECIFICATIONS, NOT CODE

DISADVANTAGES:
FORMAL SPECIFICATIONS DIFFICULT TO UNDERSTAND, CONSTRUCT
EXTENSIVE TRAINING NECESSARY TO USE FORMAL SPECIFICATION TOOLS

LEGEND

$\boxed{\text{NAME}}$	LIFECYCLE ACTIVITY
NAME	LIFECYCLE PRODUCT

Figure 1.3. Program Transformation Software Lifecycle Model

35 years to exhaustively test (2^{56} paths / 1,000,000 ips).

Another advantage of formal specification is its ability to identify missing or ambiguous requirements early in the development cycle. Barry Boehm points out that the cost of correcting errors late in the development cycle of a program can cost 100-150% more than correcting those same errors early in the program development effort (7). This cost is reflected both in the money required to bring a product to market and in the time required to develop the product.

Formal specifications also allow software developers to concentrate on the functionality of the program instead of the implementation details. This focus on functionality forces the developer to fully specify subroutine interrelationships and program structure. This rigorous specification in turn reduces the required program implementation and maintenance efforts.

Formal specifications, however, are not the perfect solution to the software industry's problems. Because of their highly mathematical nature, formal specifications tend to be very difficult to understand (9). Since users typically cannot understand formal specifications, a danger exists of miscommunication between users and developers which could result in the developers building a system that does not fulfill user requirements. This lack of understandability also means that an individual requires a significant amount of training before becoming proficient in specification methods.

There is also considerable discussion in academic circles about whether it is even possible to specify all types of software. Specification experts have expressed doubts about the formal specification of real-time and human interface systems: real-time systems because of their rigid time constraints and human interfaces because of the highly random nature of events in those systems.

Finally, formal specification methods have only begun to be used on complex, "real world" software projects. Until more information about the results of using formal specification methods for large scale software systems development becomes available, software managers will be hesitant to use these methods for commercial software development.

1.1 Problem Statement

Although the program transformation lifecycle holds great promise as a tool for increasing programmer productivity and for decreasing program development time, the document that serves as the basis of the transformation lifecycle, the formal specification, suffers from a severe understandability problem. As Bustart, *et al*, state:

> In order to communicate information clearly from one person to another each concept involved must be presented several times over. Usually, a point is first stated, then restated in a different way and then illustrated appropriately....When attempting to specify software, however, it is desirable to have a much more precise way of stating what is required. In particular, if a specification is given mathematically there is usually no redundancy in the information it contains. As a consequence, formal specifications can be difficult to understand. (9)

This research aims to create a graphical formal specification language that will improve the readability and understandability of formal specifications while preserving the numerous advantages of the program transformation lifecycle. This research also develops a set of heuristics to convert existing requirements analysis notations into graphical specification syntax which facilitates the use of existing program design documentation. A graphic representation of this research area is contained in Figure 1.4.

1.2 Summary of Current Knowledge

A survey of recent literature concerning formal specifications and graphical languages is contained in Chapter 2 of this thesis.

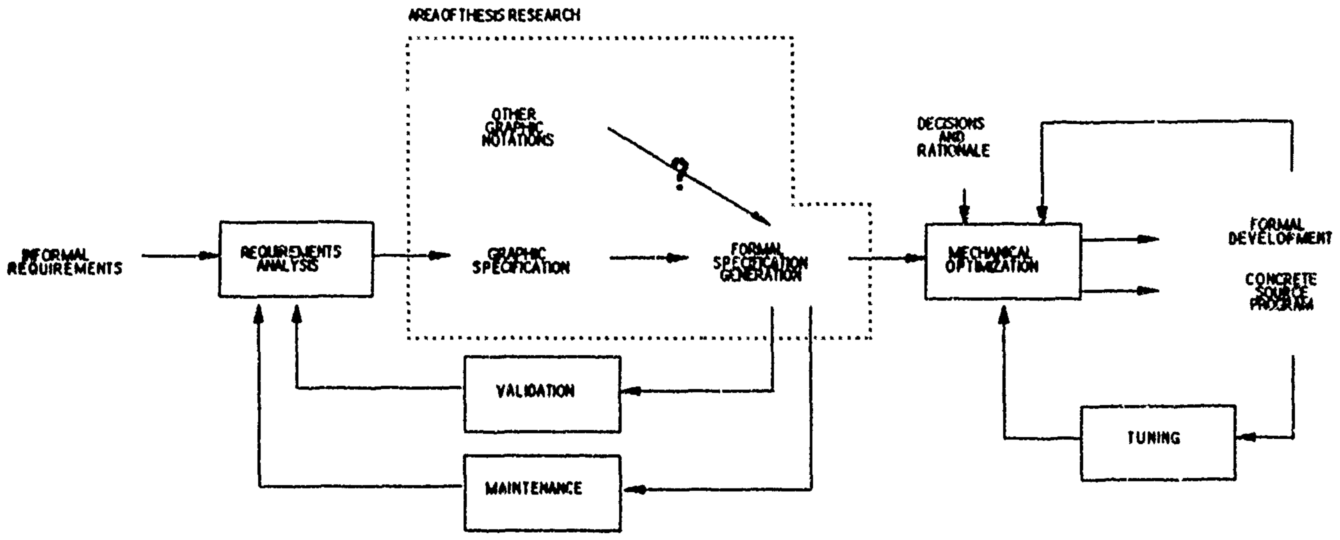
1.3 Assumptions

The graphic language to be developed in this thesis will be based on the Refine programming language. Refine is ideally suited for the formal specification of software and is currently being used by Rome Air Development Center to develop the Knowledge Based Software Assistant (12). Since Refine is already being used, any enhancements to the language may be directly applicable to Air Force development projects.

A secondary reason for choosing Refine as the foundation for this thesis effort is that Refine is a commercial product. Because Refine is commercial, it should be relatively easy to obtain technical support if needed. The choice of Refine as a foundation is also an attempt to ensure the broadest possible applicability for this research since Refine's commercial nature virtually guarantees a much wider distribution than a comparable academic product.

1.4 Scope

This research presents the desirable characteristics of a graphical language, a grammar for the language, a translational grammar to convert a specification between its graph-



ADVANTAGES OF MODIFIED LIFECYCLE:
SPECIFICATIONS EASY TO UNDERSTAND AND CONSTRUCT
REDUCED TRAINING NECESSARY TO USE SPECIFICATION TOOLS
POSSIBILITY OF INCREASED USER INTERACTION/PARTICIPATION

DISADVANTAGES:
INCREASED COMPUTATIONAL OVERHEAD DUE TO SOPHISTICATED INTERFACE

LEGEND

$\boxed{\text{NAME}}$	LIFECYCLE ACTIVITY
NAME	LIFECYCLE PRODUCT

Figure 1.4. Modified Program Transformation Lifecycle

ical and Refine representations, and a preliminary assessment of the effectiveness of the graphical language. No attempt is made to actually implement the graphical language because of the time constraints on this thesis. However, all of the system independent information necessary to implement this graphical language is provided.

Because no implementation is being attempted, no detailed assessment of the graphical language is possible since users will not have "hands on" experience with the language. A detailed user survey should be performed after the language implementation is complete to determine the effectiveness of the graphical language, but this survey is beyond the scope of this effort.

1.5 Approach/Methodology

This research effort attacks the problem of developing a graphical language for the formal specification of software in three steps. The first step establishes a set of objectives for such a graphical language. These objectives are based on current research in the areas of graphic programming and design notations, perception and learning processes of software professionals, and formal language theory. Information concerning existing graphic notations guided the development of the graphical language by providing a foundation for the development a new graphical language and by identifying the pitfalls associated with the development of such a language. Research into the characteristics of the perception and learning processes of software professionals is critical to this effort because this new graphic language must be easy to learn and use; ignoring available information about the perception and learning processes of these individuals could result in the development of a language that is neither. Information concerning formal language theory is important to ensure that the graphical language developed can be efficiently implemented using as many traditional compiler techniques as possible.

The second step decomposes a formal specification language, called SF (5), into basic classes of language constructs. A graphical language should use the minimum number of symbols possible to represent the specification language because, just as the Chinese language is unwieldy because of the number of symbols it uses, a graphical language that is too large will be difficult to understand and use. On the other hand, a graphical language that uses too few symbols is very likely to be ambiguous, again reducing the understandability of the language. Dividing the language into classes of constructs will make the design of an effective graphic representation much easier because each class can then be represented by a single symbol or a small set of closely related symbols.

The final step in this project designs a set of graphic symbols so that each set of specification language constructs can be represented by a single symbol or a small number of very similar symbols. This notation development phase will be guided by other popular graphic notations now used in software development in order to minimize the time a user would require to transition from an existing development notation to this formal specification notation.

1.6 Organization

This thesis is divided into seven chapters:

- Chapter 2 contains a survey of the current research related to this effort.

- Chapter 3 investigates the characteristics necessary for a graphical language to be an effective communication tool for humans while remaining processable by computers.
- Chapter 4 decomposes a simple specification language called SF and constructs a preliminary graphical specification language based on SF. The experience gained from working with the SF language is then applied to working with a more sophisticated specification language called Refine.
- Chapter 5 decomposes the Refine language into sets of language constructs that are representable by a graphical language and then proceeds to build a graphical specification language built upon these constructs.
- Chapter 6 discusses some of the issues associated with the conversion of existing graphical programming notations into graphical specification syntax.
- Chapter 7 presents the conclusions of this research and recommendations for continuing research in this area.

II. Survey of Current Literature

Balzer's program transformation lifecycle model is based on formal specification of software, the process of using mathematical expressions to describe a software system. Formal specification, however, is not a panacea for programmer productivity; the mathematical expressions used in formal specification are very difficult to understand. In fact, programmers frequently need special training in order to understand, use and create formal specifications. This literature search will survey current efforts to enhance formal specification languages with graphical interfaces that allow programmers to create and use these languages more efficiently and effectively.

Because the design of graphical interfaces for formal specification languages is a relatively new area of research, very little information about this research area exists. Therefore, this literature search surveys sources containing information which could be useful in the design of a graphical interface for a formal specification language.

2.1 Formal Specification Methods and Visualization

Berztiss (5:231-290) is one of the few researchers to use a graphical notation to enhance the understandability of formal specifications. In his article, Berztiss presents SF, a specification language of his own design. Using SF, Berztiss formally specifies a library system, an elevator control system, and a text formatter in order to demonstrate the utility of formal specifications in general and SF in particular.

After he develops the framework for discussing formal specification methods, Berztiss shows how graphic notations commonly used by programmers, such as data flow diagrams, which depict the flow of information from one program routine to another, entity relationship diagrams, which depict the interrelationships between the information structures being manipulated by the program, and state transition diagrams, which depict the movement of a program from one state to another, can be used to develop formal specifications. Berztiss concludes, however, that these programming notations will never serve as more than supporting documentation for formal specifications. Berztiss takes this position because of the amount of supporting text, in the form of labels and additional notations, necessary to make the graphic notations stand-alone documents.

2.2 Influence of Visual Technology on the Evolution of Language Environments

Ambler and Burnett (1:9-22) present a broad survey of visual programming environments. Although the authors' survey does not discuss formal specification, the survey is important in the development of a graphical notation for formal specifications because the survey allows the reader to compare and contrast the different features of each environment, enabling the reader to select those features that would enhance any particular graphical notation.

Ambler and Burnett divide their survey into three areas, visual user interfaces, visual editing, and visual languages, and discuss the common characteristics of systems in each area. Ambler and Burnett supplement this theoretical framework with descriptions of over a dozen visual systems to show how these common characterstics manifest themselves in different environments. This comparison and contrast is valuable to this research effort because it highlights the desirable and undesirable features of graphical design and development environments.

2.3 Visualizing Program Designs Through PegaSys

While Ambler and Burnett present a broad survey of the visual programming environments, Moriconi and Hare (21:72-85) focus on one visual environment, PegaSys, and demonstrate the benefits of using graphic images to represent computer programs. Like Ambler and Burnett, Moriconi and Hare do not address formal specification; nevertheless, the authors' system implements some characteristics that would be highly desirable in a graphic notation for a formal specification language.

One of the most important features of such a graphic notation is a simple but consistent notation for all program constructs. PegaSys' notation for its program constructs is an excellent example of a sophisticated notation that retains the simplicity and consistency necessary to make the notation understandable and usable by programmers with little training. PegaSys also enables the programmer to view a program from several different perspectives, such as showing the flow of information through the program or showing the transition of the program through different states. This ability to display a program from different perspectives is a powerful tool for increasing the understandability of graphic notations since the ability enables the programmer to view a program from the perspective most natural to him or her.

2.4 Toward Software Metrics for Visual Programming

While Ambler and Burnett and Moriconi and Hare present excellent descriptions of a variety of visual systems implementing a vast number of features, neither pair makes any attempt to establish the characteristics necessary for a "good" visual system. Glinert (15:425-445) fills this void by proposing a rigorous mathematical framework for determining the effectiveness of a visual system. Such a framework is absolutely necessary if the developers of a graphic notation are to evaluate a notation and improve that notation to increase its effectiveness.

Since individuals naturally perceive structures in different ways, the probability of developing a single metric for measuring graphic notations is very low. It is in this light that Glinert proposes the definition for a metric for visual systems: "The key to making the computing environment both accessible to the novice and appealing to the expert user, is to maximize its coefficient of attraction while minimizing its coefficient of repulsion" (15:426). Adhering to his definition, Glinert's evaluation method is based on enumerating the features of a particular system, recording the reactions of a group of users to the features, and then calculating a weighted composite score for the system based on these reactions. After developing this evaluation method, Glinert applies the method to a variety of visual systems, effectively conveying the relative strengths and weaknesses of each of the systems.

2.5 Mapping the Design Information Representation Terrain

In order to construct a graphical notation for a formal specification language, one must be able to decompose the language into its elementary parts so that each part can be associated with a unique graphical symbol. One of the major obstacles, then, is deciding how to decompose a particular language. Webster (26:8-23) addresses this problem by investigating the common mechanisms used by specification languages to represent program design information. To clarify his descriptions of these mechanisms, Webster describes a number of specification languages and points out the similarities and differences between the mechanisms these languages use to encode design information.

Webster concludes his article by attempting to organize the specification languages into "families" based on their relative characteristics. These families could be particularly useful in the design of a graphic notations because one would like to design a notation that is useful in a variety of applications; this family structure presents a useful structure for planning and analyzing the portability of a particular notation.

2.6 Literature Search Summary

Although there has been relatively little research into graphical notations for formal specification languages, this survey has shown that the tools do exist for designing and constructing such a notation. Although Berztiss is prepared to restrict graphic notations to strictly supporting formal specifications, Ambler and Burnett and Moriconi and Hare describe tools that could eliminate the problems that prompted Berztiss's restriction. Webster offers a useful taxonomy for planning the portability of a new graphic notation. Finally, after the notation is designed and implemented, Glinert proposes a methodology to assess the effectiveness of the notation and to indicate the features of the notation that require improvement.

This survey has attempted to synthesize the state-of-the-art in graphic notations for formal specifications through an examination of the tools available in closely related research areas. The articles surveyed have shown a wide variety of graphical notations already implemented for conventional programming systems. The challenge now is to modify these existing notations so that the notations can effectively represent a formal specification, thereby enhancing the understandability and usability of formal specifications and increasing programmer productivity.

III. Objectives for a Graphical Specification Language

Before beginning any project, it is vital to establish a set of requirements and objectives for the project to ensure the high quality and usefulness of the final product. The design of a graphical specification language is no exception. As stated in the introduction, this language design effort has two important objectives:

- Devise a method of presenting specification information that is clear, concise, and understandable by a variety of users.
- Devise a graphical specification environment that makes graphic formal specifications easy to create and manipulate.

For ease of use, these two objectives shall be referred to as language clarity and language utility respectively.

Using language clarity and utility as a foundation, it is now possible to further refine these characteristics in order to establish a more concrete set of objectives to guide the development of a graphical specification language. Fitter and Green (14) propose a number of ways to improve the clarity of a graphical language. One suggestion is to use "perceptual coding" whenever possible (14:257). Although they do not provide a definition of perceptual coding, Fitter and Green do provide a number of examples, such as color coding of electrical wires and terminals, and the various sounds associated with telephone systems, i.e., dialing, ringing, and busy signals. Fitter and Green's point in this suggestion is that programmers are taught early in their careers to use spatial cues, such as indentation, double spacing, and capitalization, in their programs to make those programs more understandable and readable; graphical languages offer an opportunity to take this non-textual representation a step further through the use of pictures in addition to text to represent computer instructions. Fitter and Green are emphasizing that it is not the symbology that is important in making specifications understandable, but it is how that symbology is presented and how the various symbols interact. Put simply, Fitter and Green suggest using spatial coding and inter-symbolic relationships as much as possible to present information to the user.

Fitter and Green also suggest that restricting users to easily understood objects increases the clarity of a graphical language. This point is intuitively appealing since the addition of an obscure object to a specification would tend to obscure the meaning of the entire specification.

Fitter and Green conclude their suggestions to improve language clarity with the recommendation that "different programs should be perceptually as different as possible" (14:259). This recommendation is also intuitively appealing for obvious reasons: consider for a moment a traffic light that used only one light changing colors instead of three separate lights and instead of using three widely separated colors, i.e., red, yellow, and green, this hypothetical traffic light used red for stop, a reddish orange for go, and orange for yield. This traffic light configuration would indeed cause widespread confusion because the three symbols for radically different activities are very similar. The same situation would hold for graphical specifications: similar representations for widely differing activities would cause significant confusion and uncertainty among the individuals working with the specifications, thus decreasing the individuals' productivity.

Davis and Keller (10) point out the importance of data abstraction in the clarity of a graphical language. Data abstraction refers to the ability to represent a number of operations or a complex data structure with a single named entity, typically a function, procedure, or macro for operations and a record, array, or list for data structures. As Davis and Keller state, "It is cumbersome to deal with graphical programs consisting of single very large graphs" (10:31).

Just as they did with language clarity, Fitter and Green make a couple of suggestions for improving the utility of graphic languages. First, they suggest the need for a language to provide mechanisms to "allow easy and accurate revision" of graphical specifications (14:258). In other words, the language symbology must be modular enough that a graphic editor can be constructed to manipulate language constructs. Fitter and Green's second suggestion is closely tied to their first: "reveal the underlying mechanisms and be responsive to manipulation" (14:258). As with their first, this suggestion requires a level of modularity of the graphical specification language to accommodate easy manipulation. In addition, however, both of these suggestions make demands of not only the graphical language but also of the language's supporting environment, namely the need to provide an interactive graphical editor capable of manipulating the graphical specification language.

Another feature that would greatly enhance the utility of a graphical specification language is a mechanism that would facilitate or even encourage the reuse of existing software components. One of the primary reasons for attempting to develop a graphical specification language is to greatly increase programmer productivity. There is no better way to improve programmer productivity than to help the programmer to avoid performing the work in the first place. This is the fundamental idea behind the idea of software component reuse. A mechanism for facilitating such reuse would require graphical language features that would allow the user to conveniently construct components which could be reused in future projects. The mechanism would also require the language environment to provide a "librarian" of sorts that would assist the user in cataloging and then retrieving the software components. Ingalls, et al, (16) demonstrate the use of such a mechanism in their graphical language Fabrik. In Fabrik the user constructs programs out of primitive software components contained in "kits". Although the librarian in this case is a passive filing system, Fabrik demonstrates the feasibility of software construction through the use of kits.

One more utility enhancing feature would be some mechanism for supporting data encapsulation. The term data encapsulation refers to the concept of combining a data structure and its associated operations into one program structure. Popularized by object oriented languages such as Smalltalk, data encapsulation provides the programmer with one more tool to manage program complexity. The inclusion of a data encapsulation mechanism in a graphical specification language along with mechanisms promoting software component reuse would open the opportunity for programmers to construct domain specific "toolkits". Once these toolkits are constructed, they could potentially be used by both programmers and non-programmers for rapidly assembling more sophisticated components or even entire specifications. Clearly, these mechanisms offer great potential for increasing programmer productivity.

Although all of these objectives are desirable in a graphical specification language, there may be cases where objectives conflict. In the case of such a conflict, the objectives associated with language clarity will take precedence over objectives associated with language utility. As stated earlier, the most significant problem with the use of formal specifications is the lack of understandability of these specifications. In addition, understandability affects not only the programmers and specifiers working with the specifications, but also affects the customers who need assurances that the specifier and programmer are building what the customer actually needs. Language utility, on the other hand, affects only the programmer. Powerful methods already exist for manipulating specifications in their textual form, thereby reducing the need for providing graphical manipulation tools. This subordination of language utility objectives, however, in no way establishes the graphical specification language as a read-only system. The graphical specification language system should possess powerful facilities for the creation and manipulation of graphical specifications. This prioritization does, however, allow for the rational, coherent, and consistent resolution of any conflicts between objectives.

These objectives, then, constitute a framework within which the development of a graphical specification language may proceed. To facilitate the evaluation of graphical specification languages against these objectives, it is necessary to develop a more compact form of presentation. A concise presentation of the objectives for a graphical specification language grouped by their association with language clarity and utility is contained in Figure 3.1.

- Clarity of Language
  - Abundant use of perceptual coding of information
  - Restrict users to easily understood objects
  - Graphical specifications that differ conceptually should also differ visually
  - Mechanisms to facilitate data abstraction
- Utility of Language
  - Allow easy and accurate revision of specifications
  - Mechanisms to facilitate and encourage software component reuse
  - Mechanisms to facilitate data encapsulation

Figure 3.1. Objectives for the Development of a Graphical Formal Specification Language

IV. A Graphical Representation of the SF Specification Language

SF is a simple language developed by Alfs Berztiss for the formal specification of information-control systems (6). SF is based on the concepts of sets of elements and functions for querying and manipulating those sets. Because of its simplicity and modularity, SF is an ideal foundation for developing a portable graphical specification language. This graphical language will be developed in two separate steps: first, SF must be decomposed into its basic constructs and mechanisms so that each of these essential language parts can be represented by a unique graphical symbol; second, a set of graphical symbols must be developed that is easy to use and understand while unambiguously representing the SF language; and third, a mapping function between graphical symbols and language constructs must be developed so that an SF specification can be converted into its graphical representation and vice versa.

4.1 Decomposition of SF

4.1.1 Decomposition Method In the world of conventional programming languages, compilers are the predominant tool for the translation of a program from one language to another. It would seem logical then to look to compiler theory for methods to aid in the decomposition of specification languages.

One tool from compiler theory that is well suited to language decomposition is the parse tree (13:532-533). In conventional language compilation, the compiler constructs a parse tree of some source program and then uses that tree to construct a program equivalent to the original in some destination language. In the same way, a parse tree can

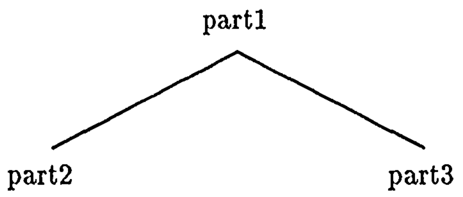


Figure 4.1. Parse Tree Example

be constructed for a particular specification language to aid in the identification of basic language constructs and to assist in the translation between specification representations.

The parse tree for any particular language can be easily constructed from the Backus-Naur Form (BNF) of the language's syntax. The BNF description of a language's syntax consists of a series of statements of the general form:

$$< part1 >::=< part2 >< part3 >$$

To construct a parse tree from a language's BNF description, start with the first BNF statement in the description. Place the term at the left of the ::= symbol at the root of the tree. Then place the terms at the right of the ::= symbol on branches descending from the first term. For example, consider the BNF statement presented earlier:

$$< part1 >::=< part2 >< part3 >$$

The parse tree in Figure 4.1 was derived from this statement using these tree construction operations. Now suppose that this example language contains another BNF statement:

$$< part2 >::=< part4 >< part5 >< part6 >$$

New BNF statements are added to the parse tree by locating the term at the left of the

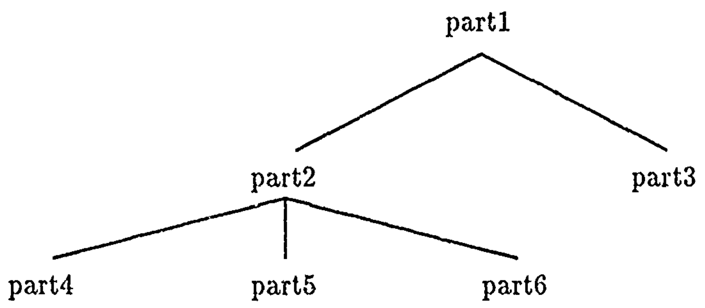


Figure 4.2. Modified Parse Tree Example

::= symbol on the existing tree and the connecting the terms at the right of the ::= symbol below the left term. Figure 4.2 shows the parse tree including the new BNF statement.

Using parse trees as an aid in decomposing a specification language has several advantages. First, because the parse tree is based on the syntax of the specification language, the parse tree contains all of the information necessary to recreate the specification that it was generated from. If the graphical specification language is created by manipulating the subtrees of the parse tree, then the formal specification can be recreated from the graphical specification by returning the parse tree to its original form and then traversing that tree. This property meets the objective of being able to create a graphical specification out of a text specification and vice versa.

Parse trees are also valuable aids in a language decomposition method because they graphically illustrate the relationships between different language constructs. Decomposition using parse trees, then, becomes a matter of creating the parse tree for a particular language and then selecting suitable parse subtrees representing major language constructs. The selection of the subtrees is based on the objectives for the graphical languages discussed earlier, on major language constructs described in language descriptions, and on

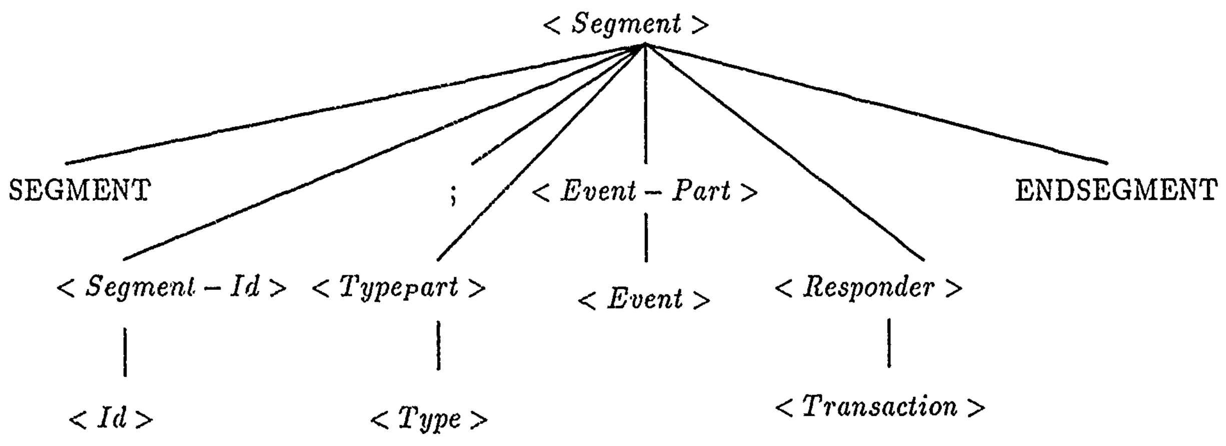


Figure 4.3. Partial Parse Tree for SF

the judgement of the language designer.

Using parse trees to decompose a specification language also aids the designer in determining the levels of abstraction at which the user may choose to view a specification. Parse trees aid this determination because of their graphic illustration of language construct interrelationships. Specification abstraction using parse trees becomes simply a matter of hiding subtrees until they are demanded by the user.

4.1.2 SF Decomposition As stated earlier, SF is a simple language designed for the formal specification of software. Because of its simplicity, SF is an ideal language to use for demonstrating the decomposition of a language using parse trees. As a starting point, the root of the parse tree for the SF language is shown in Figure 4.3. The first major SF construct that becomes apparent after looking at the parse tree, as well as language references, is the segment. The segment is SF's primary mechanism for modularizing software specifications. This modularity breaks the specification into pieces to make it

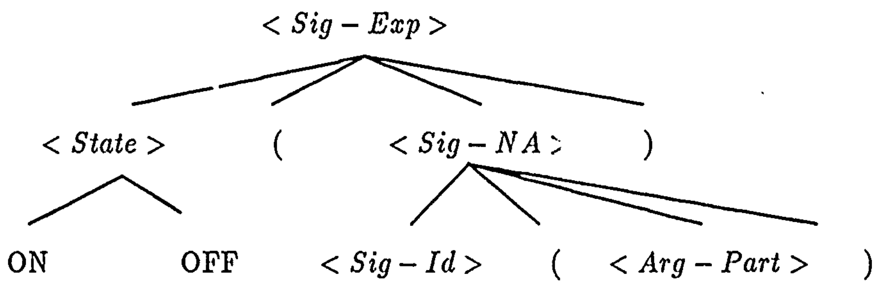


Figure 4.4. Partial Parse Tree for SF

easier to use by the specification developers and maintainers. Such modularity is also necessary to make the graphical language easy to use. The segment, then, seems to be an excellent construct to include in the graphical language.

Two levels below the segment in the parse tree are three more SF constructs that are ideal candidates for inclusion into the graphical language: the type, the event, and the transaction. The type is SF's data encapsulation mechanism. The type also provides a set of functions that can perform non-destructive inquiries on the data in the type. The event is SF's mechanism for manipulating the data contained in the type. Finally, the transaction is SF's mechanism for responding to the changes in internal and external conditions represented by changes in the states of system signals, which will be described later. All of these constructs further refine the specification modularity instituted by the segment construct. Each of these constructs is well suited for inclusion into a graphical language because each contributes to the initial objective of allowing the user to control the amount of specification detail that is displayed at any time.

Figure 4.4 contains another partial parse tree. This tree shows the derivation of the signal in the SF language. SF uses the signal to implement all interevent and intersegment communication. Although this construct resides fairly deeply in SF's parse tree, the signal is well suited for inclusion into a graphical language not because of its modularizing capabilities but because this construct explicitly defines communication between the various constructs of a specification, thus making the specification easier to understand and to use.

The signal illustrates an important point in the use of parse trees as an aid in decomposing a language: the use of a parse tree by an individual unfamiliar with the language being decomposed can result in the omission of vital language constructions. Segments, types, events, and transactions can all be derived fairly easily from the SF parse tree because their large subtrees indicate that they encapsulate substantial amounts of information. Signals, however, appear very close to the leaves of the SF parse tree. If the size of the subtrce was the only criterion for selecting the essential parts of a language, signals would likely be rejected as an essential language construct, but the signal is an invaluable tool for explicitly defining inter-construct communications within the specification. The point is that a parse tree can be a valuable aid to a designer who is at least familiar with the language being decomposed.

4.1.3 Why Stop Here? The SF language contains many more features beyond the five that were selected for graphical representation. A logical question would be "Why stop the decomposition at this point when so much of the language remains?" The primary reason is based on the limitations of human processing. This decomposition has produced a number of essential language constructs that represent most of the SF language while remaining small enough to be effectively understood and manipulated by humans.

A secondary reason for stopping the decomposition at this point relates back to the objective of having the user control the amount of specification detail presented at any time. This decomposition allows the user to select three levels of abstraction: the specification level, which displays a number of segments; the segment level, which displays the events, types, and transactions in the segment, and the event/type/transaction level which contains the specification for the behavior of the appropriate construct. Too many levels of abstraction may force the user to spend an inordinate amount of time controlling the amount of detail being displayed, impacting the graphical language's ease of use. However, as Glinert points out (15), individual learning and working styles differ, so the actual number of abstraction levels that are allowed must be based on the features of the language being defined as well as the experience and judgement of the graphical language designer.

4.2 Development of a Graphical Representation for SF

4.2.1 Considerations in Graphical Representation Development The major objective in the design of this language has been to create a graphical language that is easy to learn, understand, and use. Conventional software development methods now use a variety of graphical notations to increase the understandability of conventional software designs. Data flow diagrams, entity-relationship diagrams, and state transition diagrams seem to be the most prevalent of these graphical notations (25). By using these popular notations as a foundation for development, a graphical specification language can capitalize on the existing familiarity with these notations to reduce the effort required to learn and use the new language.

Another factor that should be kept in mind during the design of this graphical language is the size of the set of symbols used in the language. This factor is driven almost entirely by the results of the language decomposition, but another look at this critical factor is well advised during the development of the language symbology. Using too few symbols in the language can result in ambiguities that will make the language difficult for humans to learn and difficult or impossible for computers to process. On the other hand, using too many symbols in the language increases the language complexity which in turn reduces language understandability and ease of use.

4.2.2 Language Foundations The development of this graphical language is based on the previous decomposition of the SF specification language. The decomposition identified five essential SF language constructs: the segment, type, event, transaction, and signal. In a brief review of the functions of these language constructs, the segment is SF's specification modularizing mechanism. The type is SF's data encapsulation mechanism. The event is SF's data manipulation mechanism. The signal is SF's interevent and intersegment communication mechanism. Finally, the transaction is SF's signal response mechanism.

4.2.3 Language Development With the SF decomposition complete and the graphical language objectives established, the stage is now set to begin the development of a graphical specification language based on SF. As stated earlier, the symbol selection for this language will be guided by the existing notations used in data flow diagrams, entity relationship diagrams, and state transition diagrams.

The segment is most closely related to a high level process in the data flow diagram (DFD) notation. Although the DFD notation makes no formal distinction between high level and low level processes, in practical use, a high level process represents a conglomeration of a number of lower level processes whereas low level processes represent a single, well-defined activity. A segment performs a function very similar to the high level process in that it represents a conglomeration of a number of lower level specification functions. A significant difference between the two, however, is that processes are capable of manipulating program information whereas segments have no such processing capability. In DFDs, processes are represented by circles. Because of the segment's similarity to processes, a symbol resembling the circle should be used, but because of the differences between processes and segments, the circle itself should not be used. Therefore, an ellipse shall be used to represent the segment. To further distinguish segments apart from one another, a text label shall be included inside the ellipse identifying individual segments. This text label shall be the same as the segment identifier in the SF specification.

The type in the SF language is very similar to both the data store in DFDs and to the entity in entity relationship diagrams (ERDs). Both data stores and entities are data encapsulation mechanisms which cannot destroy or modify the information that they hold. Both the data store (25:253) and the entity (20:30) are represented by rectangles in their respective notations. Because of the close similarities between the type and the data store and entity, the type shall also be represented by a rectangle. Like the segment, the type shall also contain a type identifier inside the rectangle to assist in distinguishing between individual types.

The event, the data manipulation mechanism in SF, is similar in purpose to the low level process in DFDs as described earlier and to events in state transition diagrams (STDs). The event in SF and the event in STDs both represent a change of conditions within a system. Moreover, the event in SF also seems to be a close relative of the process because both the SF event and the process represent mechanisms that actively modify data structures. However, the SF event seems to be conceptually closer to the DFD process than to a STD event since the SF event and the process both represent an active modification of system conditions, where the STD event simply implies that some change in conditions has taken place. Because of their conceptual similarities, the SF event shall be represented by the same symbol as the process, a circle, in this graphical language. The circle shall also contain an event identifier.

The transaction has no counterpart in any of the existing notations being used. Because of its mission as a responder to internal and external conditions through the use of signals, it occupies a unique position in the SF environment. As such, a unique symbol should be used to designate the transaction construct. Because it is distinctive and easy to draw, a diamond shall be used to represent the transaction. Although a diamond is used to represent relations in ERDs, the difference in the use of the diamond in this specification language will be sufficient enough that the danger of confusion between the two constructs will be minimal. In SF, transactions are also capable of time dependent activities. Because transactions do not have individual identifiers in SF, the expression used to activate the transaction based on a time constraint will be included within the transaction diamond. When an SF transaction responds to a particular signal, it responds by triggering a particular event or several events. To represent this feature in the graphical language, an arrow originating from a transaction diamond and terminating at an event shall designate that the event is triggered by the response of the transaction.

The signal's closest relative in the existing notations is the data flow in the DFD. Just as the data flow represents the movement of information from one process to another, the signal represents the movement of information from one segment to another or from one event to another. Because of its close similarity in function to the data flow, the signal shall use the same symbology as the data flow, an arrow originating from the sending construct and terminating at the receiving construct. SF signals have identifiers, so arrows shall be annotated with the signal's identifier. Signals in SF may also carry arguments, either typed or untyped. If the argument is typed, then the convention of a rectangle shall be maintained by placing a rectangle on the arrow and placing the argument names within the rectangle. If the arguments are untyped, then a rounded rectangle shall be used in the same way. Signals can take on Boolean values, i.e., true or false. Since transactions only respond when a signal takes a particular value, a signal terminating at a transaction shall be annotated with the proper value to illustrate the value to which the transaction reacts. Since signals can be modified by both events and transactions, some method is required to designate the state of the exiting signal. If the emerging signal's state is deterministic, i.e., the signal is guaranteed to have a particular value when exiting an event or transaction, the signal shall be annotated with the appropiiate value. If the signal's state is nondeterministic, i.e., the signal's value can vary depending on system conditions, then no state annotation will be included.

Diagrams of all of these constructs are contained in Figure 4.5.

4.3 Graphical Language Syntax

In conventional parse trees, the only terms in the tree that actually appear in the specification are terminal terms, i.e., the terms that appear as the leaves of the parse tree. The parse tree for this graphical language uses semi-terminals, terms which are not

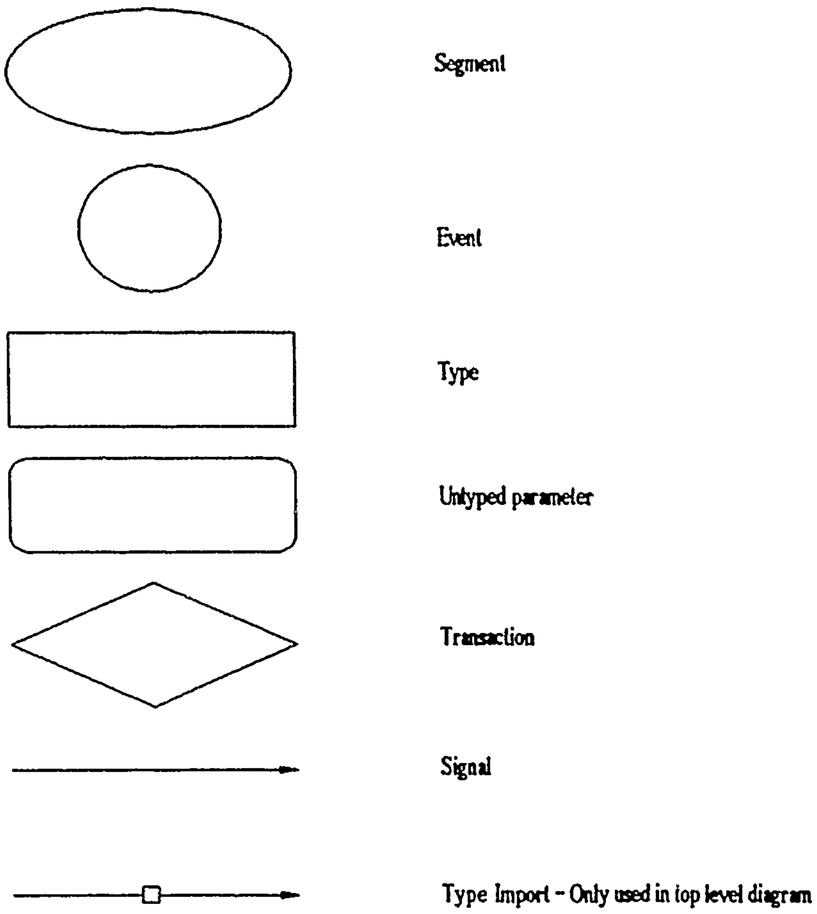


Figure 4.5. Graphical Specification Language Symbology

terminals but may still appear in the specification. The reason for using these semi-terminal terms is founded on allowing the user to control the amount of detail being displayed at any time. As a result of endowing the user with this control, a term may be a terminal, i.e., none of the term's subtree is displayed, until the user demands more information about the term. At this time, the term is no longer a terminal and disappears so that it can be replaced by the structures in the term's subtrees.

The syntax of the graphical language is identical to the SF syntax, then, except that the keywords for a particular SF construct are replaced by semi-terminals representing the corresponding symbols in the graphical language. For example, Figure 4.6 contains the parse tree of a typical SF segment. In an SF specification, the entire parse tree is always present, thus allowing the user no control over the amount of information presented to him or her. In the graphical language however, when the user initially "opened" a specification, the user would be presented with only the segment level information, represented by the parse tree in Figure 4.7 and the graphical symbology in Figure 4.8. If the user so desires, he or she can then request more detailed information about the segment which would result in the expanded parse tree in Figure 4.9 and the revised graphical symbology in Figure 4.10. In this way, the user can explicitly control the amount of detail displayed at any time thus allowing him or her to understand a specification more easily and clearly by removing all unnecessary information from the display.

4.4 Library Control System Example

Up to this point, this research has discussed the SF-based graphical specification language in fairly abstract terms. The conversion of an existing SF specification for a

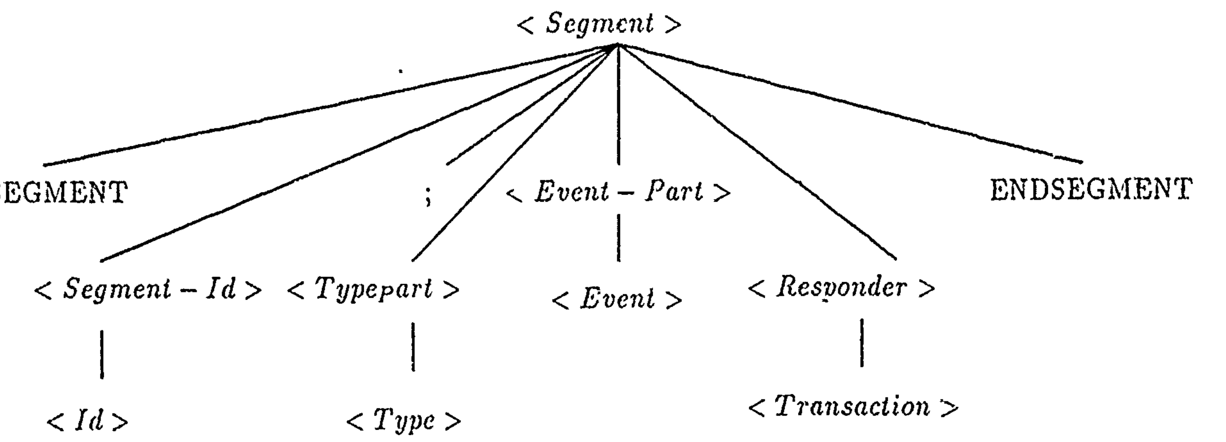


Figure 4.6. Partial Parse Tree for SF

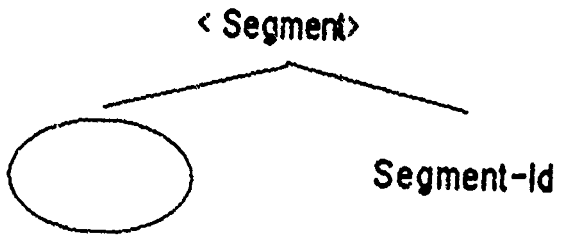


Figure 4.7. Segment Level Parse Tree for SF Graphical Language

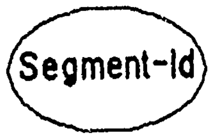


Figure 4.8. Segment Level Graphical Specification Representation

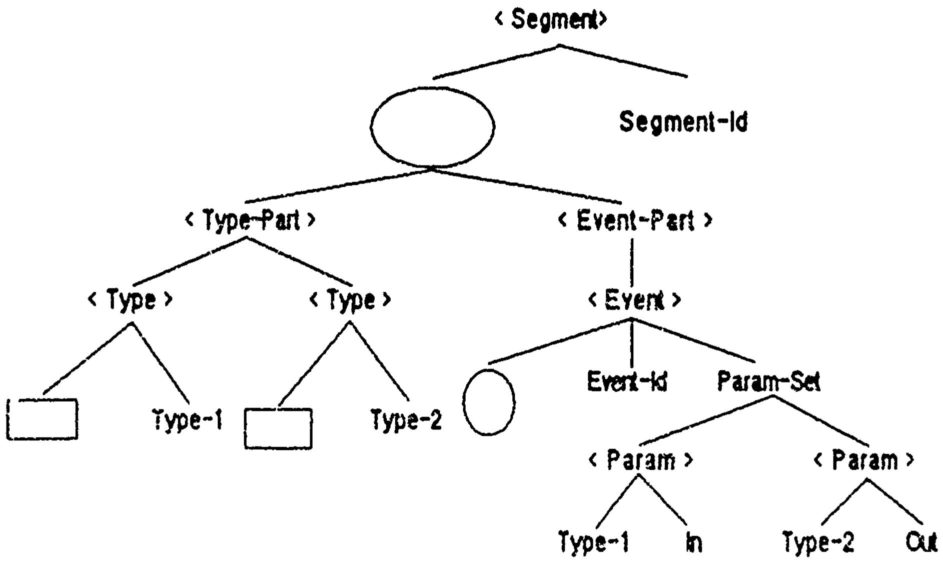


Figure 4.9. Event/Type Level Parse Tree for SF Graphical Language

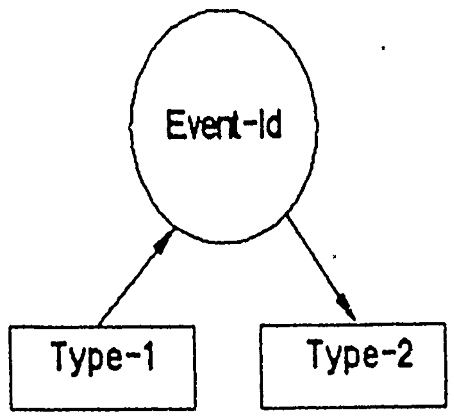


Figure 4.10. Event/Type Level Graphical Specification Representation

library control system (3) will now be demonstrated to illustrate the techniques involved in the conversion as well as to highlight the advantages of a graphical representation over a purely textual representation. In order to emphasize the advantages of graphical specifications, each graphical specification is accompanied by the portion of the SF specification from which it was derived. The complete SF specification for the library control system is contained in Appendix A.

The specification for the library control system contains two segments, Titles and Copies. Each of these segments is represented by an oval with the segment name within the oval. The Titles segment imports the signals Add˜title, Drop˜title, and Move˜title from the Copies segment; these intersegment signals are represented as arrows originating from Copies and terminating at Titles. The Copies segment imports the Title type from Titles, represented by an arrow originating at Titles, terminating at Copies, and marked with a

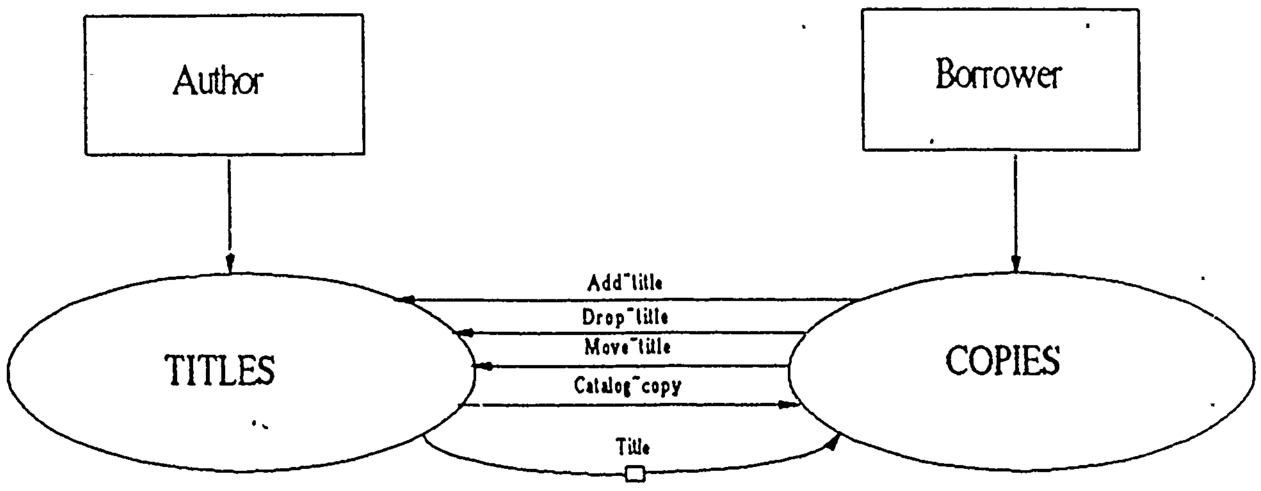


Figure 4.11. Graphical Representation of Segment Level Library Specification

small square to distinguish the type import from an intersegment signal. Titles imports a type Author which is implied to be predefined. This type import is represented by a rectangle containing the type identifier, Author, and an arrow leading from the rectangle to the Titles segment. Likewise, the Copies segment imports a supposedly predefined type Borrower, an import represented in the same manner. A diagram of these segment-level interactions and the corresponding SF specification are contained in Figures 4.11 and 4.12 respectively.

Even at this high level of specification, the graphical language reveals several basic characteristics of this specification. First, all intersegment communications are immediately apparent. These communications reveal intersegment dependencies which are extremely important for the specification maintainer to know so that he or she can be aware

```
SEGMENT Titles;

IMPORTED SIGNALS Add~title, Drop~title, Move~title;

EXPORTED SIGNALS Catalog~copy;

IMPORTED TYPE Author ENDTYPE;

TYPE Title : T(SUBSETS:INCAT,HASBEEN);

SECONDARY SETS- Subject~area: Area;

FUNCTIONS- title~text : T -> Text;
authors    : T -> Author-set;
subjects   : T -> Ares-set;
ENDTYPE;

SEGMENT Copies;

IMPORTED SIGNALS Catalog~copy;

EXPORTED SIGNAL Drop~title, Move~title, Add~title;

IMPORTED TYPE Title:T  ENDTYPE;

IMPORTED TYPE Borrower:B  ENDTYPE;

TYPE Copy : C;
FUNCTIONS- book~id: C -> T;
borrowed: C -> Boolean(false);
last~out: C -> B(nil);
books~out: B -> Integer(0);
limit:   -> Integer(0);
ENDTYPE;
```

Figure 4.12. SF Representation of Segment Level Library Specification

of any non-local side effects of a specification change. The graphical language also reveals one possible error in this particular specification. Although it is possible to declare a type as predefined in the SF language, Berztiss has chosen not to do so for the Author and Borrower imported types. Technically, then, undefined types are being imported into the specification segments. This possible error is made readily apparent by the presence of non-predefined types outside of a segment.

Within the implemented graphical language, this top level diagram would be the first diagram presented to the user. If the diagram was too large to fit within the screen area, the user would have the ability to scroll the screen in order to view the entire top level diagram. Now suppose that the user requested more detail for the Titles segment.

The Titles segment contains three transactions. Recall that these transactions are represented by diamonds containing a time expression governing the time dependent behaviors of the transaction. The first transaction is always active, denoted by the @T˜min.now expression, and responds when the Add˜title signal becomes ON. The Add˜title signal carries the untyped arguments newcopy and book, represented by a rounded rectangle containing the argument identifiers. When the Add˜title signal becomes ON, the transaction responds by activating the Add˜title event, represented by an arrow pointing from the transaction to the circle representing the Add˜title event. After responding, the transaction changes the Add˜title signal from ON to OFF, represented by an arrow exiting the transaction annotated with OFF. Figures 4.13 and 4.14 contain the graphical and SF specifications for this transaction.

Once the Add˜title event is triggered, it causes a series of actions. First, the event

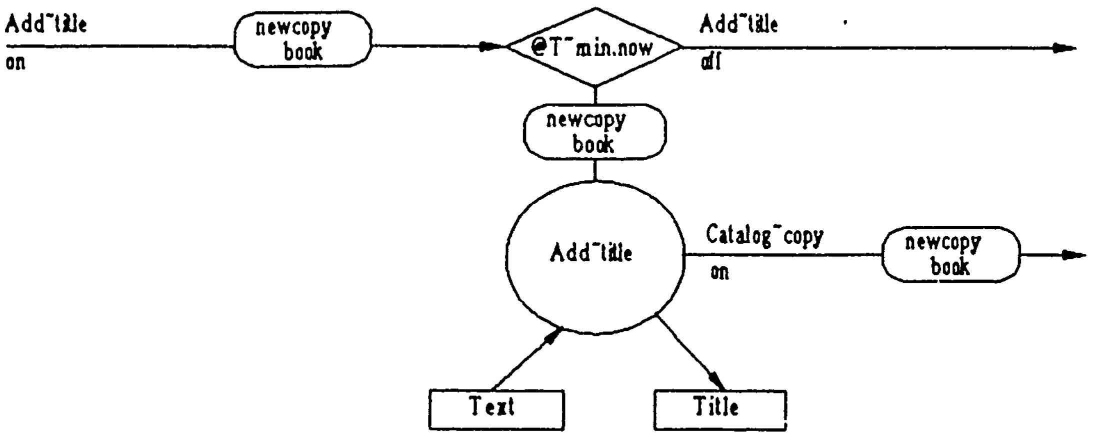


Figure 4.13. Graphical Representation of Transaction 1 in Titles Segment

```
(* Transaction 1 *)
TRANSACTION;
        @(T˜min.now):ON(Add˜title(newcopy,book))OFF:
                PROMPT(Add˜title:newcopy,book);
ENDTRANSACTIONS;
```

Figure 4.14. SF Representation of Transaction 1 in Titles Segment

checks to see if its preconditions hold. For Add~title, the precondition involves an inquiry into the set Text. This inquiry is represented by an arrow leading from the rectangle representing the type Text to the event. If the event's preconditions hold, then the event modifies the book item in the Title type, represented by the arrow leading from the event to the type. Finally, the event sets the Catalog~copy signal to ON and passes the untyped newcopy and book arguments with the signal. This signal modification is represented by an arrow leaving the event, annotated with ON and a rounded rectangle with the argument identifiers.

The convention used throughout this graphical language for representing event/type interactions is that an arrow leading from a type to an event represents a non-destructive query of the type by the event. An arrow leading from the event to the type represents a modification of the information contained by the type. A double headed arrow between the type and the event represents both a non-destructive query and an information modification by the event.

The second transaction works in much the same way as the first; Figures 4.15 and 4.16 contain the graphical and SF specifications for this transaction. The transaction is always active, denoted by the @T~min.now annotation, and responds when the Drop~title signal becomes ON. This construction is represented by a diamond containing the @T~min.now annotation with an arrow entering the diamond from the left side annotated with the signal identifier, Drop~title, and the signal state which triggers a transaction response, ON. This arrow also contains a rounded rectangle representing the untyped argument carried by the signal, book. Another arrow exits the transaction diamond from the right. This arrow is again annotated with the signal's identifier, Drop~title, and the signal's state upon

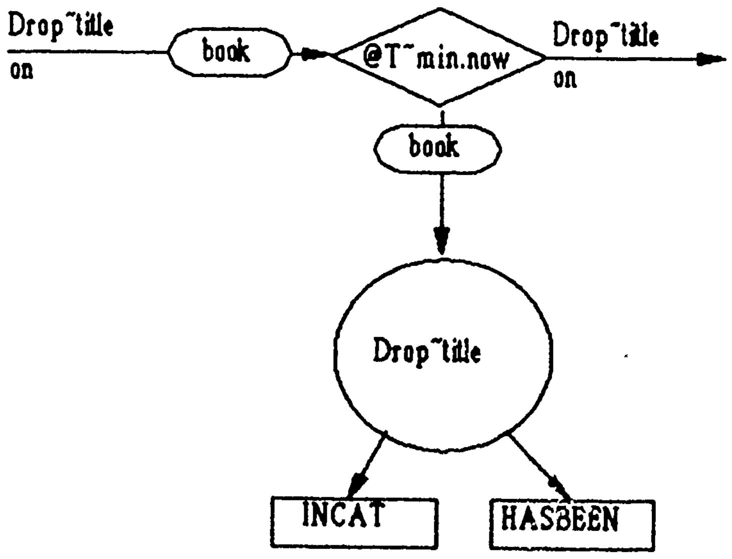


Figure 4.15. Graphical Representation of Transaction 2 in Titles Segment

```
(* Transaction 2 *)
TRANSACTION;
        @(T~min.now):ON(Drop~title(book))OFF:Drop~title(book);
ENDTRANSACTION;
```

Figure 4.16. SF Representation of Transaction 2 in Titles Segment

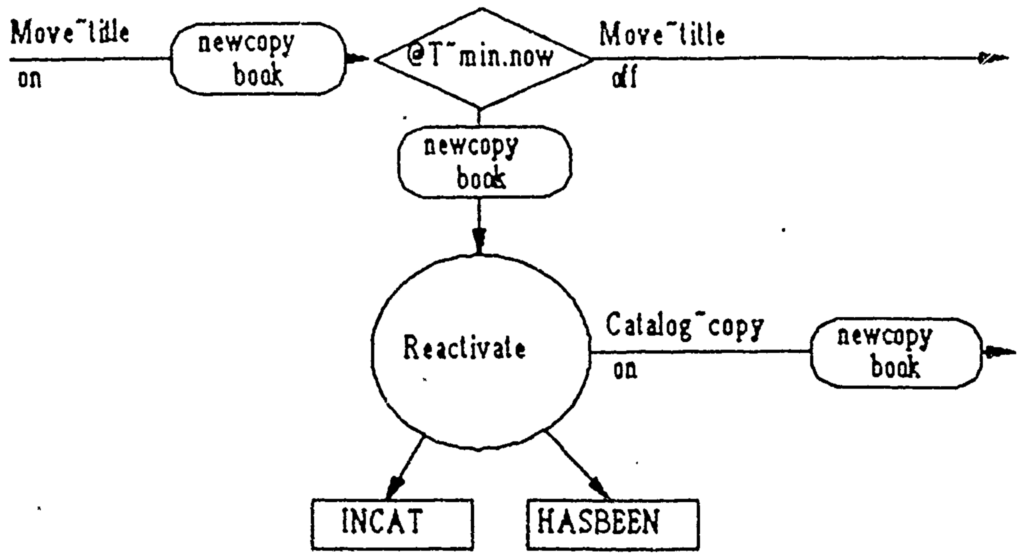


Figure 4.17. Graphical Representation for Transaction 3 in Titles Segment

emerging from the transaction, OFF.

The transaction responds to the Drop˜title signal by activating the Drop title event. The Drop˜title event performs modifications to the INCAT and HASBEEN subsets of the Title type. These modifications are represented by arrows pointing from the event to the types being modified.

The diagram for the final transaction in Titles is constructed in precisely the same manner as the construction of the diagrams for the previous transactions. The graphical and SF specifications for this final transaction are contained in Figure 4.17 and 4.18 respectively.

If the user requests more information about any event, type, transaction, or signal displayed, the SF specification for that construction is displayed. As stated previously, no attempt has been made to completely specify a system using a graphical language.

```
(* Transaction 3 *)
TRANSACTION;
        @(T~min.now):ON(Move~title(newcopy,book))OFF:
                Reactivate(newcopy,book);
ENDTRANSACTION;
```

Figure 4.18. SF Representation for Transaction 3 in Titles Segment

The Copies segment uses the same techniques to construct the diagrams for its one transaction. Copies, however, introduces the concept of external events, events which are not triggered by any transaction. These events are constructed in exactly the same way as other events. For example, Check~copy event inspects the Title type and two subsets of that type, INCAT and HASBEEN. Check~copy also inspects two untyped arguments, newcopy and book. After Check~copy performs its inspections, it sets the signals Catalog~copy, Move~title, and Add~title to ON and passes the untyped parameters newcopy and book with each of the signals. Each of the inspections is represented by an arrow leading from the appropriate type symbol, rectangles for typed objects and rounded rectangle for untyped objects, to the event performing the inspections. Each of the signals is represented by an arrow exiting the event. Each of the arrows is annotated with the signal identifier and the state of the exiting signal. Each arrow in this case also carries a rounded rectangle representing the untyped arguments carried by the signal. The graphical and SF specifications for Check~copy are contained in Figure 4.19 and 4.20 respectively.

The diagrams for the other three independent events and the transaction in the Copies segment are shown in Figures 4.21-4.24. The SF specifications for the events and transaction are contained in Appendix A.

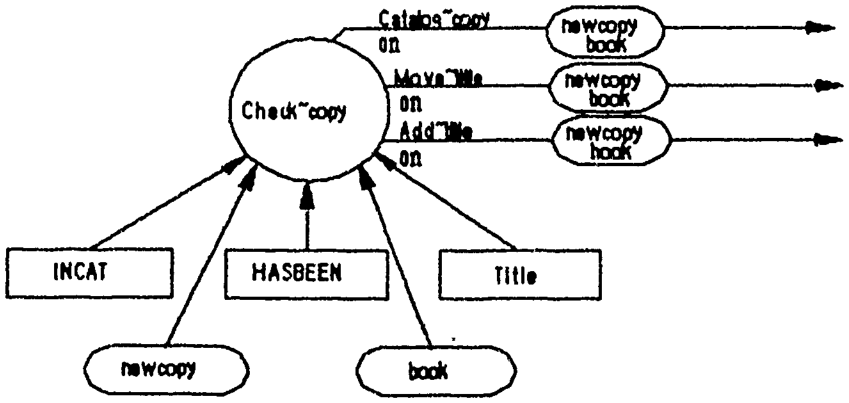


Figure 4.19. Graphical Representation of Check copy Event in Copies Segment

```
EVENT Check~copy(newcopy,book);
        SIGCONDITIONS-  member(book,INCAT) ->
                                (Catalog~copy(newcopy,book))ON;
                        member(book,HASBEEN) ->
                                (Move~title(newcopy,book))ON;
                        not(member(book,T)) ->
                                (Add~title(newcopy,book))ON;
ENDEVENT;
```

Figure 4.20. SF Representation of Check copy Event in Copies Segment

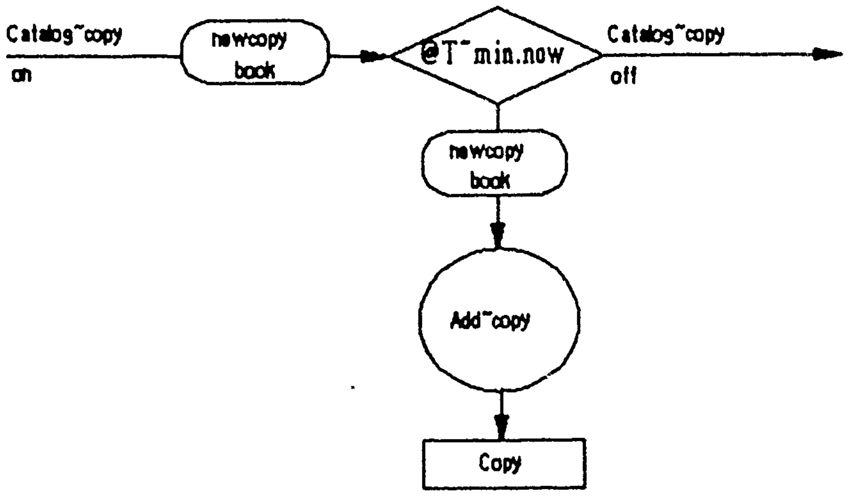


Figure 4.21. Transaction 1 in Copies Segment

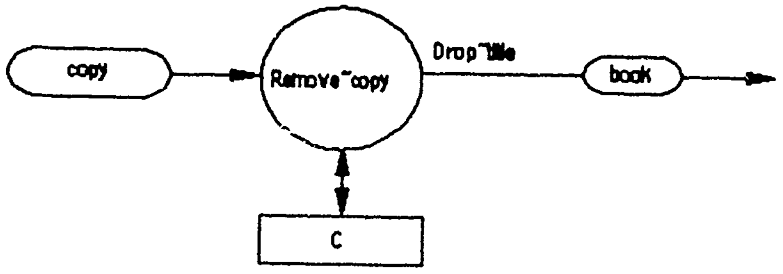


Figure 4.22. Remove copy Event in Copies Segment

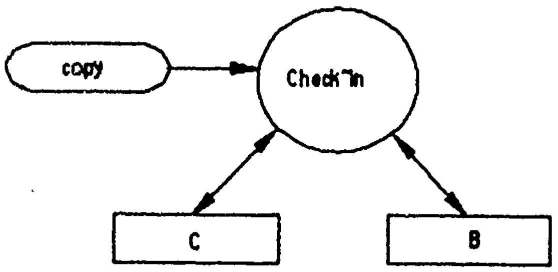


Figure 4.23. Check in Event in Copies Segment

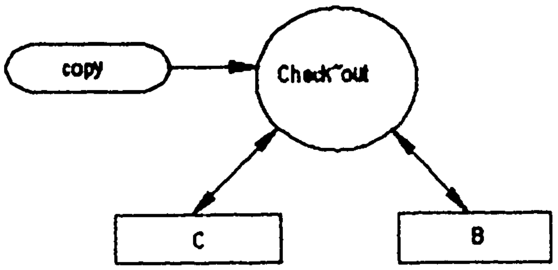


Figure 4.24. Check out Event in Copies Segment

4.5 Elevator Control System Example

The elevator control system problem (11) presents another opportunity to specify a system using this language. The full SF specification for the elevator control system is contained in Appendix B. Although the elevator control system is slightly more complex that the library control system, its graphical specification does not use any concepts not already discussed during the specification of the library system. Therefore, the graphical specifications for the elevator system will not be discussed here, but will be included in Appendix C for the benefit of the reader.

4.6 Conclusions from the Development of the Graphical SF Representation

This chapter has developed a number of techniques to decompose a specification language and to reconstitute the language in a graphical format. These techniques will provide a valuable foundation for the work in the next chapter to create a graphical representation of a far more complex specification language.

This chapter also demonstrated the large increase in specification clarity and understandability when the specification is expressed in a graphical notation rather than in a purely textual format. Although the objectives associated with language utility were not addressed in this chapter, the language clarity objectives of perceptual coding (the adjacent positioning of language symbols to represent sequential execution, restriction to easily understood objects), visual differentiation of specifications, and data abstraction (composition of individual functions into specification segments), were all fulfilled by this relatively simple graphical language. This simple graphical specification language, then, has shown that it is possible to build a graphical language directly on a text-based specification language and still meet the objectives initially established to ensure that the language truly increases specification understandability.

The development of a preliminary graphical specification language, then, proved to be a worthwhile effort because of the techniques that were developed and because the effort demonstrated the possibility of satisfying the language clarity objectives. With the development of this simple graphical specification language complete, then, it is now possible to begin the major task of this research: the development of a graphical specification language based on the Refine^{TM} specification language.

V. A Graphical Representation of the Refine Specification Language

Because of its simplicity, the SF language is ideally suited for use as a learning tool for the formal specification of software as well as a preliminary foundation for a graphical specification language. However, that same simplicity prevents SF from being useful in the specification of nontrivial, real-world systems. The Refine language, a wide spectrum software specification, design, and modeling language developed and marketed by Reasoning Systems, Inc., was developed to specify just such systems (23). Refine is built not only on sets as SF is, but also encompasses additional complex data structures such as sequences and maps as well as objects, a special data structure that will be discussed later in the chapter. These additional data structures make Refine a much more flexible and powerful language, but this power and flexibility come at a price: increased language complexity. Because Refine is more complex than SF, a graphical language based on Refine will be more complex than one based on SF.

In addition to the objectives already established, a primary objective in the development of a graphical language based on Refine is to keep the complexity of the graphical language to a minimum in order to maintain the clarity and utility of the language. The development of a graphical specification language based on Refine proceeds in the same manner as the language development for SF. First, Refine is decomposed into a set of language primitives. Second, a set of graphical symbols is developed to represent those Refine primitives. Finally, the new graphical specification language is demonstrated by using it to specify two example problems: a library control system and an elevator control system.

5.1 Decomposition of Refine

5.1.1 Decomposition Method The parse tree was the primary tool used to decompose the SF language. The parse tree was used because it provided a convenient vehicle for constructing a visual representation of the structure of the language and also provided some visual clues about which language constructs are the primary constructs in the language. The parse tree could also be used for the decomposition of Refine, but for a complex language such as Refine, the explicit construction and evaluation of the parse tree for the entire language would be overwhelming. A better way to approach the decomposition of a complex language such as Refine is to use the implicitly defined parse tree contained in the language documentation as a guide to an efficient language decomposition.

The reader might recall that one of the objectives in the development of this graphical specification language was to provide some mechanism to support data encapsulation, the combination of a data structure and its associated operations into one program structure. In order to provide data encapsulation, all language operations must be categorized by the operands that those operations accept. If all of Refine's primitive data types could be identified then all of Refine's primitive operations could be identified as well. This categorization would not only provide a convenient method of identifying key Refine operations, but would also provide a solid foundation from which to proceed with the implementation of any data encapsulation mechanisms. Because the categorization of Refine operations by their operand data types facilitates the completion of two tasks at the same time, it is the method that will be used for the decomposition of the Refine language. To summarize, then, the decomposition of Refine will be accomplished in two steps: first, all primitive data types in Refine will be identified, and second, all of Refine's primitive operations will be categorized by the data types of their operands.

5.1.2 Data Type Identification The Refine language implements a wide variety of data types ranging from the very simple and obvious to the very sophisticated. This section will identify each of the data types provided by the Refine language and will give a brief description of each data type to provide the reader with a basic familiarity with the Refine language.

The first three data types implemented by Refine are common to almost all programming languages, the number, the character, and the boolean. The number is provided by Refine to accommodate any numeric operations. The number may take the form of either an integer or a real. Refine also allows the user to construct numeric subranges to enhance portability and readability. The character is simply a printable character. The boolean is a boolean value which may have only the values of true and false. Both the character and the boolean may be represented by either a variable or a literal. The literal in the boolean's case is simply either the word "true" or "false". A character literal is somewhat untraditionally represented by the characters "#\" followed by the literal character.

The symbol is provided by Refine to allow the user to manipulate information by symbolic names. This data type is somewhat reminiscent of the use of enumerated data types in Algol based languages. The symbol may be represented by either a variable or a literal symbol, which is represented by the literal symbol preceded by an apostrophe, e.g., 'symbol-name.

The set, sequence, and string are all Refine constructs that allow the manipulation of groups of homogeneous entities. The set is an unordered collection of entities where no two entities in the set are identical. The sequence is an ordered collection of entities where two or more entities in the sequence may be identical. The string is simply a sequence of characters. All three of these data types may be used in either variable or literal form. The literal form for the set is a listing of the entities comprising the literal set separated by commas and enclosed on either end by a set of braces, e.g., {member-1, member-2}. The sequence uses the same representation except that it uses brackets instead of braces, e.g., [member-1, member-2]. The string literal is represented by either the literal string enclosed in double quotes, e.g., "string", or as a sequence of literal characters, e.g. [#\s, #\t, #\r, #\i, #\n, #\g]. Clearly the former method of representation is superior to the latter in terms of clarity and utility.

The tuple is Refine's construction for representing a collection of heterogeneous information. Quite similar to the record in Pascal, Modula-2, and Ada, or the structure in C, the tuple allows the user to manipulate groups of widely diverse information types as a unit, a feature which is extremely important in terms of both language clarity and utility. The tuple can be used in either a variable or literal form. The literal form of the tuple is a less-than sign followed by the literal representations of the tuple's data separated by commas all followed by a greater-than sign, e.g., <"string", 1000, 'symbol, true>.

According the the Refine Users Guide, maps are "partial, unary functions from a domain type D to a range type R" (23:128). The term partial function refers to the fact that a map may be defined for all, some, or even none of the elements in its domain. Maps are extremely useful for compactly representing and for facilitating point-wise definition and manipulation of relationships between entities. As with other Refine data types, maps and manipulation of relationships between entities. As with other Refine^{TM} data types, maps can be used in either a variable or literal form. The literal form of the map is the character set "{|" followed by all literal map assignments separated by commas, followed by the characters "|}", e.g., {| domain-element-1 → range-element-1, domain-element-2 → range-element-2 |}. A map can also be constructed to represent an n-ary relationship by defining the range type as a tuple.

The binary relation is defined by Refine^{TM} to be a set of paired entities. Whereas maps are useful for finding out what entity if any has a relationship with an entity in question, binary relations are useful for finding out whether a relationship exists between two known entities. The binary relation can be used in either a variable or literal form. The binary relation's literal form consists of the set notation enclosing a number of pairs enclosed in less-than and greater signs and separated by commas, e.g., {<domain-1, range-1>, <domain-2, range-2>}.

The final data type implemented by Refine^{TM} is the object. The object is a Refine^{TM} entity capable of storing heterogeneous information, like a tuple, but is also capable of existing outside of the Refine^{TM} construct that it was defined in. In other words, all other Refine^{TM} data types cease to exist once the function that the data type was defined in is terminated. Objects, however, continue to exist even after their defining function has been exited. This feature is extremely useful for creating persistant information structures that must be used throughout a Refine^{TM} program or information structures that must be shared between Refine^{TM} programs. An object is maintained in the Refine^{TM} Knowledge Base (22:185) and must be accessed through functions defined by Refine^{TM}. Because objects are maintained in the Knowledge Base, objects have no literal representation.

In summary, Refine provides a wide variety of data types to facilitate many diverse information representation schemes. As a quick reference, then, the fundamental data types are

- Numbers
- Characters
- Booleans
- Symbols
- Sets
- Sequences
- Strings
- Tuples
- Maps
- Binary Relations
- Objects

5.1.3 Refine Operation Identification and Categorization Now that the Refine data types have been identified, it is now possible to categorize Refine operations according to their operand data type in order to fully decompose the Refine language. This section is divided into subsections by Refine data type. Each section then will identify and briefly describe each operation associated with that particular data type. The Refine User's Guide (23) is the major source of information for this section.

5.1.3.1 Numbers As stated before, the numbers data type in Refine includes integers, real numbers, and integer subranges. Refine provides the four common number functions, addition, subtraction, multiplication, and division, as well as integer division and integer remainder, i.e. the modulo operation. Refine also provides the common relational operators, greater than, greater than or equal to, less than, less than or equal to, and equality. Finally, Refine provides an operation to convert an integer into a real number.

5.1.3.2 Characters The reader should recall that the character data type in Refine consists of the alphanumeric characters as well as special printable system characters such as the backslash, caret, and so on. Refine provides relational operators for the character data type. The relational operators include equality, greater than, greater than or equal to, less than, and less than or equal to.

5.1.3.3 Booleans The Refine language provides the basic operations of first order predicate logic as the primitive operations for the boolean data type. These operations include negation, conjunction, disjunction, implication, universal quantification, existential quantification, and equality. In addition to these basic operations, Refine also provides ordered conjunction and ordered disjunction. The basic conjunction and disjunction operations operate on their operands in a nondeterministic order, i.e., the user has no control over the order in which the operands are evaluated. In contrast, the ordered conjunction and disjunction operations allow the user to specify a precise order in which the operands will be evaluated. Finally, Refine provides a nondeterminstic choice operation which returns an arbitrary element from a set of items satisfying a given set of predicates. The nondeterministic choice operation is undefined if there are no items satisfying the set of predicates.

5.1.3.4 Symbols Since symbols are simple labels used to increase the clarity of the Refine language, they do not require a great number of operations to facilitate necessary symbol operations. Indeed, Refine provides only two operations for symbol manipulation, an operation to test for the equality of symbols and an operation to convert a symbol into a string to facilitate more extensive manipulation of the symbol.

5.1.3.5 Sets The Refine language provides a set of primitive operations associated with the mathematical concept of sets, size (cardinality), element addition, element deletion, union, intersection, set difference, equality, and membership, as well as tests for whether a set is empty and whether one set is a subset of another. In addition to these basic operations, Refine also provides operations that are more conceptually advanced. The filter operation returns the elements of a set, if any, that cause a specified predicate be true. The set reduction operation applies a user specified operation on each element of a set and returns the result, for example the operation reduce(+, {1, 2, 3}) would return the value 6, i.e., $1 + 2 + 3$. (23:88) Finally, Refine provides an operation to transform a set into a sequence.

5.1.3.6 Sequences The reader might recall that a sequence is an ordered collection of items where multiple occurrences of identical items is allowed. The Refine language provides a large variety of operations for the sequence data type consummate with the need not only to manage the elements of a sequence but also the order of those elements. Refine starts by implementing a group of basic operations closely related to several set operations, size (in this case the length of a sequence), equality, and membership as well as a test for emptiness. Noting these similarities in operations between data types will become very important in the development of a graphical language. Refine then provides a number of operations to manage the ordering of a sequence. These operations include returning the n-th element, returning the first element, returning the last element, returning a subsequence of a sequence, returning the subsequence following a specified position, assigning the n-th position of a sequence a value, inserting an element into the n-th position, appending an element to a sequence, prepending an element to a sequence, deleting the n-th element of a sequence, reversing the order of a sequence, and concatenating two sequences. Furthermore, Refine provides filtering and reduction operations for sequences akin to those provided for sets, the filtering operation returns only those elements satisfying a specified predicate and the reduction operation returns the result of applying a specified operation to each element in turn. Refine also provides operations for returning the image of a sequence under a map, returning the domain of a sequence, and returning the range of a sequence. The operation for returning the image of a sequence under a map returns a sequence containing the results of applying a specified map to each element of a specified sequence in turn. If the map for a particular element in the specified sequence is undefined, then that result is not included in the sequence resulting from the image operation. The domain operation returns a set of integers representing the position indices of the sequence that are defined. For example, if the sequence S is defined to be [apple, orange, banana], then domain(S) would return { 1, 2, 3 } since the first, second, and third positions in S are defined. The range operations returns the set of elements contained in a specified sequence. Thus, range(S) would return { apple, orange, banana }. Finally, Refine defines two type coercion operations for the sequence data type, one operation to coerce sequences to sets and the other to coerce sequences to maps.

5.1.3.7 Strings Since strings are defined as sequences of characters, all of the operations defined for the sequence data type also apply to the string data type. In addition, Refine defines a set of string comparison operations which allow the user to alphabetically order strings. These comparison operators are greater than, greater than or equal to, less than, and less than or equal to.

5.1.3.8 Tuples The tuple is Refine's mechanism for maintaining and manipulating groups of heterogenous data. Because the types of the fields contained in any tuple cannot be specified beforehand, it is impossible to define operations to manipulate the contents of the tuple type; the operations on tuple contents are governed by the operations defined on the contents' type. Refine's designers have realized this fact and have provided two operations to facilitate the maintenance of tuple contents but not manipulation of the tuple information. The first operation if the field retrieval operation, which allows the user to either set or get the contents of a particular tuple field. Refine also provides an equality operation which compares two tuples on a field by field basis. The equality operation is only defined if both of the tuples in question have the same number of fields and if corresponding fields in each tuple have the same data type.

5.1.3.9 Maps Maps represent a partial, unary relationship between two data types (23:128). Once a particular map is established, that map is then used by calling it with some member of the map's domain type; after this call, the map will return the range element associated with the domain element, if one exists. If there is no range element associated with that particular domain element, then the map is undefined. Refine first defines a number of very simple operations on the map data type: size (the number of elements in the map's domain) and equality, as well as a test for map emptiness. Refine also defines filter, image, domain, and range operations for the map which closely resemble the operations of the same name defined for the sequence data type. The filter operation for the set and sequence returned a subset or subsequence respectively containing all of the elements of the initial set or sequence satisfying a specified predicate. The filter operation for maps extends this concept so that if a map is defined with a domain type of a set or sequence and a range type of a boolean, the filter may be called with such a map and a set or sequence as arguments; the result of this operation will be a subset or subsequence which contains all of the elements of the initial set or sequence where the map of the element is true. For example, if a map M was declared as {| apple -> true, orange -> false, banana -> true |} and set S was declared as {apple, orange, banana}, filter(M,S) would return {apple, banana} since apple and banana are the only members of S whose range elements in M are true. The image operation takes a map and a set or sequence as arguments and returns a set or sequence containing the results of the application of the map to each element in the set or sequence in turn. The domain operation returns the set of all of the domain elements defined in a specified map. The range operation returns the set of all of the range elements defined in a specified map. The closure operation and composition operation allow the user to perform even more abstract manipulations of the map data type. The closure operation requires a map where the domain and range types are identical. The closure operation will then return the smallest subset of the domain type such that the result of a map evaluation for any element in the set will also be an element of the set. The composition operation allows the user to combine two maps so that the result of a composition evaluation is equal to the evaluation of the first map which is then evaluated by the second map. The composition operator is actually a representation of the function composition operator in advanced algebra. Finally, Refine defines one type coercion operation for the map type, an operation to coerce a map to a binary relation.

5.1.3.10 Binary Relations As stated previously, the map and the binary relation data types are very closely related data types. While the map returns the range element associated with a particular domain element, if one exists, the binary relation simply returns whether a relationship between a particular domain element and range element exists. Because of their close similarity, maps and binary relations share a number of common operations. The image, domain, range, closure, and composition operations are all common to both the map and binary relation. As in the map, the image operation accepts a set or sequence as an argument and returns a set or sequence containing the results of the application of the binary relation to each and every element in the set or sequence. The domain operation returns all of the domain elements defined in a particular binary relation. The range operation likewise returns all of the range elements defined in a particular binary relation. The closure operation returns the smallest subset of elements such that the range element associated with any element in the set is also in the set. The composition operation combines two binary relations to create a new binary relation where the domain element of the new relation is the domain element of the first old relation and the range element of the new relation is the range element of the second old relation. As the reader can see, these operations perform similarly to their counterparts defined for the map data type. Since the binary relation is defined to be a set of ordered pairs, all set operations are also defined for the binary relation data type. In addition to all of these shared operators, Refine defines two unique operators on the binary relation. The first is a transitive closure operator. This operator returns the smallest set of ordered pairs such that the existence of <x, y> and <y, z> in the set implies that <x, z> is also in that set. Finally, Refine provides one type coercion operator for coercing a binary relation into a map.

5.1.3.11 Objects Refine's definition of operations on the object data type is primarily limited to the creation and destruction of objects (since objects are persistant data types) and assignment and retrieval of object attributes. Refine defines other minor object operations for the querying and manipulation of the object environment, but those operations are beyond the scope of this effort. In practice, objects behave quite similarly to tuples except that objects continue to exist beyond the scope of the function in which they were defined. This fact will be used further in the development of the graphical language. In this operation definition stage, however, the object operations which will be dealt with are creation, destruction, attribute assignment, and attribute retrieval.

All major Refine operations have now been identified and associated with operand data types. It is now possible to begin the construction of the graphical formal specification language based on the Refine language. However, before concluding this section, it would be quite beneficial to construct a concise listing of the operation categorization just completed, since this information will be referenced frequently during the development of the graphical language. A listing of the Refine operations, categorized by data type, is contained in Appendix D.

5.2 The Development of a Graphical Representation for Refine

Now that the numerous pieces of the Refine language have been identified and categorized, it is now possible to proceed with the development of a formal graphical language based on Refine.

5.2.1 A Foundation for the Graphical Language Davis and Keller (10) present a graphical language framework which will be extrememly useful for the development of this graphical specification language. Davis and Keller use directed graphs to represent program actions. In their representation, the nodes of the graph represent action performed by the program on data while the arcs of the graph represent the flow of information from one information processing site to another. Davis and Keller make several arguments supporting the use of data flow graphs as graphical languages. First, data flow graphs enhance the understandability of a program by using "a simple data availability firing rule" (10:26). The rule refers to the fact that the information processing function of a particular node occurs only when all of the arguments for that particular node are present. Perhaps the easiest way to visualize this firing rule is to view individual pieces of information as tokens flowing throughout a network of pipes. A node then is somewhat like a small refinery in the midst of this network, accepting input as one form of data and delivering output as another form of data. The data availability rule may then be perceived as the refinery performing its data conversion only when tokens are present at every inlet of the refinery. The refinery then destroys each of those tokens and places the appropriate new tokens at its outlets. The new tokens then continue to make their way through the rest of the network. Although perhaps difficult to grasp at first, this data availability firing rule greatly simplifies the graphic depiction of control flow in a computer program.

Davis and Keller's second argument for the use of data flow graphs is that individual graphs are easily composable into larger, more abstract graphs. As the reader might recall, one of the objectives for this graphical specification language was to provide a mechanism for data abstraction. Because data flow graphs inherently provide this mechanism, they are well suited to be used as a foundation for this graphical language.

Davis and Keller's last argument relevant in this context is the fact that data flow graphs avoid expressing a specific execution order for nodes, but instead depict data dependencies between the nodes. This fact is important because modern computer systems, especially embedded systems, are rapidly moving toward multiprocessing and distributed architectures which require many different portions of a program to run concurrently. Since data flow graphs reflect data dependencies but not execution order, data flow graphs naturally lend themselves to the decomposition of programs into parts that can be concurrently executed and make program synchronization requirements immediately obvious.

These arguments, the data availability firing rule, the ability to compose data flow graphs into larger, more abstract graphs, and the expression of data dependencies but not execution order, show how naturally the concept of the data flow graph fits into the envisioned format of the graphical specification language. Because of these reasons, and some that will become apparent later on, the data flow graph will be the foundation for the graphical specification language.

5.2.2 Graphical Language Development With the foundation for the graphical language firmly established, it is now time to begin formulating the graphical notations that will be used to represent the various Refine language constructs. Davis and Keller's data flow graph formulation uses circles to represent both data and information within the graph, and squares to represent "complex" functions, i.e., functions that have been composed out of lower level primitive operations. Although this notation seems to work for small programs where there is no confusion about which elements are data and which are operations, in larger programs where the individual constructing the program is not necessary the same individual maintaining it, this notation could become very confusing and decrease the understandability of a program. For this graphical language, then, it would seem necessary to develop a notation that would differentiate between data and operations, thereby increasing the clarity of the language. The data flow diagram notation (DFD) represents operations with a circle and represents information stores with a rectangle (25:235). The entity relationship diagram also uses rectangles to represent data (20:7). The choice of a rectangle to represent program data and the circle to represent program operations, then, would be consistent with existing graphical programming notations, thus reducing the training time necessary to learn this new language. For this reason then, the rectangle shall be used to represent program data and the circle shall be used to represent program operations. For reasons that will be apparent later though, it will be necessary to relax the graphical notation for operations to include the use of ellipses in order to accomodate long, meaningful operation names.

The use of the arrow to represent the movement of information from node to node in the data flow graph is a convenient and intuitive notation. Indeed, this notation is consistent with that of DFDs (25:235). Therefore, the arrow notation will be maintained. The data flow graph notation, however, will require one minor enhancement. In real-time system and embedded system design, the designer frequently needs to dictate an execution order on a set of operations regardless of data dependencies. This need usually arises out of system timing considerations. Therefore, in addition to the solid arrow depicting data flow from one operation to another, the dashed arrow shall be used to indicate control flow from one operation to another. The dashed arrow will not denote the flow of information from one operation to another, but will simply indicate that a particular execution order is dictated in that program function.

Having established the basic notations for program data, operations, information flow, and control flow, it is now possible to concentrate on the development of the notations distinguishing individual operations and data types. The individual operation notations shall be handled first. There are two approaches available for the development of operation notation. The first is using simple textual names for the operations and simply enclosing this name in operation notation, i.e., a circle or ellipse. This approach has the advantage of being extremely simple to implement. However, text names can take up a lot of space on a bit mapped screen, an important consideration in the utility of the language since screen space will be extremely limited when the language is implemented on a workstation. The other approach is to implement operation notations using icons, small pictures which graphically symbolize the operation they represent. Icons have several advantages over textual labels: first, they convey information more quickly than their textual counterparts;

second, they are language independent, i.e. a Japanese programmer would be able to understand an icon as well as an American programmer; third, icons take up less space than textual labels (19:208). Because of their high information content and language independence, the primitive graphical language operations will be represented as icons. However, to ease the user's task of creating new functions out of language primitives, the graphical language will support both icons and textual labels for user defined functions.

Now that icons have been selected as the primary representation medium for the graphical language operations, it is possible to begin the design of the language's icons. One way to approach this icon design would be to design a unique icon for each operation in the Refine language. This approach would be cumbersome, tedious, and most importantly, would affect the clarity and utility of the language because of the sheer number of icons that the user would have to learn in order to use the language. Operator overloading, the concept of using the same symbology to represent similar functions, e.g., using "+" to represent the addition of two numbers as well as the addition of an element to a set, offers one possible mechanism for reducing the number of icons required to represent the Refine language. Operator overloading requires a more complex language translator that can replace the overloaded operator based on the number and type of arguments associated with the operator, but this additional complexity is well justified in this case by the increased clarity and utility of the language. The Refine operations have been identified and categorized by data type. In order to implement operator overloading, it will be necessary to recategorize the operations by operator type rather than by operand type. At this point, the reader might reasonably ask the reason for the first categorization of operations if the operations now need to be recategorized. The categorization of operations by operand type provided a convenient and even necessary framework for the initial identification of Refine operations. Also, categorization by operand type will be used later to provide a foundation for data encapsulation.

Instead of attempting a laborious discussion of the fairly obvious similarities between operations, this research will simply present a recategorization of Refine operations based on conceptual similarity. A listing of the recategorized operations is contained in Appendix E.

With the recategorization of the Refine operations, it is now possible to proceed with icon design for the operations. Many of the mathematical operations already have universally accepted symbolic representations. It would be extremely foolish to disregard this standard notation in favor of some new symbolism. Therefore for the operations of addition, subtraction, multiplication, division, ıality, greater than, greater than or equal to, less than, and less than or equal to, standard mathematical notations will be used. The resulting icons are contained in Figure 5.1.

As the reader will see later in this research, the simple assignment operation is one of the most widely used operations in this language. The symbology for this important operation should be distinctive, yet simple, to facilitate quick association between the icon and the assignment operation by the user. Since the rectangle has been adopted as the standard symbology for program data, a rectangle, or the more compact square, should also

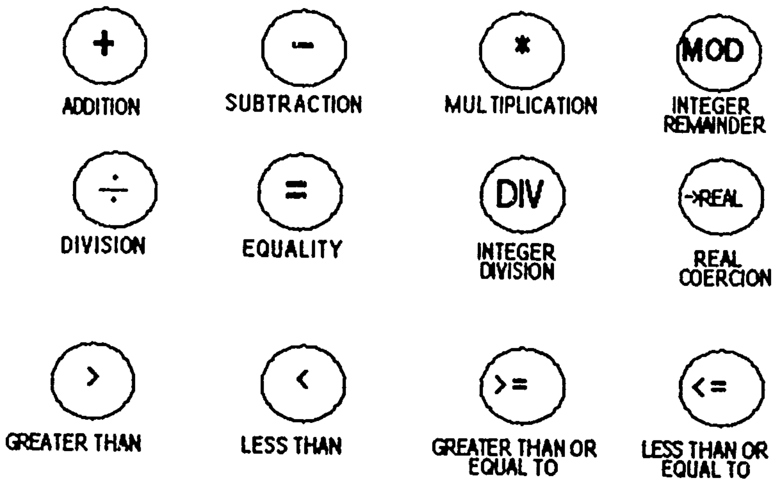


Figure 5.1. Graphical Representations for Mathematical Operations

be used in this representation. In a convention that will be followed throughout the rest of the icon development, a darkened square will be used to denote the portion of the data, such as a field of a tuple of an element of a segment or sequence, that is being modified or inspected. This feature is not critical in this context because the entire data structure is of interest, but in later icon developments for more complex data structures, this feature will be important for providing clues to the user about which portion of a data structure is being affected by a particular operation. Finally, an arrow pointing to the darkened square represents the placement of data into the data structure, a representation which uses Rohr's findings that action oriented icons more effectively convey their meanings than their non-action oriented counterparts (23:327). The icon representing the simple assignment operation is contained in Figure 5.2.

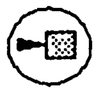


Figure 5.2. Graphical Representation of the Simple Assignment Operation

Boolean operators also have an accepted standard symbolic representation. Unlike the standard mathematical notation however, standard boolean symbology is not widely used outside of the academic environment. Since one of the goals of this graphical specification language is to construct a symbolic representation that is accessible to both programmers and non-programmers, using an infrequently used, albeit standard, notation would not serve to enhance the clarity of this graphical language. Indeed, programmers and nonprogrammers are accustomed to referring to the concepts of negation, conjunction, disjunction, etc., by their simpler, popular names, i.e. not, and, or, etc. These popular names are also short enough that they would not take up a large amount of space. Therefore, in order to take advantage of the popular naming conventions for these boolean operations, the popular names themselves, "NOT", "AND", and "OR", shall be used in icons representing the negation, conjunction, and disjunction operations, similar to the usage of the word "STOP" in the common stop sign. However, the three boolean operations having standard notations without the benefit of short, popular names, implication, universal quantification, and existential quantification, will be represented by their standard symbology, implication by $\Rightarrow$, universal quantification by $\forall$, and existential quantification by $\exists$.

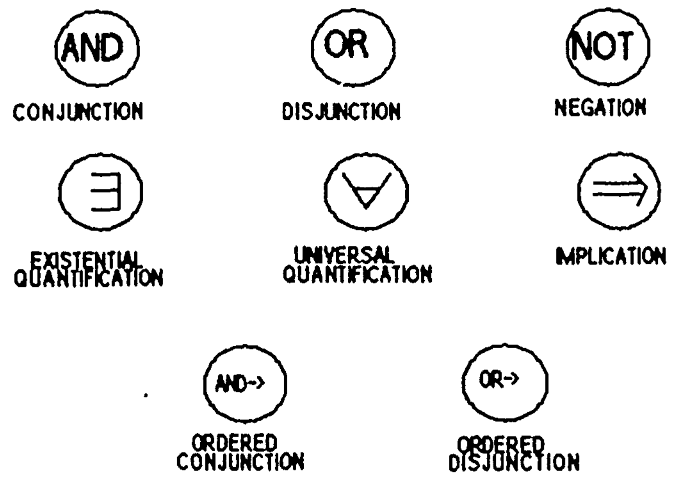


Figure 5.3. Graphical Representations for the Boolean Operations

The two remaining boolean operators, ordered conjunction and ordered disjunction have neither a popular name nor a standard symbology. Because of the close similarity ordered conjunction and ordered disujunction have with the common conjunction and disjunction operators, it would be quite beneficial to construct icons for the ordered operators resembling their unordered couterparts. Ordered conjunction and ordered disjunction, then, will be represented by the words "AND→" and "OR→" respectively. This notation is distinct enough to prevent confusion between the ordered and unordered operations, yet similar enough to symbolize the close similarity between the two types of operations. A summary of the icons representing the boolean operations is contained in Figure 5.3.

Like the numeric operations, several of the set operations also have a standard symbology. Adhering to the principle of using existing, popular notation when practical, the union and intersection operations as well as the subset and membership tests will be represented by their standard symbology, i.e., $\cup$, $\cap$ $\subset$, and $\in$ respectively. The element addition and deletion operations, as well as set difference operation, are grouped with their numeric counterparts, i.e. addition for element addition and subtraction for element deletion and set difference, because of their conceptual similarity to the numeric operations. The arbitrary element operation will be represented by a question mark, the common symbol for the unknown. The size operation will be represented by a ruler, a common measuring device. The empty test will be represented by ϕ?, thus using the standard empty set notation with a question mark to symbolize the query about the emptiness of a set. The reduction operation shall be simply represented by passing a set variable as an argument to the operation that the set will be reduced by. Since the overloaded operators for element addition and deletion require two arguments to complete the expression, there is no problem with ambiguity in selecting the representation. The set to sequence coercion operation is best represented by a graphical representation of the process of transformation. Therefore, this coercion will be represented by the symbol "$\rightarrow$ SEQ" which represents the movement of the data object to a sequence, represented by the abbreviation "SEQ". Finally, the filter operation is the most conceptually abstract concept yet handled which complicates the task of icon design. However, if one were to visualize the actual purpose of a filter, to create a smaller collection of items out of a larger collection, the icon design process becomes relatively simple. The icon representing the filter operations, then is a large box, symbolizing the large collection, above a large "V" structure, representing the filter, above a small box, symbolizing the smaller collections. This icon, then, graphically depicts the actual filter process making the icon more meaningful and intuitive for the programmer

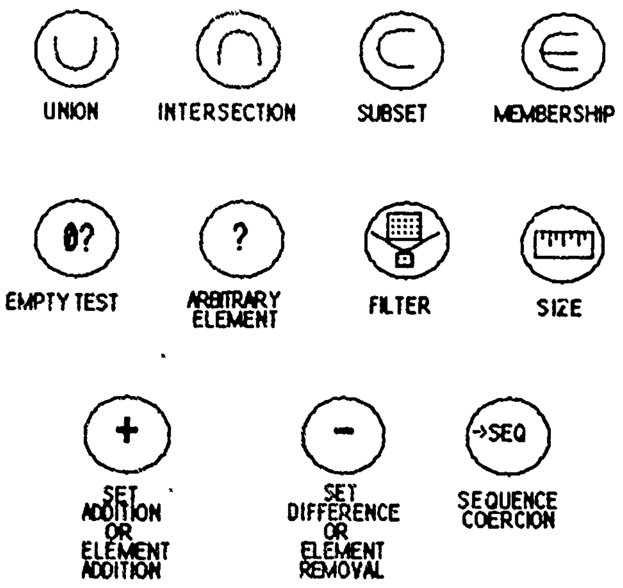


Figure 5.4. Graphical Representations for Set Operations

and nonprogrammer alike. A summary of the set operation icons is contained in Figure 5.4.

The literature search for this research did not reveal any standard notation for the fundamental sequence operations. Therefore, unique representations were constructed for each of these operations. Rohr points out the importance of using icons depicting action rather than using simple picture icons (23). Taking this into mind, then, it is now time to develop icons for the sequence operations. Rather than attempt a tedious explanation of each icon individually, this paper shall only explain key features of the icons as a group. Since the representation for data in this graphical language is the rectangle, that symbol is used throughout the sequence icons also to represent a single piece of data. An elongated rectangle represents a sequence of data. The darkened portion of each icon points out

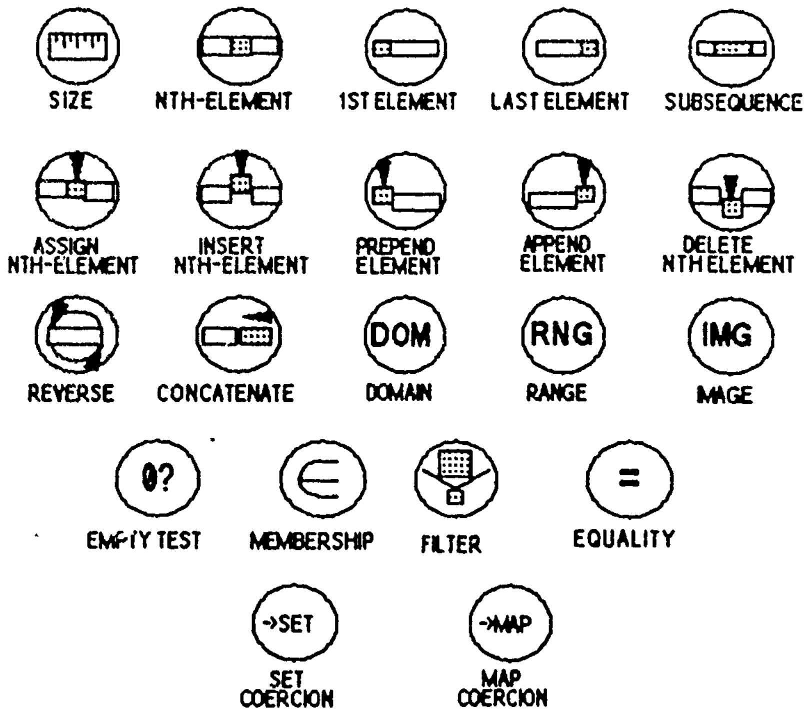


Figure 5.5. Graphical Representations of Sequence Operations

the portion of the sequence that is being manipulated by the operation. In each icon, an arrow is used to provide the user additional cues about the direction that the operation is taking. For example, in the icon representing the insertion of an element into a sequence, the icon uses an arrow pointing to a slightly raised element to symbolize the "pushing" of the element into the sequence. Likewise with the icon for the element deletion, an arrow points to a slightly lowered element symbolizing the "pushing" of the element out of the sequence. With these clues in mind, the reader can find the sequence operation icons in Figure 5.5

Five of the icons in the sequence operation group bear special mention. First, the sequence to set and sequence to map coercion operators follow the "standard" that was established in the development of the set to sequence coercion operation, i.e., the right arrow pointing to the target data type symbolizing the movement of the operand from its original data type to its final data type. The other three sequence operators, the image, domain, and range operators, differ significantly from the other sequence operators in the abstractness of the operations that they represent. Whereas it is fairly easy to depict the process of inserting an element into a sequence, it is not nearly as easy to depict the operation of returning the range of a sequence. It is also important to realize that novice users of this language are unlikely to begin by using such advanced concepts while advanced programmers using these concepts need a way of quickly recognizing these operations. Rather than attempting to construct some abstract symbology to represent these operations, perhaps a textual symbology would better enhance the clarity of the language in this case. The three letter symbols IMG, DOM, and RNG would not take up much space and, when working in the context of these operations, these abbreviations would provide the user with enough information to allow him or her to quickly associate the abbreviations with the appropriate operations. In fact, although no experiments have been performed to verify this hypothesis, it seems logical that the abbreviations would allow the user to make the association between the abbreviation and the operation more quickly than would be possible with some abstract iconic representation. Therefore the abbreviations IMG, DOM, and RNG will represent the image, domain, and range functions.

The icons for the field assignment and retrieval operations for the tuple and object data types are relatively simple to construct. As the reader might recall, the symbology for simple assignment was the arrow pointing to a darkened square. Since the field assignment affects only a portion of a data structure, the logical symbology for the field assignment would be an arrow pointing to a darkened portion of a square which is exactly the symbol-

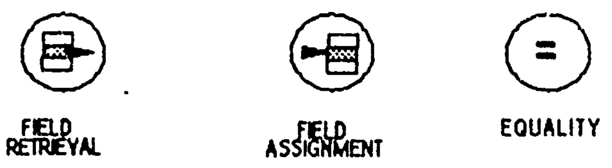


Figure 5.6. Graphical Representations for Tuple Operations

ogy that has been chosen. The field retrieval operation is handled similarly; the operation is represented by an arrow leading away from a darkened portion of a square. These iconic representations are contained in Figure 5.6.

The operations closure, composition, inverse, and transitive closure, present the same problems that the image, domain, and range operations presented, i.e. the iconic representation of highly abstract mathematical operations. The composition and inverse operations already have standard notations, $f \circ g$ and f^{-1} respectively. The closure operation returns the smallest set such that a specified operation performed on a member of the set results in another member of the set. This operation is represented by using a circle, the standard notation for a set, with an arrow, representing an operation, that originates and terminates inside the circle. Transitive closure is a similar operation. If a relation holds for two members of a set, say x and y, and if the same relation holds for the consequent of the first relation and another member of the set, say y and z, then a property called transitivity holds if and only if a relation exists between the antecedent of the first relation, x, and the consequent of the second relation, z. The transitive closure operation returns the smallest set for which the transitivity property holds for all of the members of that set. By coincidence, a cloverleaf pattern describes just such a relationship, i.e. if any two relationships exist between three elements in order, then a relationship also exists between the first and third elements. A circle imposed over a cloverleaf could then represent the membership of all of these points in one set. These then will be the admittedly abstract icons adopted to represent these operations. The map to relation and relation to map coercion operation icons follow the previously adopted patterns. All of these icons may be found in Figure 5.7.

The final icons to be developed for this language are icons representing the creation and destruction of objects. The development of a unique icon representing the creation of objects would be in line with the procedure that has been followed up to this point. Such an icon would require that the programmer include an additional icon in a network to show that he or she wants to create an object when the presence of an object data type implies this very fact. Thus the development of a unique object creation would not enhance the utility of this graphical language but would reduce programmer productivity, thereby decreasing the utility of the language. Common sense dictates, then, that no object creation icon be developed; instead, for each unique object representation encountered in the data flow graph the object creation function will be automatically called. However, an object's persistant nature does necessitate the development of a unique object destruction icon.

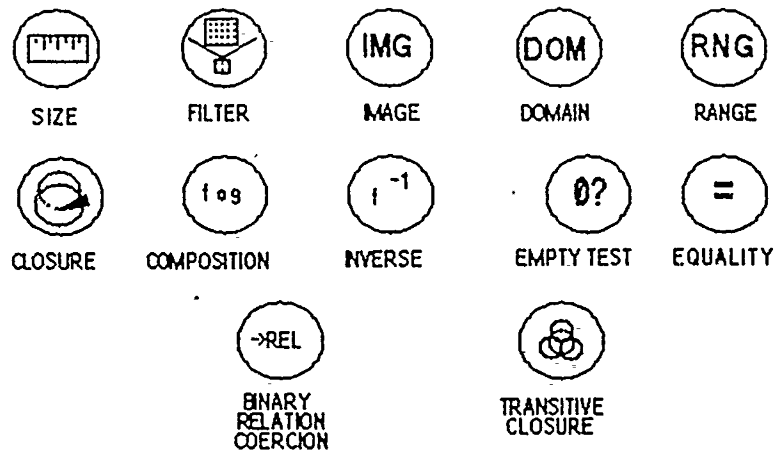


Figure 5.7. Graphical Representations of Map and Binary Relation Operations

Keeping with the rule that simpler is better, the object destruction icon shall use a large 'X' imposed on a double rectangle, the representation for an object whose development will be discussed later. This symbology is contained in Figure 5.8.

The translation of the Refine language into a graphical language requires some additional icons to represent Refine features that are execution control constructions rather than explicit data operations. The first two additional icons are the start and finish icons. Although a language translator could find the starting point of any network by following the arcs in the network backward until it can go no further, explicit start and stop icons improve the clarity of the language by helping the user to find the starting and stopping points of a network without requiring him or her to perform extensive network backtracking. The most common starting and stopping symbols in society today are the red and

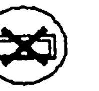


Figure 5.8. Graphical Representations of the Object Destruction Operation

green lights of the common traffic light. Because of the familiarity of these symbols, then, they will represent the starting and stopping points for any function in the graphical specification language. One additional icon necessary for the graphical language is an icon to represent enumeration over a collection of homogeneous data. The enumeration operator allows the execution of a series of operations over each element in a group of data. Enumeration will be one of the most commonly used operations in the graphical language, thereby deserving a unique icon to facilitate easy access to the operation. With no common symbology available to represent enumeration, it is necessary to develop a symbology. The symbology to be used will consist of three adjacent blocks, representing a collection of data, and two arrows, one originating from the first block and terminating on the second block and another originating from the second block and terminating on the third block, symbolizing the processing of each element in the collection. Although not particularly intuitive, this symbology is simple enough that it should be relatively easy for a user to

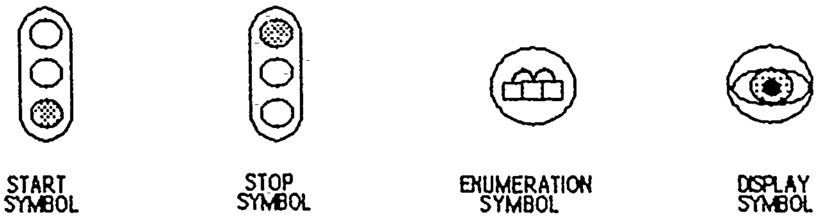


Figure 5.9. Graphical Representations of Start, Stop, Enumeration, and Print Operators

construct an association between this representation and the enumeration operation. The final additional icon is an icon to represent the display of information for the user. Refine does not provide an explicit operation for the display of program information since it relies on its underlying Lisp foundation to provide that functionality. This graphical language, however, must explicitly provide such an operation. Since vision is the primary method that humans use to investigate their world, an icon based on sight would probably allow users to make the most rapid association between the print operation and an icon. For this reason, as well as for its singular distinctiveness, an eyeball will be used to represent the display operation. These four icons are portrayed in Figure 5.9.

Now that the operator icon development is complete, the data type representation development can proceed. It would be possible to develop icons for each data type that would represent all instances of that particular data type. This approach is unreasonable however because programs commonly use many different instances of the same data type and using the same representation for each instance would give the user no way of differentiating between the various instances. In this case, language clarity and utility clearly dictate that textual labels be assigned to each data structure to aid in structure identification. Therefore, data in this graphical language shall be represented by a rectangle containing a meaningful name denoting a particular instantance of a data type. As the reader will see later, when constructing large networks, connecting one data icon to a number of operator icons can result in a complicated and largely unreadable network. By facilitating the assignment of unique names to instances of a data type, this language can allow multiple representations of a uniquely named data type to represent the same piece of data thus allowing the programmer to duplicate data in various places in the network thereby increasing the clarity of the language. A requirement to name each and every piece of data required to construct a program, however, can turn into a tedious and unnecessary task depending on the purpose of the data. In order to give the user the option of deciding when to use unique naming of program data, this graphical language shall also support the concept of anonymous data, i.e. an instance of a data type without a unique name. Unlike uniquely named data, each duplication of a piece of anonymous data will represent a new instance of that particular data type. Anonymous data will also be particularly useful when used with the enumeration operator. Such a conjunction would simply apply an enumeration over all of the elements of that particular data type. It is presumed that the language translator would be sophisticated enough to recognize and restructure enumerations over an infinite set, such as integers or reals. It would be prudent at this point, however, to caution that this feature could be abused and actually decrease the clarity of the language. However, the user is the expert in any situation and therefore he or she is

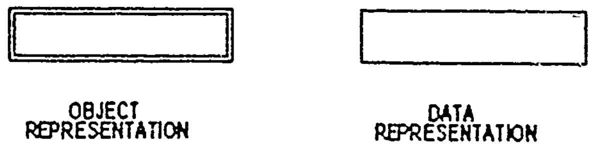


Figure 5.10. Graphical Representation of Data Types

entrusted with this capability in the hopes that it will greatly improve the utility of the language.

Because of their unique persistant character, objects deserve special consideration as a unique data type. A distinctive representation would alert the user to the persistant nature of objects, a characteristic that can cause very subtle errors in program execution if not fully understood. Objects are still a valid data type though, and should therefore have a similar representation. Objects will therefore be represented by a two nested rectangles. Objects, like the other data types, may also be anonymous or uniquely named. The representations for the data types in the graphical language are contained in Figure 5.10.

Now that representations for all operations, data types, and information flows have been developed, it is now possible to begin putting these pieces together to investigate the syntax of this graphical language.

5.3 Graphical Language Syntax

In traditional programming languages, language syntax for a particular expression is described in terms of precise sequences of keywords and user defined names. The syntax of a graphical language, on the other hand, is described in terms of how various nodes are connected in a graph. Since all of the operations in the Refine language require some data to operate on, there will be no operator node in the graphical language that is not connected to some data node. In addition, an operator node may also be connected to another operator node with a control flow, the dataless control signal construct that was discussed. In Refine, all functions have a definite beginning and ending point; likewise the graphical language should also have a definite beginning and ending point, although this feature is more for the benefit of language clarity than processibility. Based on these simple foundations then, the basic syntax, in Backus-Naur Form notation, for a statement in this graphical language is

$$< statement >::= [< data - flow >]\{< data - flow > \mid < control - flow >\}$$
$$< op - node > [< data - flow >]\{< data - flow > \mid < control - flow >\}$$

The formula above compactly states that an operator node must have at least one data flow input into the operator with any number of additional data and control flow inputs, and that an operator must have at least one data flow output and any number of additional data and control flow outputs. The reader is reminded that this formula is only a general statement of the graphical language syntax and that the syntax of particular operators is governed by the operator itself.

Individual operator syntax is governed by the number, order, and type of arguments as defined in the Refine User's Guide (23). This strict compliance with Refine language definitions and conventions preserves a one-to-one and onto translation scheme between the graphical language expressions and the corresponding Refine expressions. Other than compensating for the overloading of graphical operator representations, the graphical language maintains a one for one correspondence between its operators and Refine operators which makes conversion between the two language a straight forward construct replacement process. The translation process between the graphical language and the Refine language will be greatly clarified in the upcoming graphical language application examples.

5.4 Examples Using the Graphical Specification Language

Up to this point, all of the discussion of this graphical language has been in relatively abstract terms. Although abstract explanations about language use suffice for acquainting the reader with basic language concepts, concrete examples are far superior in demonstrating the utility of the language features developed. Therefore, the clarity and utility of this graphical specification language is demonstrated by using the language to specify two problems used during the Fourth International Workshop on Software Specification and Design (11, 18), a library control program and an elevator control program. Selected portions of the graphical specifications foı each of these problems are presented here; complete graphical specifications are contained in Appendix G for the library control system and in Appendix I for the elevator control system. The graphical specifications presented here are accompanied by their corresponding Refine specifications in order to allow the user to compare the two specification formats. Complete Refine specifications for the library control system and the elevator control system are contained in Appendices F and H respectively.

5.4.1 Library Control System Example The first problem concerns a library database control program (18). This program allows the library to perform the usual library functions:

- Add a book to the library stacks
- Remove a book from the library stacks
- Check out a book
- Return a book
- Produce a listing of the books on a particular subject
- Produce a listing of the books by a particular author
- Produce a listing the books currently checked out by a patron
- List the patron who last checked out a particular book

The graphical specifications, along with the corresponding Refine specifications, for adding and removing books from the library stacks, and checking out and returning books are presented in this section. Complete graphical and Refine specifications for this problem are contained in Appendices G and F respectively.

5.4.1.1 Add a Book to the Library Stacks The graphical and Refine specifications of the Add-Book function are contained in Figures 5.11 and 5.12 respectively. The execution of the Add-Book function begins at the start symbol, i.e. the green light, which passes a control signal to the enumeration icon. The enumeration operator takes the anonymous book icon as an argument which results in an enumeration over all of the books currently defined in the environment. The scope of the enumeration operator is defined by the box drawn around the four operators near the top of the page. During the enumeration, each book is passed to the field retrieval operator which retrieves the "Title-of-Book" field from the current book. The title of the book is then checked against the function argument "Title". "Title" is distinguished as an incoming argument by the right pointing arrow adjacent to the data icon which represents data coming into the function. The result of the comparison between the title of the book and the "Title" argument is then passed to a true/false selector. If the title of the book and "Title" are identical then that book is added to the variable "Bookset". After the enumeration is complete, "Book-set" is tested for emptiness. If the set is not empty, the function terminates immediately. If the set is empty, then the operations enclosed by the block near the bottom of the figure are executed. First, a book is created (by virtue of the "Newbook" icon being used in the figure") and then "Title-of-book", "Author-of-book", "Subject-of-book", "On-shelf", and "Checked-out" fields of "Newbook" are set to their respective values. After the "Checked-out" field assignment is complete, "Newbook" is passed to the port on th p icon, which terminates the function and passes the object out of the function.

5.4.1.2 Remove a Book from the Library Stacks The graphical and Refine specifications of the Remove-Book function are contained in Figures 5.13 and 5.14 respectively. The Remove-Book function starts with an enumeration over the books currently defined in the environment. The "Title-of-book" field is retrieved from the currently enu-

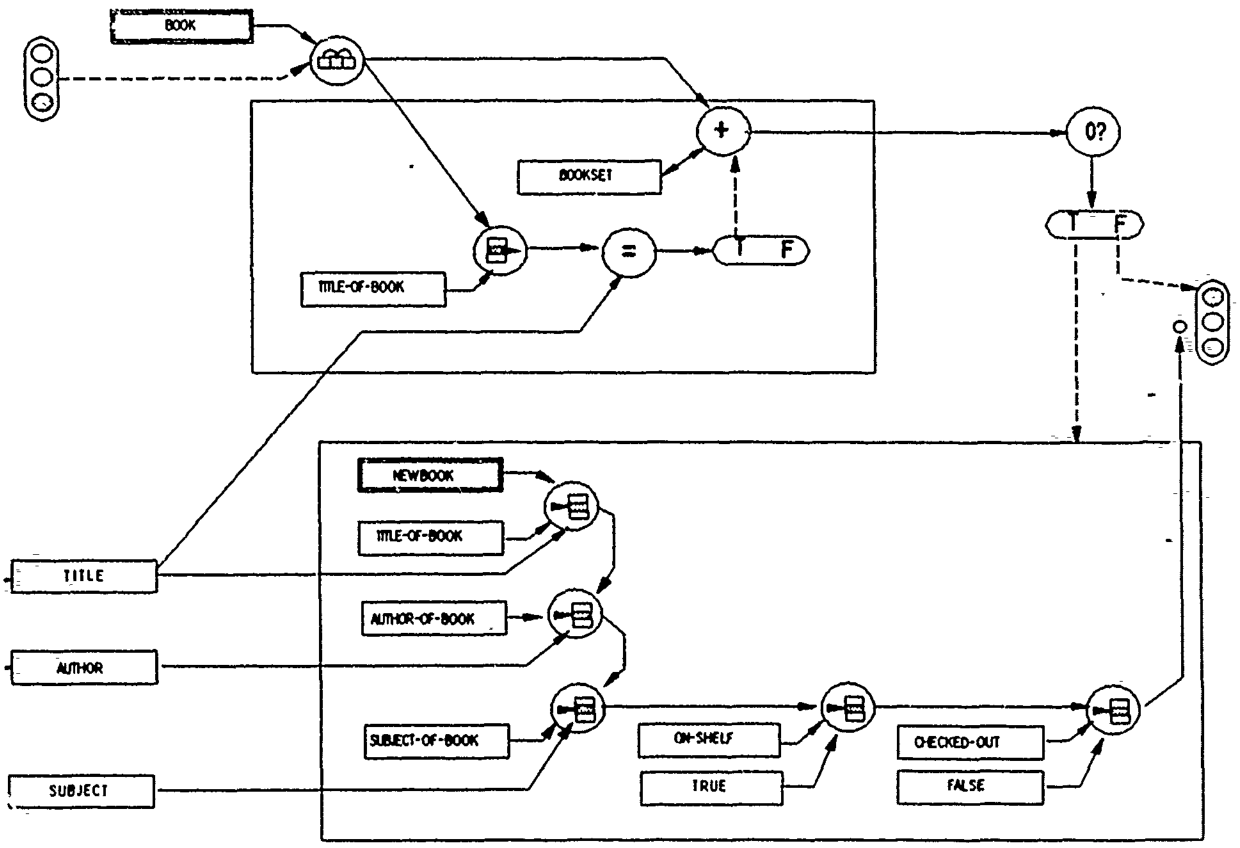


Figure 5.11. Graphical Specification of the Add-Book Function

```
% Library maintenance operations
rule Add-Book-To-Library( author : string,
                          title : string,
                          subject : set( string ) )
    % If there's not already a book with this title
    % (to guarantee unique titles)
    empty({ b | (b: book) book(b) & title-of-book(b) = title }) -->
            % Create a new book object
            let ( new-book: book = make-object( 'book ) )
              % And set the attributes of the book
                author-of-book(new-book) <- author;
                title-of-book(new-book) <- title;
                subject-of-book(new-book) <- subject;
                on-shelf(new-book) <- true;
                book-out(new-book) <- false
```

Figure 5.12. Refine Specification of the Add-Book Function

merated book. This field is then compared with the function argument "Book-title'. If the field is equal to "Book-title" then the book is added to the variable "Bookset". An arbitrary element of "Bookset" is then retrieved and that element, a book, is destroyed. After the book destruction, the function terminates. The reader should notice that the "Bookset" variable is guaranteed to either be empty or to have one element because of the checks performed during the Add-Book operation that ensure that every book in the library has a unique title. Therefore the arbitrary element operation simply returns the only element in the set. The reader might also notice that this function assumes that a user inputs a book title that exists in the library.

5.4.1.3 Check Out a Book The graphical and Refine specifications of the Check-Out-Book function can be found in The Check-Out-Book function starts by performing three enumerations, two over the set of users currently defined in the environment

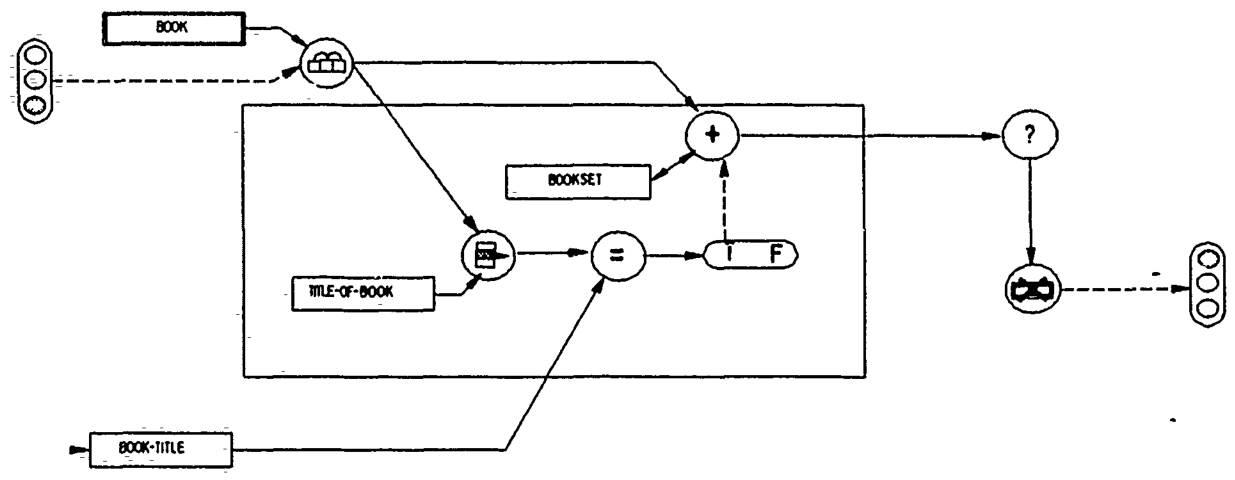


Figure 5.13. Graphical Specification of the Remove-Book Function

```
% Assumes that every book has a unique title
rule Remove-Book-From-Library( book-title : string )
   % If a book of this title exists
   ~empty({ b | (b: book) book(b) & title-of-book(b) = book-title }) -->
    % Find the book with this title
    let ( book-to-delete: book = arb({ b | (b:book) book(b) &
                                            title-of-book(b) = book-title }))
      % And delete it
      erase-kb-object( book-to-delete )
```

Figure 5.14. Refine Specification of the Remove-Book Function

and one over the set of books currently defined in the environment. The first two enumerations call the User-With-Name function, which will be described later, with arguments of the currently enumerated user and the variables "User-set-1" and "User-set-2" and the incoming arguments "Whos-asking" and "Users-name" respectively. Since no bounding box exists around the enumeration, the enumeration's scope is restricted to the single following operation by default. The enumeration over the books repeatedly calls the "Books-with-title" function, which will also be described later, with the currently enumerated book, the variable "Book-set", and the incoming argument "Which-book" as arguments. The results of each enumeration, a set of users for the first two enumerations and a set of books for the final enumeration, are then tested for emptiness. The test results are each negated then all of the results are ANDed together. In plain language, the top portion of this function is simply checking to ensure that users with names of "Who-asking" and "Users-name" and a book with the title "Which-book" actually exist. If users with the proper names and a book with the proper title exist, then the block of operations at the bottom of the figure are executed; if those users and the book do not exist then the function is immediately terminated. If the function is not terminated, then an arbitrary element of the

"User-set-1" and "Bookset" variables is retrieved (the user might recall that because of the checks to ensure unique names for all users and unique titles for all books that were performed in the Add-Book and Add-User functions, "User-set-1" and "Bookset" are each guaranteed to have no more than one element). The "Staff" field of the user is retrieved and the "On-shelf" field of the book is retrieved. These two values are ANDed together and if the result is true then the inner block is executed; if the result is false then the function is terminated. In the inner block, the "Book-set" variable is again reused and an arbitrary element is returned from this set. At this point, it would be wise to remind the reader that because of the unique naming guarantees of both users and objects in the system, each set of users and books is guaranteed to have only one element; the arbitrary element operation is simply a tool for retrieving this single element from the set. The "On-shelf", "Book-out", and "Last-checked-out" fields of the book are then set to their respective values. At this point, both blocks are exited and the function is terminated.

5.4.1.4. Return a Book The graphical and Refine specifications for the Return-Book function can be found in Figures 5.17 and 5.18 respectively. Return-Book is very similar to Check-Out-Book; enumerations are used to fi all of the users having a name matching the incoming argument "Whos-asking" and all of the books matching the incoming argument "Which-book". The reader is reminded that because of the guarantees for unique names in the book and user creation functions, there will not be more than one book with a particular title or more then one user with a particular name. If a user with the proper name and a book with the proper title exist then the outer block of operations is executed; if not, the function is terminated. In the outer block, the user is checked to

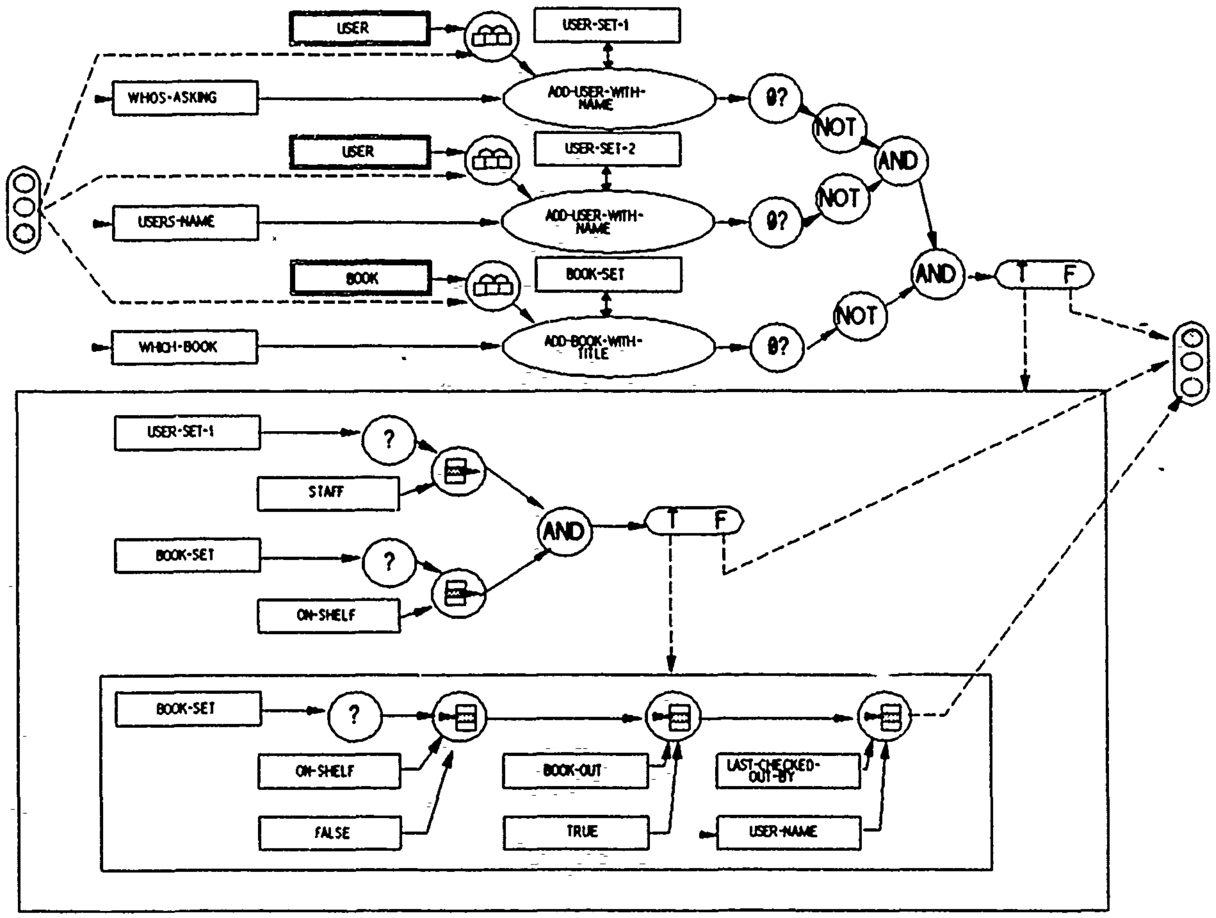


Figure 5.15. Graphical Specification of the Check-Out-Book Function

```
rule Check-Out-Book( whos-asking: string,
                     users-name: string,
                     which-book: string )
   % If there is a user with the name of whos-asking
   ~empty({ u | (u: user) user(u) & user-name(u) = whos-asking }) &
   % And if there is a user with the name of users-name
   ~empty({ u | (u: user) user(u) & user-name(u) = users-name }) &
   % And if there is a book of the title which-book
   ~empty({ b | (b: book) book(b) & title-of-book(b) = which-book }) -->
      % Then if whos-asking is on the staff
      staff( arb({u | (u: user) user(u) & user-name(u) = whos-asking}) ) &
      % And if the requested book is on the shelf
      on-shelf( arb({ b | (b: book) book(b) &
                              title-of-book(b) = which-book })) -->
      % Then find the book with the title of which book
      let( book-to-check-out: book = arb({ b | (b: book) book(b) &
                                      title-of-book(b) = which-book}))
        % And check the book out
        set-attrs( book-to-check-out, 'on-shelf, false,
                                      'book-out, true,
                                      'last-checked-out-by, users-name )
```

Figure 5.16. Refine Specification of the Check-Out-Book Function

ensure staff membership and the book is checked to ensure that it is currently checked out. If both of these conditions hold then the inner block is executed; if not then the function is terminated. The inner block simply sets the "Oh-shelf" and "Book-out" fields of the book to the proper values and then terminates the function. The reader should recall that because objects exist until explicitly destroyed, there is no need to pass the modified object out of the function.

5.5 Graphical Specification of the Elevator Control System

The second problem that will be used to demonstrate the use of the graphical specification language is an elevator control system. As stated in (11), the requirements for the elevator control system are:

1- Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the lift.

2- Each floor has two buttons (except ground and top floor), one to request an up lift and one to request down-lift. These buttons illuminate when pressed. The illumination is cancelled when a lift visits the floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled. The algorithm to decide which to service should minimize the waiting time for both requests.

3- When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests.

4- All requests for lifts from floors must be serviced eventually, with all floors given equal priority.

5- All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.

6- Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status.

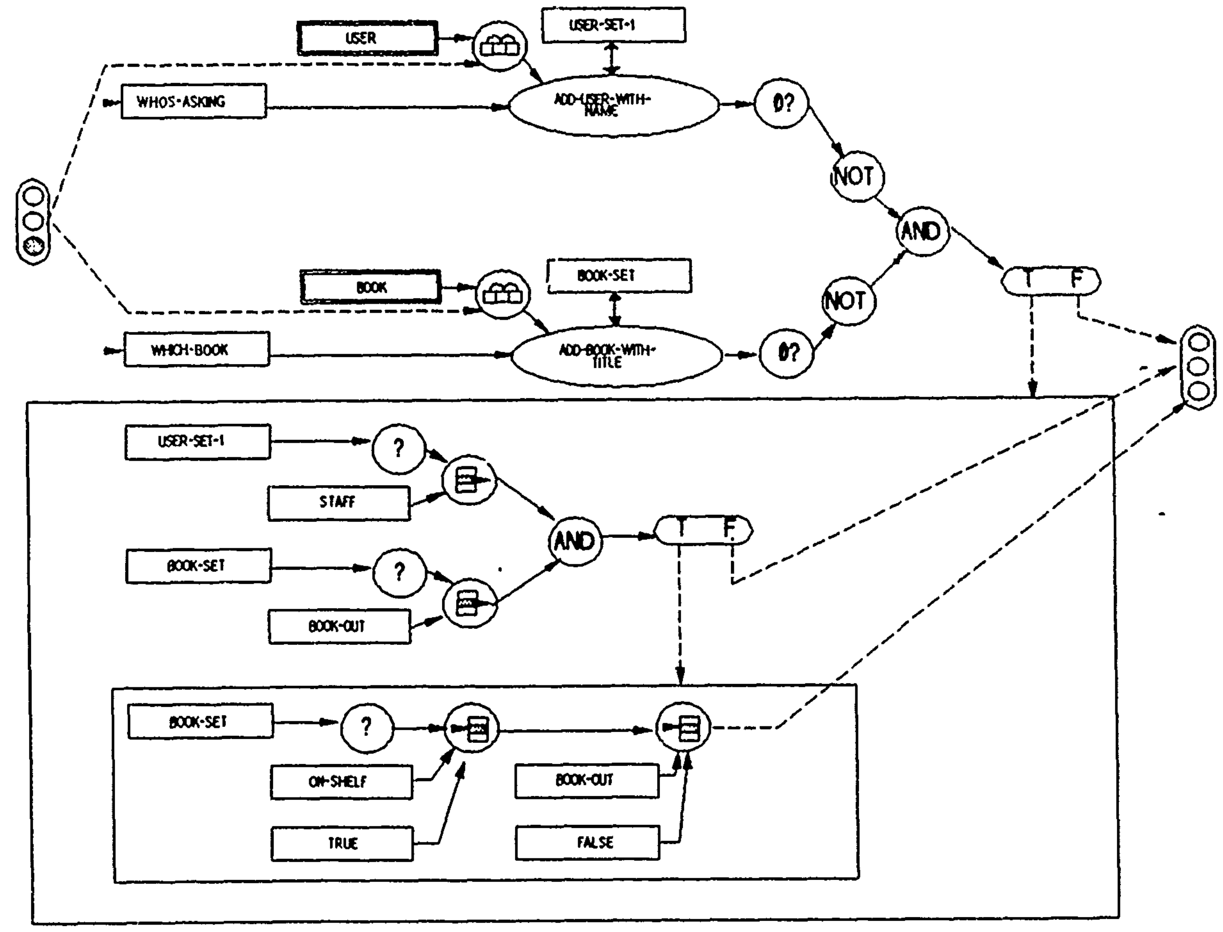


Figure 5.17. Graphical Specification of the Return-Book Function

```
rule Return-Book( whos-asking: string, which-book: string )
   % If there is a user with the name of whos-asking
   ~empty({ u | (u: user) user(u) & user-name(u) = whos-asking }) &
   % And if there is a book with the title of which-book
   ~empty({ b | (b: book) book(b) & title-of-book(b) = which-book }) -->
      % Then if whos-asking is on the staff
      staff( arb({ u | (u: user) user(u) & user-name(u) = whos-asking })) &
      % And if the book is checked out
      book-out( arb({ b | (b: book) book(b) &
                          title-of-book(b) = which-book })) -->
         % Then find the book with the title which-book
         let ( book-to-be-returned: book = arb({ b | (b: book) book(b) &
                          title-of-book(b) = which-book }))
            % And check the book back in
            set-attrs( book-to-be-returned,
                          'on-shelf, true,
                          'book-out, false )
```

Figure 5.18. Refine Specification of the Return-Book Function

This problem is further complicated by the fact that supporting logic must be provided to drive a discrete event simulation of the actions of the elevator mechanisms.

The Refine solution to the elevator control system problem can be found in Appendix H. Since the graphical and Refine specifications for this problem do not introduce any new concepts, only the most important parts of the graphical specification, the Do-Elevator, Handle-Events, and Check-State functions, will be discussed in this section. The entire graphical specification can be found in Appendix I.

5.5.1 The Do-Elevator Function The Do-Elevator function, shown in Figure 5.19, is the main function in the elevator system and drives both the simulator and the elevator control subsystems. Do-Elevator continues executing basic system activities until all of the schedules of the elevators are empty and all of the elevators are idle. Until this situation occurs, the elevator finds all current system events (events that occur on or before the current system time), and calls the function to handle all of these events. Do-Elevator then calls the function Check-State for each of the elevators in the system to modify the state of each elevator, if necessary, based on current system conditions. Do-Elevator completes a cycle by removing all old events from the system, scheduling new events, incrementing the system clock, and displaying the current state of the system.

5.5.2 The Handle-Events Function The Handle-Events function, shown in Figure 5.20, uses an enumeration to handle each of the events handed to it. With each event, Handle-Events uses a multi-way selector to execute the proper event handler based on the type of event that is encountered. Once all of the events handed to Handle-Events have been handled, Handle-Events terminates.

5.5.3 The Check-State Function Like Handle-Events, the Check-State function, shown in Figure 5.21, uses the multi-way selector to determine the proper course of action given a particular elevator state. Because Check-State handles only one elevator at a time, Check-State has no need for the enumeration operation. Once the proper state handler has been determined by the multi-way selector, that handler is executed. Upon termination of the appropriate state handling function, Check-State also terminates.

5.6 The Graphical Specification Language Environment

Thus far, discussion about this graphical specification language has centered about the language itself. This concern with the development is obviously extremely important, but any programming or specification language is virtually useless without a supporting

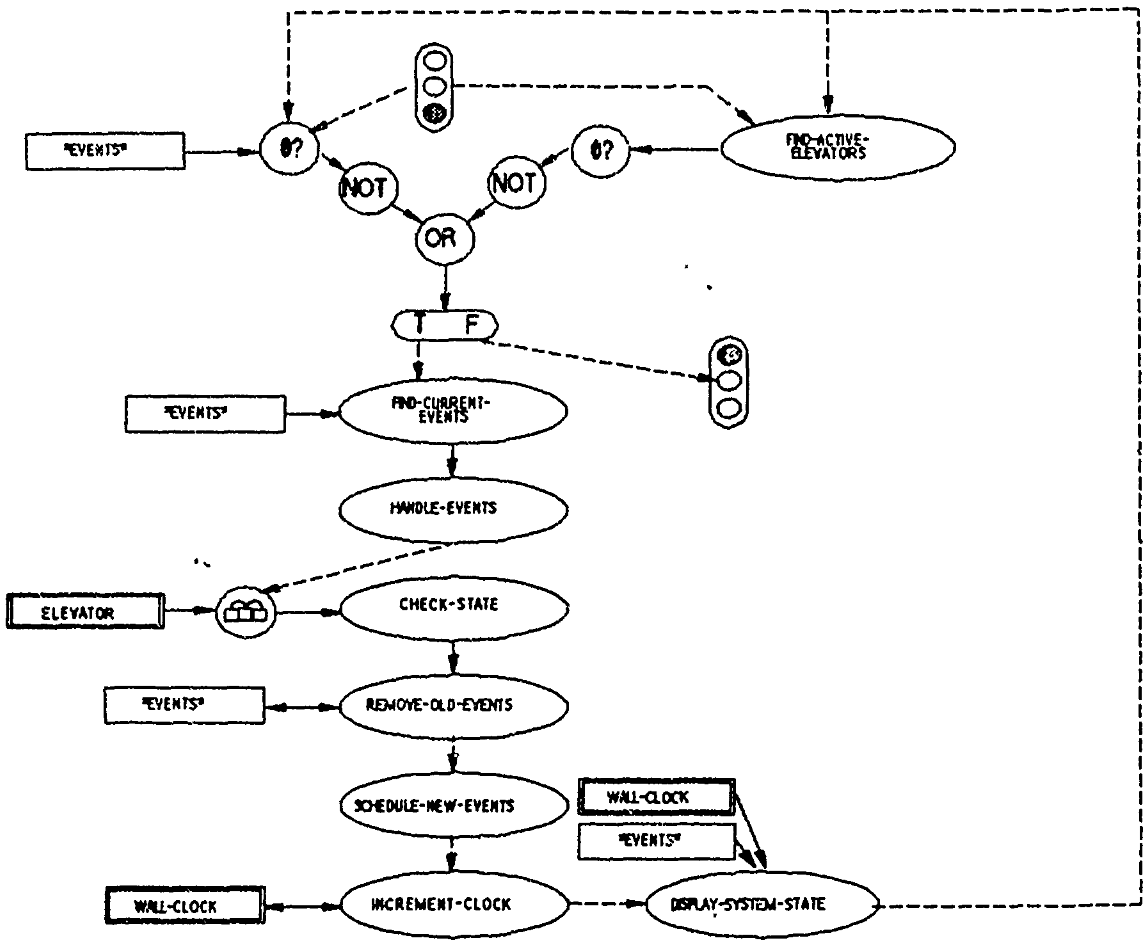


Figure 5.19. Graphical Formulation of the Do-Elevator Function

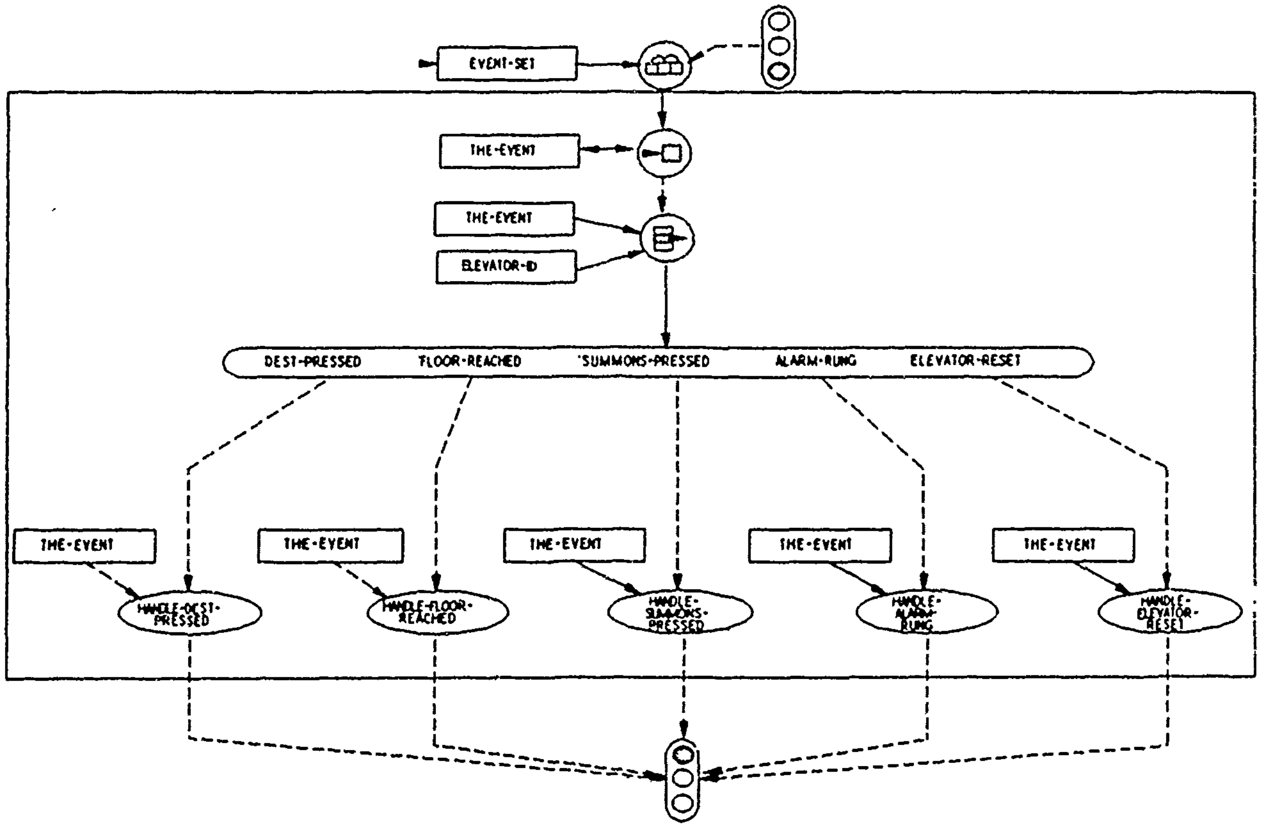


Figure 5.20. Graphical Formulation of the Handle-Events Function

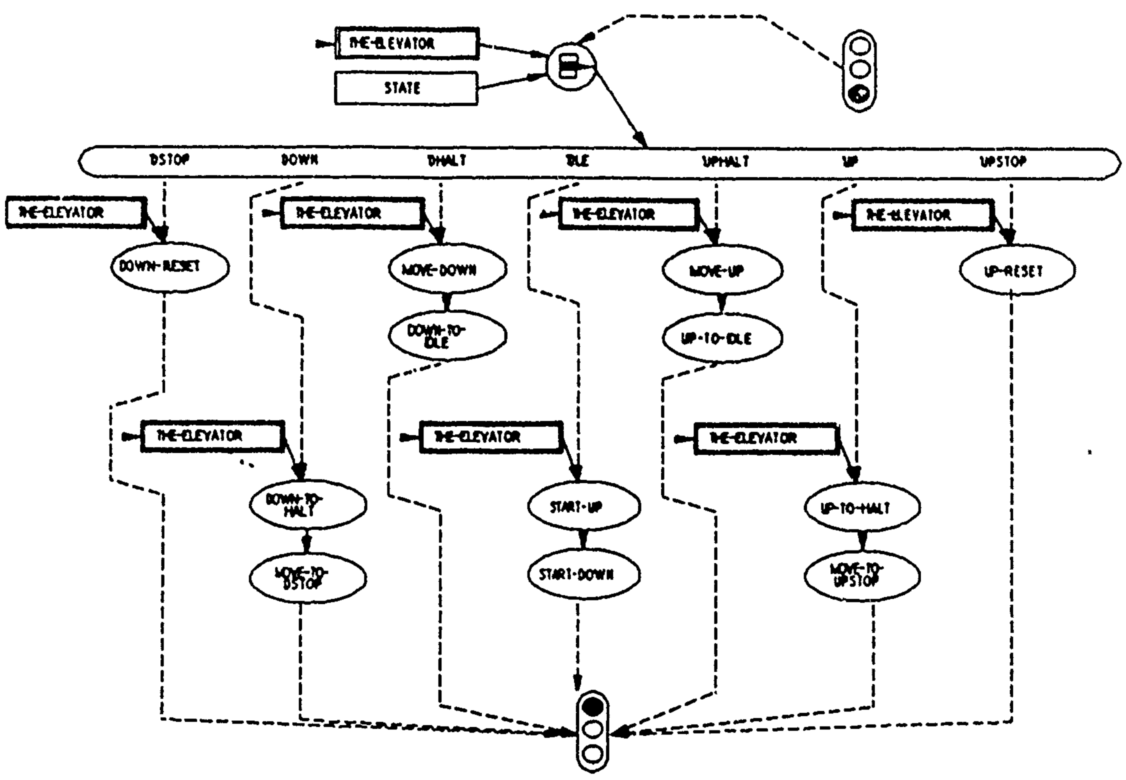


Figure 5.21. Graphical Formulation of the Check-State Function

environment to assist the programmer in using the language. This section will briefly discuss the features of an envisioned graphical specification language environment.

The environment to support this graphical specification language would ideally have seven components: an editor for the creation and manipulation of graphical specifications, a librarian to assist the user in locating and using existing software components, a debugger which would graphically display the execution of a particular specification as well as allowing the user to set breakpoints and to retrieve and assign values to variables at any time during the execution process, an icon editor that would allow the user to design new icons for user defined functions, a converter that would assist in the conversion of existing graphical programming notations into this language, a translator which would produce the executable Refine code from a graphical specification and vice versa, and an on-line help facility which could provide the user with information about any part of the graphical environment at any time. With these tools at his or her disposal, the programmer would not be forced to leave the graphical environment for any reason; all software development and maintenance could occur within the graphical specification environment. A graphical representation of the envisioned system architecture is contained in Figure 5.22.

The editor would be the user's primary interface with this graphical system and as such, should be as comfortable for the user as possible. Because of the graphical nature of the entire environment, the editor would have much more in common with full featured drawing programs or computer assisted design programs than with traditional text oriented editors. The editor would ideally allow the user to have multiple windows open at once, allowing the user to cut and paste sections of graphical "code" between windows. The editor should allow the user highly flexible routing of network arcs and should support a

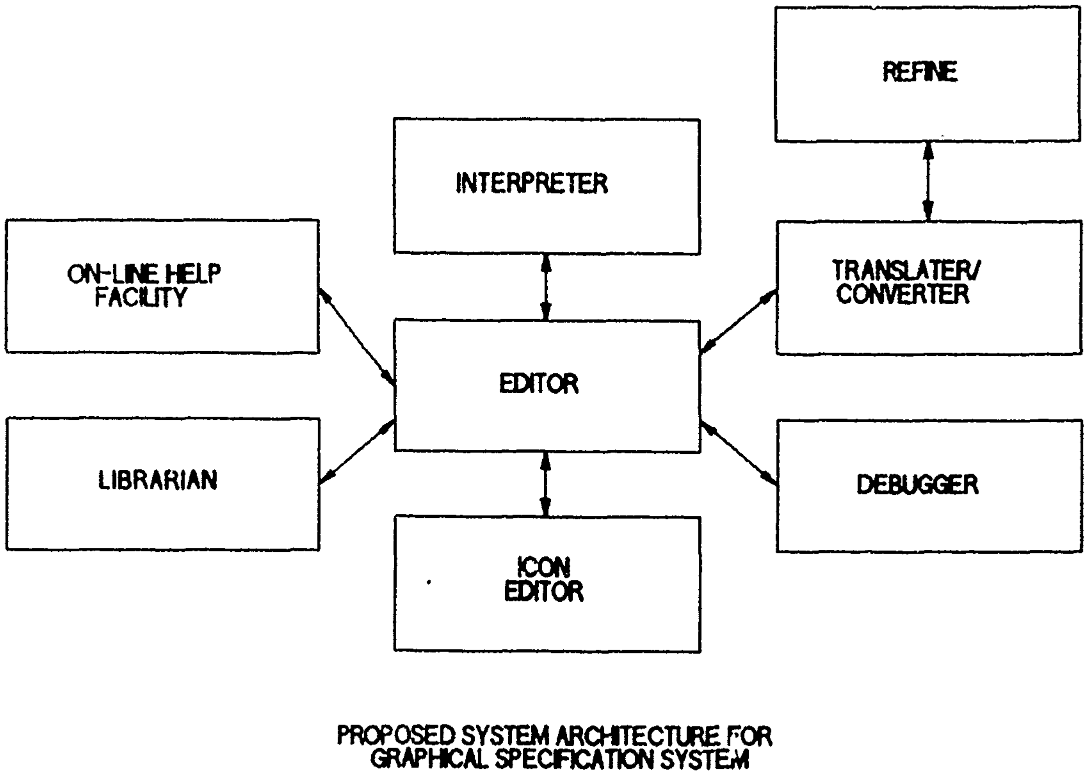


Figure 5.22. Architecture of Graphical Specification System

very large construction space for specification creation and maintenance.

The icon editor would allow the user to create new icons for user defined function or could even possibly be used to change the iconic representations for existing language operations, although such an application would be discouraged in order to maintain a relatively standardized graphical language. This control over iconic representations would provide the programmer a powerful tool for constructing representations that are meaningful both for the programmer and the programmer's clients.

The librarian would be the primary tool for categorizing and then retrieving software components. The librarian would be the primary vehicle for implementing data encapsulation mechanisms, the binding of data and operations together, through the use of software component toolkits similar to those discussed by Ingalls, et al (16). Used in conjunction with the icon editor, the librarian could conceivably be used to construct domain specific toolkits which would allow non-programmers to at least partially specify their system requirements.

A graphical debugger would enhance programmer productivity by allowing him or her to actually see a specification execute in its graphical format. A graphical representation of specification would improve programmer/client communications by allowing the client to see exactly what a specification is doing, thus enhancing the client's ability to understand in ramifications of his or her system requirements. The graphical debugger would also have features commonly found in today's debuggers such as the ability to set breakpoints and the ability to inspect and set variable values.

Because of the large amount of program documentation that now exists in some form of graphical programming notation, it would be extrememly beneficial to have a tool capable of converting these existing notations into graphical specification notation. Although a fully executable graphical specification is unlikely to be produced by such a conversion tool, the tool could provide a significant head start in the reimplementation of existing systems using formal specification methods. The heuristics already presented offer a starting point for the construction for such a conversion tool, but an effective implementation of a conversion tool should be based upon a formal foundation defining a set of mappings between formally defined program design notations and a formal definition of the graphical specification language. Handling the varied dialects of these existing notations likely to be encountered would require an set of mappings to be defined between each dialect and the graphical specification language combined with some sort of knowledge base to assist in deciding which sets of mappings to apply to a particular notation.

An on-line help facility would greatly increase programmer productivity by reducing the time the programmer spends retrieving and searching through system documentation. The help facility would allow the programmer to search system documentation through keyword searches or through a documentation index.

The final component, the translator, provides the means for translating graphical specifications into textual specifications and vice versa. This tool is obviously important because it provides the graphical specification with access to the Refine translator which can then translate the Refine code into a more traditional programming language, e.g. C or Ada.

Each of these components, then, provide key capabilities not only to access the graphical specification language, but also to help the programmer manage and maintain complex specifications with as little additional effort as possible.

5.7 Assessment of the Graphical Specification Language

As the reader might recall, a number of objectives were established at the outset of this research in order to guide the development of this graphical specification language. These objectives, grouped into the categories of language clarity and language utility, are

- Clarity of Language
  - Abundant use of perceptual coding of information
  - Restrict users to easily understood objects
  - Graphical specfications that differ conceptually should also differ visually
  - Mechanisms to facilitate data abstraction
- Utility of Language
  - Allow easy and accurate revision of specifications
  - Mechanisms to facilitate and encourage software component reuse
  - Mechanisms to facilitate data encapsulation

With the language development complete, it is now possible to assess the graphical language against these criteria.

5.7.1 Abundant Use of Perceptual Coding of Information The original purpose of this objective was to emphasize the importance of non-symbolic representations, such as spatial orientation, as well as symbolic representations in the graphical language. This graphical specification language uses perceptual coding in several forms. The first use of perceptual coding in this language is the use of directed arcs to join operator nodes to data nodes and to other operator nodes. This coding is perceptual and not symbolic because the joining relies on the spatial adjacency of the directed arc and its originating and terminating nodes. This perceptual coding is perhaps the most important coding in the graphical specification language because it is this physical adjacency that allows the user to visualize the flow of data through a particular specification. This perceptual coding is also important because its absence immediately notifies the user that some part of the specification has been incorrectly specified.

Another example of perceptual coding in this graphical specification language is the use of the start and stop symbols denoting the beginning and ending of execution of a particular specification segment. The start and stop symbols are perceptual because they are placed near the end of a specification function and connected to the starting and ending node or nodes by a directed arc, thereby providing the user additional clues about the starting and stopping points of a particular graphical specification function. This perceptual coding is important because it, like the directed arcs connecting operator nodes, helps the user to visualize program execution, in turn providing the user with a better overall understanding of a specification's behavior.

A final example of perceptual coding in this graphical language is the use of boxes around groups of operator nodes to indicate the execution of an entire block of operations under the control of another operation, for example a selector operator or an enumeration operator. Although the box itself is an example of symbolic coding, the placement of the box around a group of operations is clearly an example of perceptual coding because such a placement entails a spatial relationship between the box and its "client" operators. As with the other two examples of perceptual coding, the box is important because it too helps the user to visualize and to understand the underlying behavior of the function in which it is used.

Perceptual coding, then, has been widely used throughout this graphical specification language and, indeed, is an integral part of the language.

5.7.2 Restrict Users to Easily Understood Objects Great pains have been taken to represent each of the Refine primitive operations in symbols that, if not immediately recognizable, can be rapidly associated with a particular Refine concept or operation. The provision of an icon editor in the graphical specification language environment provides the user with a valuable tool for customizing the language environment even further to assure the existence of easily understood objects.

It is important to point out here that there has been absolutely no attempt at restricting users to any set of components, however. The approach adopted in the development of this language was to provide the user with a set of specification tools which would be applicable in a wide variety of situations. As mentioned in the discussion of the graphical specification environment, however, it would be within the user's power to construct domain specific "toolkits" from which special purpose specifications could be rapidly assembled. The user providing the toolkits, then, could conceivably restrict the toolkit user to only the "easily understood" objects found in the toolkit.

5.7.3 Graphical Specifications that Differ Conceptually Should Also Differ Visually The conceptual foundation for this graphical specification language, the dataflow graph, enforced this objective well. Since any change to the dataflow network is a change to the specification itself, and since network changes are made immediately obvious to the user through the presence, or lack thereof, of a directed arc, the user can quickly detect a difference in specifications by simply comparing the dataflow networks constituting the specifications in question. The carefully chosen icons serve to make differences between specifications even more apparent because of the distinctiveness of each operator's iconic representation.

This objective was compromised somewhat by the decision to overload graphical operators. That decision, made in an attempt to reduce the overall number of concepts that the user was required to master, ultimately made different specifications look somewhat more similar because the same icon was used to represent different operations. This increase in similarity is very slight, however, because the user still has a number of clues, such as the number and types of the operands, that allow him or her to distinguish between the various operations that the icons represent. The gain in language clarity and utility by minimizing the number of primitive symbols used in the language far outweighed the minor compromise of this objective. Overall then, this objective has been satisfactorily achieved.

5.7.4 Mechanisms to Facilitate Data Abstraction The demonstrations of this graphical specification language in the library control system and elevator control system problems bear witness to the excellent data abstraction capabilities of the language. Using the dataflow graph as a foundation for this graphical specification language proved to be a wise choice for satisfying this objective. Davis and Keller point out how naturally dataflow graphs lend themselves to functional composition (10:31). This graphical language has taken these natural properties and expanded them not only to include operator composition but also data type composition in order to provide the user with powerful tools for managing specification complexity.

5.7.5 Allow Easy and Accurate Revision of Specifications Although the modularity of this graphical specification language provides the basic facilities to accomplish this objective, language features alone cannot satisfy this objective. The editor provided as part of a graphical specification environment, as discussed earlier, would be the primary means of satisfying this objective. In order to acheive the desired ease of editing, the graphical specification editor must behave similarly to a computer aided design tool than to any of the traditional text editors that are now popular. Such an editor would allow the user to move, alter, add, and delete specification icons through the use of a mouse, a graphics pad, or any similar input device. The editor would also allow a user to create connections between nodes by simply pointing to the two nodes that he or she wishes to join. The editor would also perform continuous type checking within the specification, refusing to allow connections that are not defined within the language. This continuous type checking would provide the user real-time feedback about the validity of a particular specification, thereby increasing programmer productivity and increasing specification accuracy.

Unfortunately, a full implementation of this editor is beyond the scope of this research. The satisfaction of this objective lies in the successful graphical specification editor.

5.7.6 Mechanisms to Facilitate and Encourage Software Component Reuse This objective, too, cannot be satisfied by the language development alone, but relies on the eventual implementation of the graphical specification environment. The data abstraction facilities of the language provide the language modularity necessary to construct reusable components and the categorization of the Refine primitive operations by data type provides an elementary foundation for the construction of a component librarian system. Neither the data abstraction facilities nor the operation categorization, however, provide the retrieval facilities or the user help facilities necessary to actually encourage component reuse. The satisfaction of this objective lies in the implementation of the component librarian, discussed earlier, which would automatically categorize and then retrieve existing software components based on a user's particular need.

5.7.7 Mechanisms to Facilitate Data Encapsulation The early categorization of the primitive language operations by data type provides an elementary foundation for the binding of data types and their associated operations. The real associations between data types and their operations would be forged in the construction of the toolkits which would be partitioned by data type, each partition containing the appropriate operations. Data encapsulation, however, also typically implies the protection of data through the restriction of data access to a limited set of operations. While this graphical specification language discourages the manipulation of a data type's internal representation by only displaying a top level view of each instance of a data type, the language provides no protection against access of internal data structures. This objective, then, also waits on an environmental implementation, specifically a librarian function, for partial satisfaction. Unfortunately, because data protection facilities are unavailable in the Refine system underlying the graphical language, such facilities would be extremely difficult to implement within the graphical specification language.

5.7.8 Overall Language Assessment Not surprisingly, the objectives associated with language clarity were met while the objectives associated with language utility were not satisfied because of their dependence on a supporting graphical specification environment which does not exist at this time. Nevertheless, facilities were provided within the language to support the tools that would eventually satisfy the language utility objectives. Overall, this graphical specification language has fulfilled, or will potentially fulfill, all of the objectives initially established to ensure the usefulness of the language.

VI. Conversion Heuristics for Existing Graphical Requirements Analysis/Specification Notations

Although this graphical specification language provides an excellent tool for creating, maintaining, and understanding the intended behavicr of new software systems, there has been no discussion up to this point about the creation of tools designed to ease the maintenance of the enormous number of existing systems which have been documented using traditional requirements analysis and specification notation, such as dataflow diagrams (DFDs), entity relationship diagrams (ERDs), and state transition diagrams (STDs). This chapter addresses this need to support the maintenance of existing systems as well as the creation of new systems. The chapter discusses three existing graphical programming notations, DFDs, ERDs, and STDs, and shows how each notation can be converted into the developed graphical specification notation. The reader should be aware that the conversion processes described in this chapter are informal heuristics meant to guide the conversion process. Because of the numerous dialects in use for each of these graphical notations, it is extremely difficult to develop a general conversion heuristic; one should not hesitate to tailor these heuristics for a particular particular.

6.1 Dataflow Diagram Conversion

Data flow diagrams have long been a popular method for facilitating requirements analysis and documenting high level software system specifications. Data flow diagrams are popular because, as Sommerville points out, "They represent one way of describing a system and have the advantage that they are intuitive and readily understood without special training" (17:234). Because both data flow diagrams and the graphical specification language are based on the concept of data flow graphs, the two notations have a great deal in common, and, therefore, it should be relatively easy to convert between notations.

The graphical specification notation is intentionally very similar to the data flow diagram notation, although the existence of several dialects of data flow notation (17:235) hampers any attempt at general consistency between the notations. In the version of DFD notation circles discussed earlier, are used to represent transformation operations where a set of inputs is transformed into a set of outputs. Rectangles represent data stores; circles represent system input or output; arrows represent the direction of data flow in the system, and the words "and" and "or" "are used to link data flows when more than one data flow may be input or output from a transformation centre" (17:235).

One important point to make here is that the proposed graphical specifications contain much more detailed information than DFDs. This difference between the information content of the two notations basically eliminates any hope of conversion from a formal graphical specification to DFD formal graphical specifications can preserve all of the information contained in the DFD because of the graphical specification's more sophisticated data modeling techniques. One rule of thumb that should be adhered to in the conversion from graphical specification to DFD is that any node in the graphical specification containing a primitive specification operation, e.g. a set operation, a numeric operation, etc., becomes a primitive node in the DFD. In other words, no node containing a primitive operation will be further decomposed in the conversion from graphical specification to DFD.

The conversion of the DFD transformation representation into the graphical function representation is as simple as exchanging a circle for a rounded rectangle and vice versa. Because DFD notation does not have a representation for any explicit variables (due to the high level nature of DFD notation), no variables would appear in the conversion from DFD to specification notation and all such variables would disappear in the conversion from specification to DFD notation. The one exception to the elimination of explicit variables would be in the case of input/output (I/O) variables. In this case, the conversion from DFD to specification notation would create argument variables (a rectangle with a right pointing arrow) for each input circle encountered in the DFD and would create ports on the stop signal for all output circles. In the conversion from the specification to the DFD, all I/O variables and stop symbol ports would be replaced with circles.

In DFDs, the rectangle represents a data store, a persistent, non-sequential storage area for homogeneous data. When described in this manner, the data store sounds very similar to a set (non-sequential, homogeneous) of objects (persistent). Since there is no direct counterpart for the data store in specification notation, data stores will be converted into sets of objects during the conversion into specification notation. In the conversion from specification notation into DFD notation, the situation is more difficult. Since the graphical specification formulation is necessarily more detailed than a typical DFD and since a set of objects is a perfectly valid data structure in the graphical specification notation, there is no way to differentiate between the sets that had originally been data stores and the sets that have been added since the original conversion from DFD to specification notation. At the risk of introducing too much detail into the derived DFD, the conversion process shall create a data store in the derived DFD for every set of objects in the graphical specification. Since sets of objects and data stores are used for roughly the same purpose, this approach seems like a reasonable trade-off in order to keep the conversion process as simple as possible.

The use of the keywords "and" and "or" to indicate associations between data flows can be roughly approximated in the graphical specification language. Although not explicitly discussed up to this point, the reader might have already realized that if multiple arcs originate from a node, data is emitted on all arcs simultaneously upon emergence from a node, a situation which is analogous to "and" association in DFD notation. "Or" association, on the other hand, is handled by the passing of a boolean value to a selector which then passes control flow to the proper node. With these correspondences in mind then, the conversion process from DFD notation to specification notation would entail removing all "and" associations from data arcs and replacing all "or" associations with an additional arc originating at the same node as the data arcs and terminating at a selector node. The selector node then sends one control signal for each alternative to the proper terminating node. The conversion from graphical specifications to DFDs essentially reverses this process; replace selector/control signal constructions with "or"associations and then place "and" associations on all other data arcs that originate from nodes which have multiple output arcs.

Finally, start and stop symbols in graphical specifications must be handled. In the conversion from DFD to graphical specification, the DFD network can be traversed to find the node or nodes which do not have incoming data flows which originate from any other node. This node(s) is then the start of execution for the DFD and a start symbol may be located with control signal(s) emanating from the start symbol and terminating at the node(s). The stop symbol is handled similarly; a node(s) is found that does not have any data arcs originating from it that terminate at anothe. node. A stop symbol is then placed on the graphical specification. If the node(s) has an arc the passes data out of the node, then a corresponding arc is constructed in the graphical specification with an arc leading from the node(s) to a port in the stop symbol. If the node(s) passes no data out of the node, the a control signal is constructed from the node to the stop symbol.

In concise checklist format then, one possible heuristic for converting a DFD to a graphical specification is:

- Replace rounded rectangles with circles
- Replace input circles with incoming data symbols and replace output circles with ports on the stop symbol
- Replace data stores with sets of objects
- Remove all "and" associations
- Replace all "or" associations with an additional data flow to an appropriate selector node and appropriate control signals to the terminating nodes
- Locate the start(s) of the DFD network and establish control signals originating at the start symbol and terminating at the starting node(s).
- Locate the end(s) of the DFD network. If the end(s) do not pass data out of the transform then construct a control signal from the end(s) to the stop symbol.

As an example of the use of this heuristic, the reader should consider the DFD shown in Figure 6.1. This DFD depicts incoming data, "data-1", being sent to the process "proc-

1" and then, based on an evaluation by "proc-1", a transformation of "data-1" is either sent to processes "proc-2" and "proc-3" as "data-2" and "data-3" or is sent to process "proc-4" as "data-4". The results of the data transformations performed by the selected process(es) are then passed out of the structure depicted in the DFD.

The graphical specification contained in Figure 6.2 was constructed by following the steps of the heuristic just established. All rounded rectangles have been replaced with circles and input and output circles or the original DFD have been replaced with incoming data symbols and stop symbol ports respectively. The one "OR" association has been replaced with a two way selector which selects the proper execution path based upon an undefined set of conditions "cond-1" and "cond-2". The "AND" condition is simply removed, indicating that either both "proc-2" and "proc-3" will be executed or "proc-4" will be executed. Finally, start and stop symbols are established to indicate the starting process, "proc-1", and stopping processes, "proc-2", "proc-3", and "proc-4".

6.1.1 State Transition Diagram Conversion State transition diagrams (STDs) can basically be viewed as DFDs turned inside out: an arc in a DFD is represented as a node in a STD and a node in a DFD (whether represented by rounded rectangles or circles) is represented as an arc in a STD. Because of their conceptual similarity to DFDs, the conversion of STDs into graphical specification notation and vice versa should bear a strong resemblance to the conversion process for DFDs.

The first step in the conversion process from the STD to the graphical specification is the creation of one node in the graphical specification for every arc in the STD. These nodes in the graphical specification will eventually contain the transition logic necessary

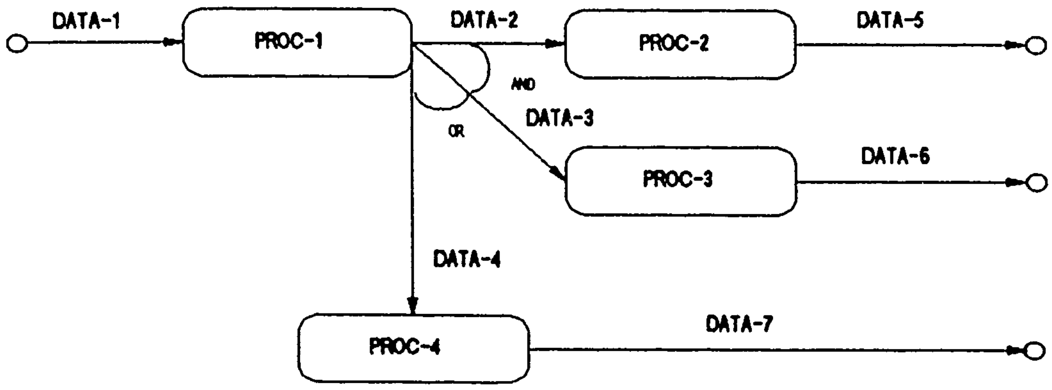


Figure 6.1. Example Data Flow Diagram

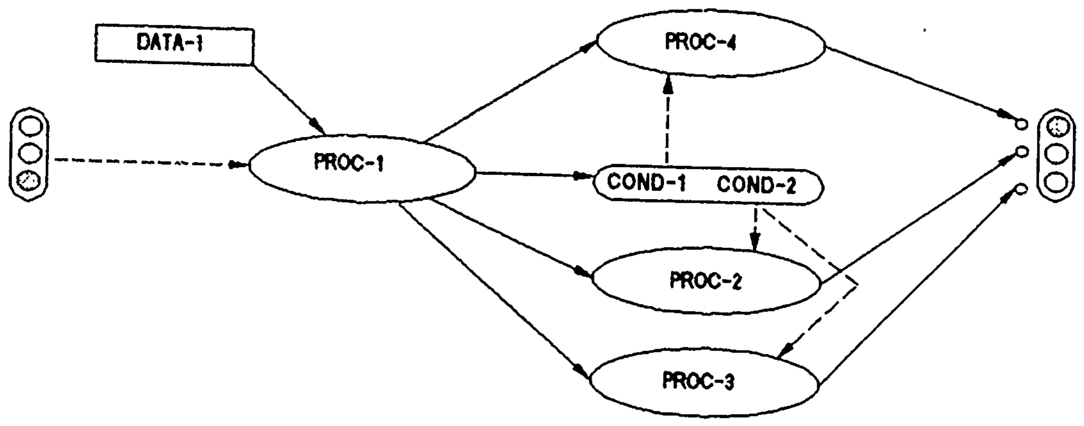


Figure 6.2. Graphical Specification of Example Data Flow Diagram

to determine whether to perform a state transition and the processing logic to carry out a state transition.

The next step in the conversion process is to group the newly created specification nodes by their originating state. This step is important because it will determine which state transition determination and processing nodes must be called when the system is in a particular state.

Finally, a STD usually represents a system or object of some sort that is going through a series of state changes. An object in the graphical specification must then be constructed to represent the system or object being represented in the STD. An graphical specification object is chosen as the data structure to represent the STD system because of the persistent nature of specification objects. Just as the system continues to exist from transition to transition, a specification object continues to exist from transformation to transformation. Also, the STD system is likely to be modeled through a collection of highly diverse information; likewise, an object represents a collection of heterogeneous information. Therefore, an object is a very appropriate data structure to model a STD system.

The final step in the conversion process is to establish a large, multi-way selector in the graphical specification. The state of the object should then be retrieved and handed to the selector. The selector then passes a control signal to the node or nodes which are responsible for determining the need for a state transition and for processing information to make that state transition occur.

To summarize then, the steps in the heuristic for converting a state transition diagram into a graphical specification are:

- Create one node in the graphical specification for every arc in the state transition diagram and give each node a unique, meaningful name
- Group the nodes by their originating states
- Establish an object in the graphical specification to represent the system represented in the state transition diagram
- Establish the mechanisms to retrieve the object's current state and pass this state to a multi-way selector; the multi-way selector then activates the function or functions responsible for determining the need for a state transition and for implementing that transition if needed.

To demonstrate this process, the reader should consider the elevator system described earlier. The state transition diagram for the elevator system is shown in Figure 6.3 (3). The first step in the conversion process is to create nodes for every arc in the STD, in this case 12 nodes. The second step is to group the nodes by the originating state of the arcs they represent, a simple task in this case. The next step is to construct an object to represent the system depicted in the STD, in this case an elevator. The final step is to establish mechanisms to extract the state of the object and pass that state through a multi-way selector so that the proper functions can be executed. The resulting graphical specification is shown in Figure 6.4.

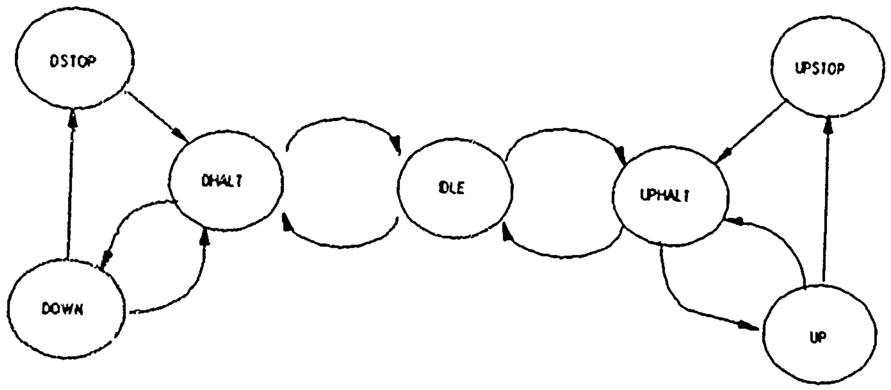


Figure 6.3. State Transition Diagram for Elevator System

6.1.2 Entity Relationship Diagram Conversion Entity relationship diagrams (ERDs) represent, as the name suggests, relationships between different collections of data. There is no restriction on the number of entities that may participate in such a relationship and each entity may have an unlimitied number of attributes associated with it (20:30-32). Since ERDs represent only data relationships, the conversion from ERDs to graphical specifications results only in the creation of graphical data structures; no graphical functions will be created in the ERD conversion process.

The map data structure immediately comes to mind when searching for a graphical representation for the ERD. Just as relations in ERDs depict associations between data structures, maps in graphical specifications represent associations between diverse data types. Although maps are primarily used for the representation of unary relations, the use of a tuple as the returned data type facilitates the representation of the n-way relations that may be encountered in ERDs. When depicting relationships, ERDs use straight lines

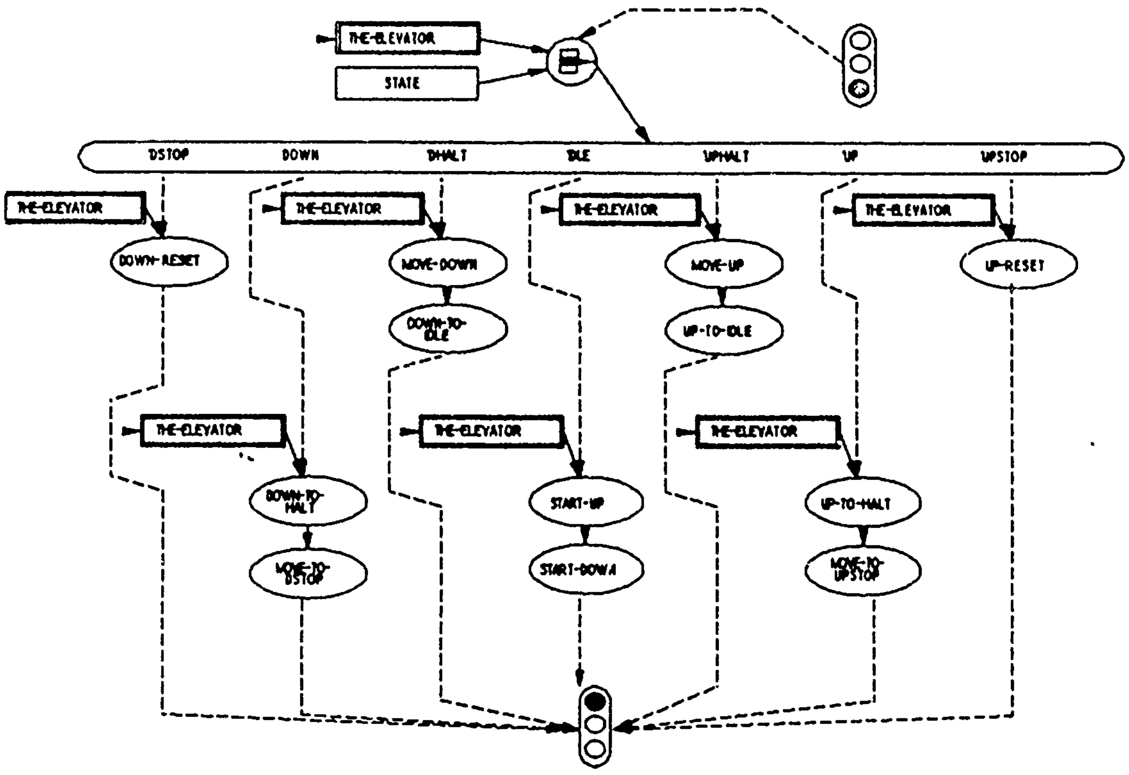


Figure 6.4. Graphical Specification Notation for Elevator State Transition Diagram

leading from the relation (a diamond) to the entity (a square) to represent the fact that many such entities can participate in the same relation; ERDs use an arrowed line from the relation to the entity to represent the fact that only one such entity may participate in each such relation. This distinction between single or multiple participation in each relation can be reflected in the map by returning either a value or a set of values, corresponding to the use of the arrow and the straight line in ERDs respectively. Since the complete ERD is simply a series of these relations, the graphical specification resulting from the conversion will simply be a series of maps corresponding to each relation.

The ERD conversion process, then, starts with the creation of specification data structures corresponding to the entities depicted in the ERD. The data structure corresponding most closely to the ERD entity is the tuple since both maintain collections of heterogeneous data and both have an unlimited number of attributes or fields repectively. Thus, the first step in the ERD conversion is to create tuples corresponding to each entity in the ERD. Each field name in the tuple should correspond to an attribute of the entity.

Now that the entities are represented in the graphical specification notation, the relationships between the entities can be encoded in the specification. As discussed earlier, maps will be used to represent relations because of their similar characteristics. Because relationships in ERDs are nondirectional, i.e. if entity A has a relationship with entity B, then entity B also has that relationship with entity A, within the limitations of the single or multiple entity participation described earlier, one map will be constructed for each entity participating in a relation. This allows a query of the relation from the viewpoint of any of the participating entities. Each map for a particular relation will have a domain data type of one of the entity tuples constructed earlier and will have a range data type

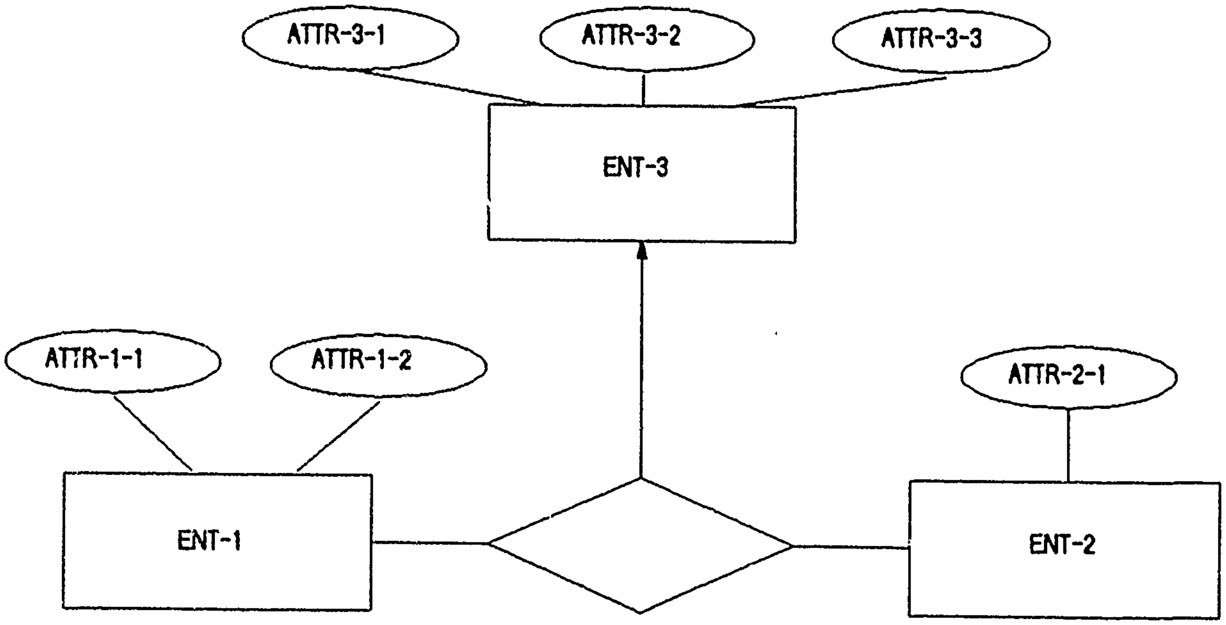


Figure 6.5. Example Entity Relationship Diagram

of a tuple with fields corresponding to each of the remaining entities participating in the relation.

For example, consider the simple ERD shown in Figure 5.26. This ERD has only one relation. Three entities participate in this relation. In order to convert this ERD into graphical specification notation, the entities must first be represented by tuples. Three tuples would be constructed: the first with fields named attr-1-1 and attr-1-2, the second with one field named attr-2-1, and the third with fields named attr-3-1, attr-3-2, and attr-3-3. The three tuple data types will be named ent-1, ent-2, and ent-3 respectively. The next step is to construct maps representing the relation. Since the relation has three entities participating, three maps are required. Since each map must have a domain data type matching one of the participating entities, the first map will have a domain type of ent-1, the second map will have a domain type of ent-2, and the third map will have the domain type of ent-3. Finally the range types must be constructed. Three tuples are required: one with fields set(ent-2), because there is no limitation and the number of entities that may participate, and ent-3, since only one entity may participate; one with fields set(ent-1) and ent-3; and one with fields set(ent-1) and set(ent-2). The notation "set(ent-X)" simply denotes a set of ent-X. Finally, the maps must be associated with their proper range types. The first map is associated with the first range type, the second map with the second type, and the third map with the third type. Thus the resulting structure of the first map is:

- Map 1
    - Domain
        - Ent-1
            - Attr-1-1
            - Attr-1-2
    - Range
        - Set(Ent-2)
            - Attr-2-1
        - Ent-3
            - Attr-3-1

· Attr-3-2

· Attr-3-3

The other maps have corresponding structures. The reader should note however that because of the data abstraction features of the graphical specification language that have been discussed throughout this paper, this entire map data structure would appear as a single rectangle labeled "Map-1" in graphical specification notation.

Concisely stated then, the heuristic for converting entity relationship diagrams into graphical specification notation is:

- Create tuples for each entity in the ERD. Each tuple should have fields corresponding to the attributes of its associated entity
- Create one map for every entity in every relation in the ERD. Each map for each relation should have a different entity as its domain type.
- Construct the range tuples for each relation by creating fields for each entity in the relation except one. If the arc connecting the entity with the relation is directed, then the field type is simply the tuple already created to represent the entity. If the arc is undirected, then the field type is a set of the tuples already constructed to represent the entity.
- Associate each map for each relation with its appropriate domain tuple. If a relation has n entities participating, then the domain of the map should have one of those entities and the range of the map should be a tuple composed of the $n-1$ remaining entities.

6.2 Assessment of Notation Conversion Heuristics

The notation conversion heuristics presented offer one possible way of building formal specifications for existing systems using available system documentation. However, these heuristics assume one particular dialect of each of the notations discussed; the heuristics must be modified to handle other dialects. The development of a general conversion heuristic is beyond the scope of this research effort. As a future step in the development of this research area, these heuristics should be used as the foundation for a set of formalized algorithms which would not only allow the conversion of a number of dialects of each of these notations, but would also allow the automation of the conversion process.

VII. Conclusions and Recommendations for Further Research

7.1 Summary

The graphical specification language developed in this research represents a tool that could significantly decrease the training time required to use formal specifications as well as increasing the understandability and utility of formal specifications. Because this language overcomes many of the obstacles that have hindered the widespread use of formal specification for the development of software, this language could be a valuable vehicle for promoting a wider use of formal specification methods in the software industry.

This graphical specification language was developed by first establishing a number of objectives based upon expert opinions (7, 10, 11, 12) of the most critical elements contributing to the clarity and utility of graphical languages. A preliminary graphical specification language was then developed based on the SF specification language (4). This preliminary language was used to develop the techniques necessary to decompose a textual specification language as well as to develop symbologies to represent the key concepts in the formal specification of software. The techniques and symbologies developed during the initial graphical language construction were then applied in the development of a more sophisticated graphical specification language based on the Refine specification language. In addition to the language, a graphical specification environment was described which would allow the user to specify and execute formal specifications within a graphical framework.

7.2 C nclusions

A number of conclusions can be drawn from this research into the development of graphical representations for formal specifications.

- Graphical representations are viable alternatives to purely textual representation of formal specifications. Graphical representations should have significant advantages over textual specifications such as increased clarity and understandability and reduced training costs, but these benefits must be proven using a comprehensive set of metrics on the implemented graphical specificaton environment, a task beyond the scope of this effort.
- It is possible to implement a graphical specification system based upon the Refine wide-spectrum language as a foundation. Such a graphical specification system will dramatically improve the productivity of software engineers by retaining the advantages of the program transformation lifecycle while alleviating the severe problems of the clarity and understandability of the formal specifications upon which the program transformation lifecycle is based.
- Because of its emphasis on data types and operations, the language decomposition method developed here are applicable in the development of a graphical representation for any well-defined textual formal specification language.
- The proposed conversion heuristics facilitate the creation of graphical formal specifications based on existing graphical program documentation. This finding is extremely important to the acceptance of any fundamentally new system such as this graphical specification language because it allows a new user to begin creating formal

specifications for existing systems almost immediately using off-the-shelf documentation.

- Because of their additional information content, graphical specifications tend to be larger and more complex than existing graphical programming notations. Preliminary analysis and specification design using the simpler existing notations is still a worthwhile exercise since it enables the specifier to gain a greater understanding of the system before tackling the more complex graphical specification notation.

- The proposed graphical specification system architecture supports an incremental implementation of this system. By following the implementation recommendations in the following section, the system developer can deliver useful subsets of the graphical specification environment to the user for evaluation, thereby enabling the developer to make beneficial changes to the specification system before the system is completely implemented.

7.3 Recommendations

Recommendations for further research emphasize the need to continue this effort though the implementation and validation of the concepts developed in this thesis. The logical continuation of this research would

- Implement the complete graphical specification environment as described in Section 5.7.

- Formalize the graphical notation conversion heuristics presented in Chapter 6.

- Establish a set of metrics to test and validate the graphical symbology used to represent formal specifications.

7.3.1 Graphical Specification Environment Implementation The graphical specification environment to support this graphical language could be implemented in phases. The environment should be implemented in the following order:

- Graphical Specification Editor
- Graphical Specification Interpreter
- Translator/Converter
- On-Line Help Facility
- Librarian
- Icon Editor
- Debugger

in order to ensure that each phase of implementation produces a fully functional and useful tool that that can be used to test graphical specification concepts and can also be used to encourage outside experimentation with these tools and concepts. Any workstation-class computer supporting bit-mapped graphics, mouse interaction, and an event-driven environment would be a suitable platform for the implementation of the graphical specification environment. Both Sunview^{TM}, a graphical user interface shipped with Sun workstations, and X Windows, a non-proprietary graphical user interface developed at the Massachusetts Institute of Technology, are excellent examples of the user environments that would adequately support the development of the proposed graphical specification environment.

7.3.2 Formalize the Graphical Notation Conversion Heuristics As stated earlier, the proposed graphical notation conversion heuristics handle only one particular dialect of each of the graphical notations that were discussed. The construction of a general conversion heuristic is beyond the scope of this effort. As part of a future research effort, the conversion heuristics presented in Chapter 6 should be used as a foundation for the development of formalized conversion algorithms which would not only facilitate the conversion of a larger number of dialects, but would also allow the automation of the conversion process.

7.3.3 Validation Metrics for Graphical Specification Systems Glinert (11) proposes a set of metrics designed to evaluate the effectiveness of visual interfaces which could be used to evaluate this language. Once the first phase of environment implementation is complete, a set of metrics based on Glinert's work should be developed and administered to determine the effectiveness of this graphical language. Since Glinert's metrics use extensive surveys after hands-on use of the visual interface, they are not of much use at this point in the development of this graphical language. Before a graphical language like this can be widely adopted, however, some measure of its effectiveness must be available to justify the software industry's investment in such a tool. A set of metrics would also be extremely valuable in detecting and correcting any language "features" which detract from the effectiveness of the language.

All in all, a graphical specification language offers a significant improvement over traditional software development techniques. Such a language would build on the advantages of using the program transformation development technique while removing the obstacles that prevent the technique's widespread use.

Appendix A. *SF Specification for the Library Control System*

This appendix contains Berztiss's SF specification for the Library Control System problem (3) discussed in Chapter 3.

```
SEGMENT Titles;

IMPORTED SIGNALS Add~title, Drop~title, Move~title;

EXPORTED SIGNALS Catalog~copy;

IMPORTED TYPE Author ENDTYPE;

TYPE Title : T(SUBSETS:INCAT,HASBEEN);

SECONDARY SETS- Subject~area: Area;

FUNCTIONS- title~text : T -> Text;
authors     : T -> Author-set;
subjects    : T -> Ares-set;
ENDTYPE;

EVENT Add~title(newcopy;book; t:Text; A:Author-set; S:Area-set);
PRECONDITIONS- not(member(book,T));
SETCONDITIONS- INCAT' = INCAT \union {book};
MAPCONDITIONS- title~text'(book) = t;
authors'(book) = A;
subjects'(book) = S;
SIGCONDITIONS- (Catalog~copy(newcopy,book))ON;
ENDEVENT;

INTERNAL EVENT Reactivate(newcopy,book);
SETCONDITIONS- INCAT' = INCAT \union {book};
HASBEEN' = HASBEEN \union {book};
SIGCONDITIONS- (Catalog~copy(newcopy,book))ON;
ENDEVENT;

INTERNAL EVENT Drop~title(book);
SETCONDITIONS- INCAT' = INCAT - {book};
HASBEEN' = HASBEEN \union {book};
```

```
ENDEVENT;

(* Transaction 1 *)
TRANSACTION;
@(T~min.now):ON(Add~title(newcopy,book))OFF:
PROMPT(Add~title:newcopy,book);
ENDTRANSACTIONS;

(* Transaction 2 *)
TRANSACTION;
@(T~min.now):ON(Drop~title(book))OFF:Drop~title(book);
ENDTRANSACTION;

(* Transaction 3 *)
TRANSACTION;
@(T~min.now):ON(Move~title(newcopy,book))OFF:
Reactivate(newcopy,book);
ENDTRANSACTION;
ENDSEGMENT;
```

```
SEGMENT Copies;

IMPORTED SIGNALS Catalog~copy;

EXPORTED SIGNAL Drop~title, Move~title, Add~title;

IMPORTED TYPE Title:T  ENDTYPE;

IMPORTED TYPE Borrower:B  ENDTYPE;

TYPE Copy : C;
FUNCTIONS- book~id: C -> T;
borrowed: C -> Boolean(false);
last~out: C -> B(nil);
books~out: B -> Integer(0);
limit:   -> Integer(0);
ENDTYPE;

EVENT Set~limit(k:Integer);
MAPCONDITIONS- limit' = k;
ENDEVENT;

EVENT Check~copy(newcopy,book);
SIGCONDITIONS- member(book,INCAT) ->
(Catalog~copy(newcopy,book))ON;
member(book,HASBEEN) ->
(Move~title(newcopy,book))ON;
not(member(book,T)) ->
(Add~title(newcopy,book))ON;
ENDEVENT;

INTERNAL EVENT Add~copy(newcopy,book);
SETCONDITIONS- C' = C \union {newcopy};
MAPCONDITIONS- book~id'(newcopy) = book;
ENDEVENT;

EVENT Remove~copy(copy);
PRECONDITIONS- member(copy,C);
not(borrowed(copy));
SETCONDITIONS- C' = C - {copy};
SIGCONDITIONS- card({x| book~id'(x) = book~id(copy)}) = 0 ->
(Drop~title(book~id(copy)))ON;
ENDEVENT;

EVENT Check~out(copy,borr:B);
```

```
PRECONDITIONS- member(copy,C);
not(borrowed(copy));
books~out(borr)<limit;
MAPCONDITIONS- borrowed'(copy) = true;
last~out'(copy) = borr;
books~out'(borr) = books~out(borr) + 1;
ENDEVENT;

EVENT Check~in(copy,borr:B);
PRECONDITIONS- member(copy,C);
last~out(copy) = borr;
MAPCONDITIONS- borrowed'(coy) = false;
books~out'(borr) = books~out(borr) - 1;
ENDEVENT;

(* Transaction 1 *)
TRANSACTION;
@(T~min.now):ON(Catalog~copy(newcopy,book))OFF:
Add~copy(newcopy,book);
ENDTRANSACTION;

ENDSEGMENT;
```

Appendix B. *SF Specification for the Elevator Control System*

This appendix contains Berztiss's SF specification for the Elevator Control System problem (4) discussed in Chapter 3.

```
SEGMENT Elevator;

        IMPORTED SIGNALS Activate~elevator, Add~to~agenda, Move~idle;

        SENSOR SIGNALS Next~floor~sensor;

        MECHANISMS Door~open, Alarm, Light, Uplight, Dlight, Motion~up, Motion~down;

        IMPORTED TYPE Time: T ENDTYPE;

        IMPORTED TYPE Time~interval: TI ENDTYPE;

        TYPE Elevator: E;

                SECONDARY SETS- S = {"idle", "up", "uphalt", "upstop", "down",
                                                "dhalt", "dstop"};
                        Floor: F = Integer;

                FUNCTIONS-      state:        E -> S;
                                lowfloor:     E -> F;
                                highfloor:    E -> F;
                                clock:        E -> T;
                                delay:        E -> TI;
                                agenda:     EXF -> Boolean  (false);

                SENSORS-        floor~now:    E -> F;
                                nullweight:     E -> Boolean;

        ENDTYPE;

        EVENT Initialize~elevator(e; low, high: F; interval: TI);

        (* Parameter interval indicates the time for which the elevator door is
           to be kept open after it was last opened or a person stepped through
           it *)
        MAPCONDITIONS-  state'(e) = "idle";
                        lowfloor'(e) = low;
                        highfloor'(e) = high;
                        delay'(e) = interval;

ENDEVENT;

INTERNAL EVENT Activate~elevator(e; x: S);

(* Initiated by the dispatcher via signal Activate~elevator. *)

        MAPCONDITIONS-  state'(e) = x;
                        clock'(e) = T.now;

        SIGCONDITIONS-  (Door~open)ON;
                        x = "uphalt" -> (Uplight(floor~now(e)))OFF;
                        x = "dhalt" -> (Dlight(floor~now(e)))OFF;
                        (process~halt(e))ON;

ENDEVENT;

INTERNAL EVENT Enter~halt(e);

        MAPCONDITIONS-  agenda'(e, floor~now(e)) = false;
                        not(nullweight(e)) -> clock'(e) = T.now;

        SIGCONDITIONS-  not(nullweight(e)) ->
                        BLOCK
                                (Door~open(e))ON;
                                state(e) = "uphalt" ->
                                        (Uplight(floor~now(e)))OFF;
                                state(e) = "dhalt" ->
                                        (dlight(floor~now(e)))OFF;
                                (Light(e, floor~now(e)))OFF;
                                (Process~halt(e))ON;
                        ENDBLOCK;
                        nullweight(e) -> (Idle~elevator(e))ON;

ENDEVENT;

EVENT Press~button(e; floor: F);

(* Only floors in the direction of travel of the elevator may be added
   to the agenda *)
```

```
        PRECONDITIONS-  state(e) = "up" V state(e) = "uphalt" ->
                                floor > floor~now(e);
                        state(e) = "down" V state(e) = "dhalt" ->
                                floor < floor~now(e);
                        not(member(state(e), {"idle","upstop","dstop"}));

        MAPCONDITIONS-  agenda'(e, floor) = true;

        SIGCONDITIONS-  (Light(e, floor))ON;

ENDEVENT;

INTERNAL EVENT Add~to~agenda(e; floor: F);

(* Initiated by the dispatcher *)

        MAPCONDITIONS-  agenda'(e, floor) = true;

ENDEVENT;

INTERNAL EVENT Process~halt(e);

        SIGCONDITIONS-  Allop(^;{not(agenda(e, x))|lowfloor(e) <= x <=
                                highfloor(e)}) -> (Idle~elevator(e))ON;
                        Allop(V;{agenda(e, x)|lowfloor(e) <= x <=
                                highfloor(e)}) -> (Set~in~motion(e))ON;

ENDEVENT;

INTERNAL EVENT Passing~sensor(e);

        MAPCONDITIONS-  agenda(e, floor~now(e)+1) -> state'(e) = "uphalt";
                        agenda(e, floor~now(e)-1) -> state'(e) = "dhalt";

        SIGCONDITIONS-  agenda(e, floor~now(e)+1) -> (Motion~up(e))OFF;
                        agenda(e, floor~now(e)-1) -> (Motion~down(e))OFF;
                        agenda(e, floor~now(e)+1)Vagenda(e, floor~now(e)-1) ->
                        (Enter~halt(e))ON;

ENDEVENT;

EVENT Stop~elevator(e);

        MAPCONDITIONS-  state(e) = "down" V state(e) = "dhalt" ->
```

```
                                    state'(e) = "dstop";
                            state(e) = "up" V state(e) = "uphalt" ->
                                    state'(e) = "upstop";

            SIGCONDITIONS-  state(e) = "down" -> (Motion~down(e))OFF;
                            state(e) = "up" -> (Motion~up(e))OFF;
                                    (Alarm(e))ON;
                            (Door~open(e))ON;

    ENDEVENT;

    EVENT Reactivate~elevator(e);

            MAPCONDITIONS-  state'(e) = "upstop" -> state(e) = "uphalt";
                            state'(e) = "dstop" -> state(e) = "dhalt";

            SIGCONDITIONS-  (Alarm(e))OFF;
                            (Enter~halt(e))ON;

    ENDEVENT;

    INTERNAL EVENT Idle~elevator(e);

            MAPCONDITIONS-  state'(e) = "idle";
                            Allop(^;{not(agenda'(e,x))|lowfloor(e) <= x <=
                                    highfloor(e)});

            SIGCONDITIONS-  Allop(^;{(Light(e,x))OFF|lowfloor(e) <= x <=
                                    highfloor(e)});
                            (Door~open(e))OFF;

    ENDEVENT;

    INTERNAL EVENT Move~idle(e; floor: F);

    (* Initiated by the dispatcher *)

            MAPCONDITIONS-  agenda'(e, floor) = true;
                            floor > floor~now(e) -> state'(e) = "uphalt";
                            floor < floor~now(e) -> state'(e) = "dhalt";

            SIGCONDITIONS-  (Set~in~motion(e))ON;

    ENDEVENT;
```

```
EVENT Update~clock(e);

(* Initiated by breaking a light beam across the door of the elevator or
   by some similar device *)

        MAPCONDITIONS-  clock'(e) = T.now;

ENDEVENT;

EVENT Open~door(e);

(* This event is required for people to get out who somehow find
   themselves in an idle elevator.  Raising the flag Process~halt
   ensures that the opened door will ultimately close again. *)

        MAPCONDITIONS-  clock'(e) = T.now;

        SIGCONDITIONS-  (Door~open(e))ON;
                        (Process~halt(e))ON;

ENDEVENT;

TRANSACTION;

        @(T.NOW): ON(Activate~elevator(e))OFF:Activate~elevator(e);

ENDTRANSACTION;

TRANSACTION;

        @(T.now): ON(Add~to~agenda(e,floor))OFF: Add~to~agenda(e,floor);

ENDTRANSACTION;

TRANSACTION;

        @(T.now): ON(Move~idle(e,floor))OFF: Move~idle(e,floor);

ENDTRANSACTION;

TRANSACTION;

        @(T.now): ON(Enter~halt(e))OFF: Enter~halt(e);

ENDTRANSACTION;
```

```
          TRANSACTION;

          (* The delay is to give passengers time to press destination buttons *)

                    @(clock(e)+delay(e)): ON(Process~halt(e))OFF: Process~halt(e);

          ENDTRANSACTION;

          TRANSACTION;

                    @(T.now): ON(Set~in~nmotion(e))OFF: Set~in~motion(e);

          ENDTRANSACTION;

          TRANSACTION;

                    @(T.now): ON(Next~floor~sensor(e))OFF: Passing~sensor(e);

          ENDTRANSACTION;

          TRANSACTION;

                    @(T.now): ON(Idle~elevator(e))OFF: Idle~elevator(e);

          ENDTRANSACTION;

ENDSEGMENT;
```

Appendix C. *Graphical SF Specification of the Elevator Control System*

This appendix contains the formal specification for the Elevator Control System problem (4) using the SF-based graphical specification language developed in Chapter 3.

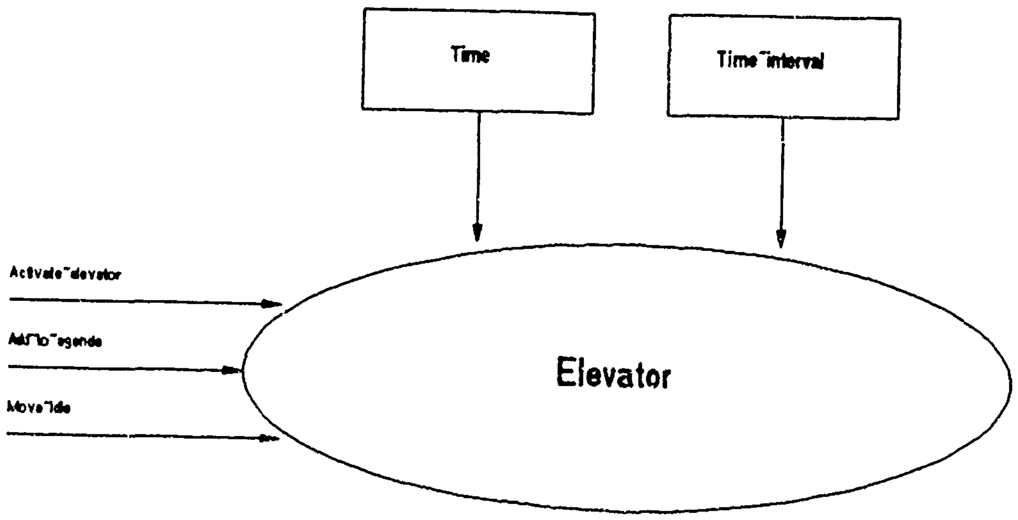


Figure C.1. Elevator Control System Segment Diagram

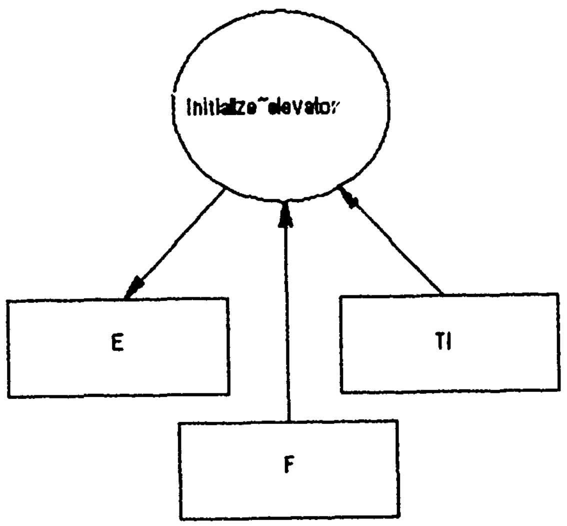


Figure C.2. Graphical Specification of Initialize elevator Event

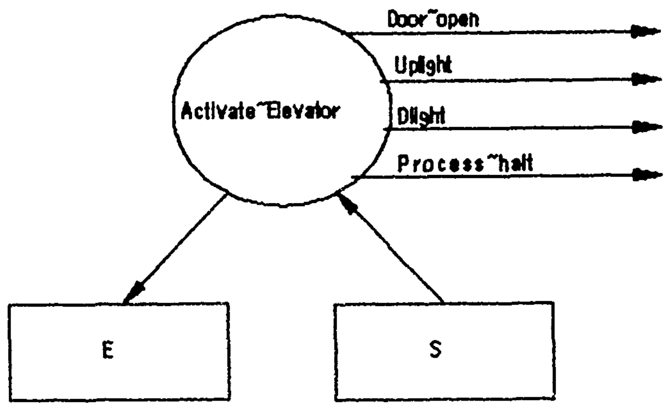


Figure C.3. Graphical Specification of Activate elevator Event

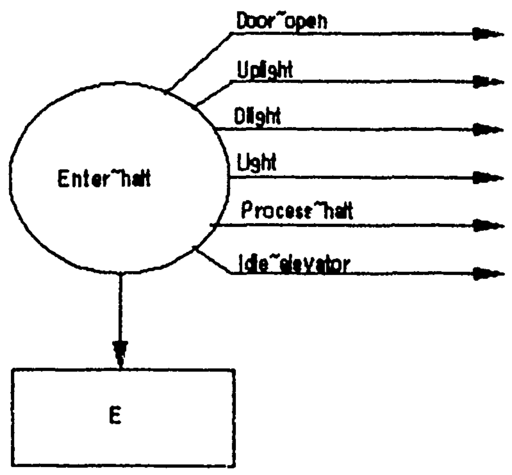


Figure C.4. Graphical Specification of Enter halt Event

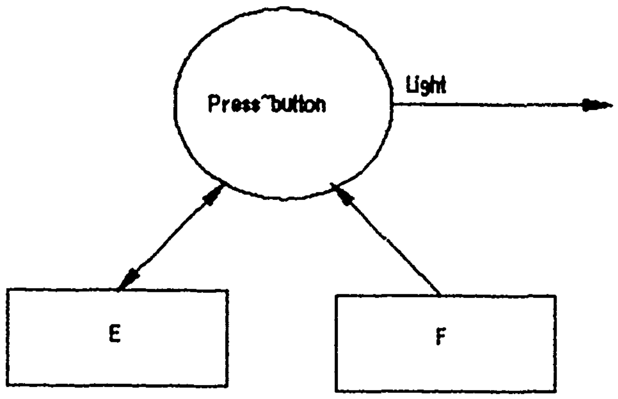


Figure C.5. Graphical Specification of Press button Event

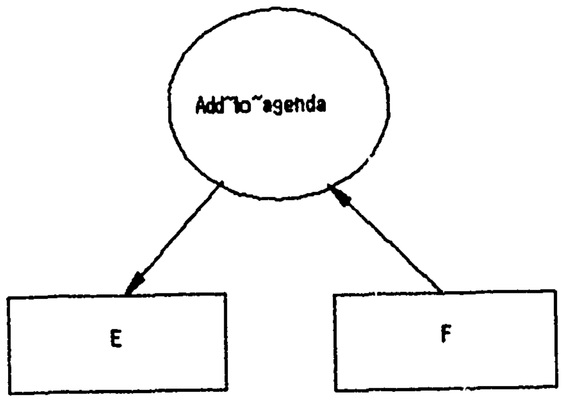


Figure C.6. Graphical Specification of Add to agenda Event

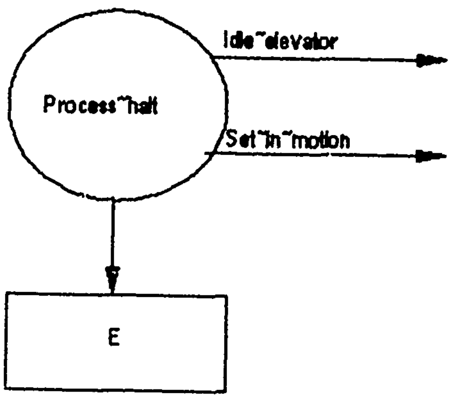


Figure C.7. Graphical Specification of Process halt Event

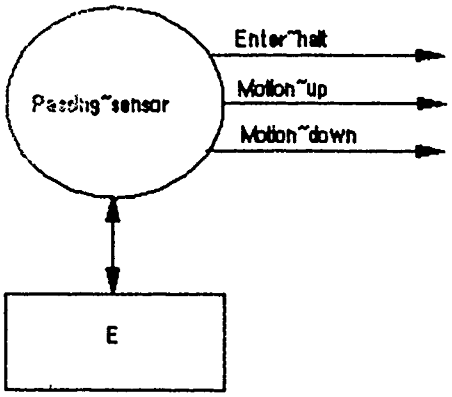


Figure C.8. Graphical Specification of Passing sensor Event

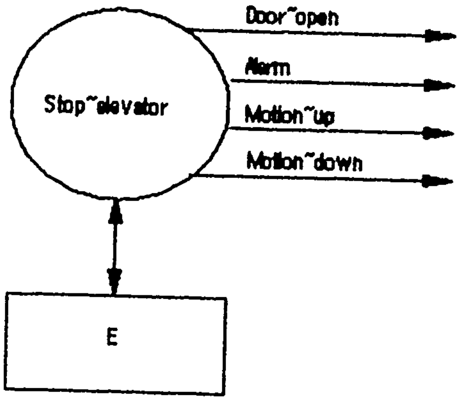


Figure C.9. Graphical Specification of Stop elevator Event

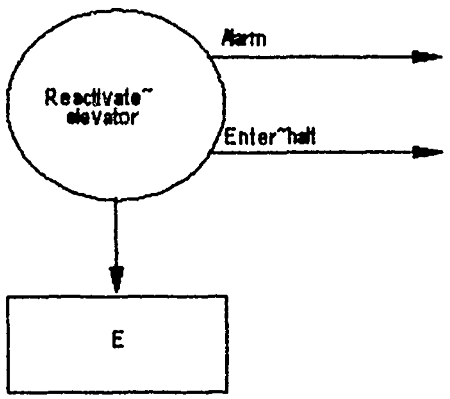


Figure C.10. Graphical Specification of Reactivate elevator Event

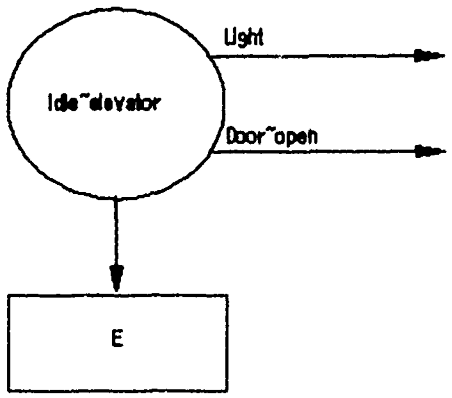


Figure C.11. Graphical Specification of Idle elevator Event

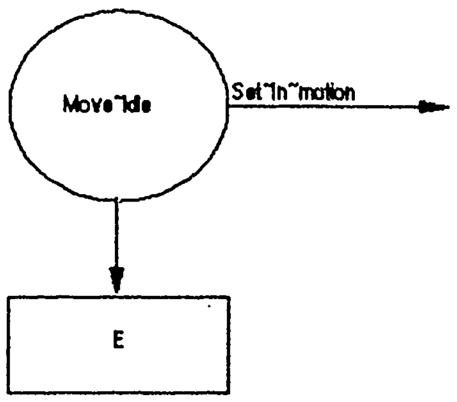


Figure C.12. Graphical Specification of Move idle Event

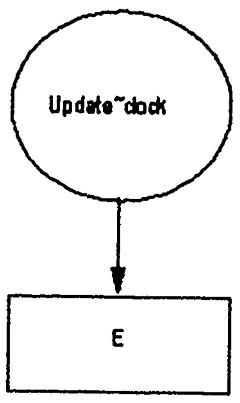


Figure C.13. Graphical Specification of Update clock Event

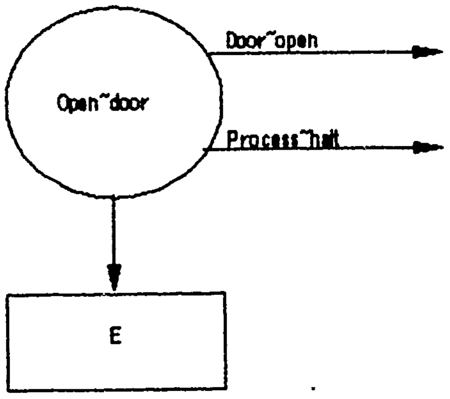


Figure C.14. Graphical Specification of Open door Event

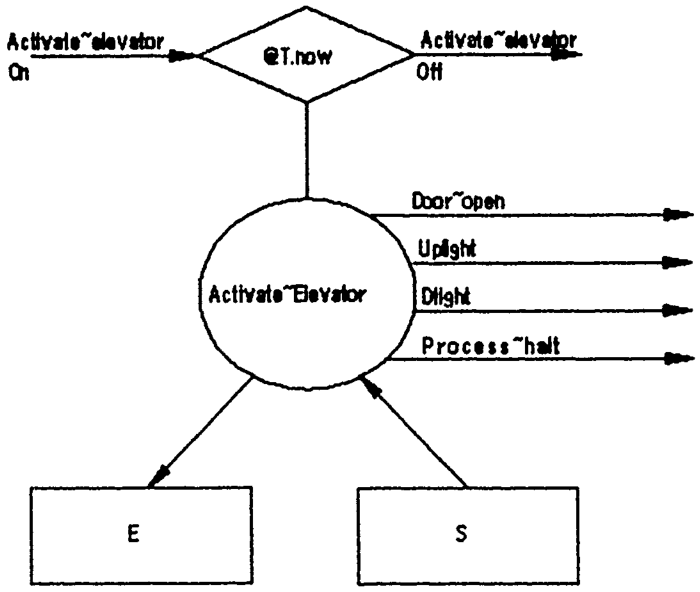


Figure C.15. Graphical Specification of Transaction 1 (Activate elevator)

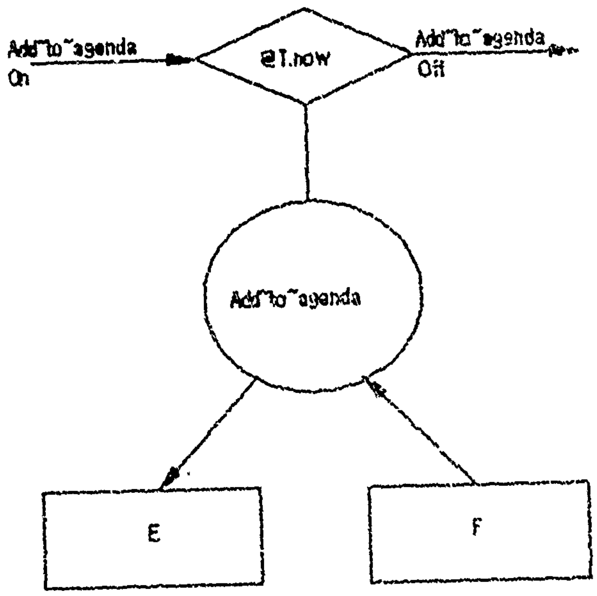


Figure C.16. Graphical Specification of Transaction 2 (Add to agenda)

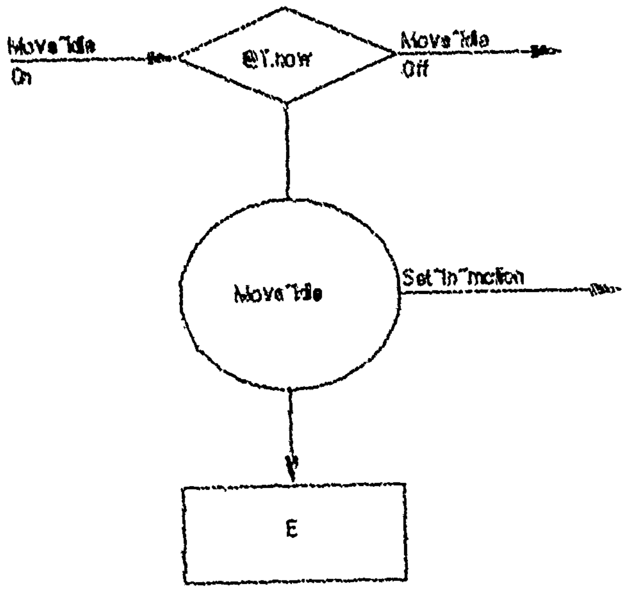


Figure C.17. Graphical Specification of Transaction 3 (Move idle)

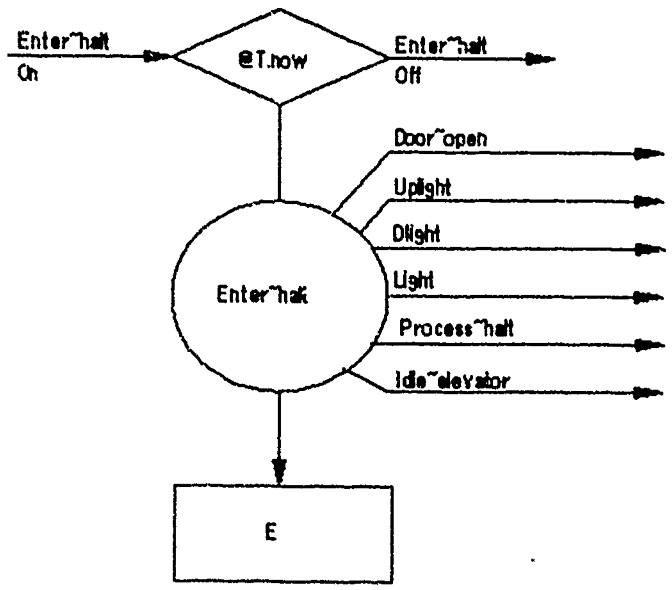


Figure C.18. Graphical Specification of Transaction 4 (Enter halt)

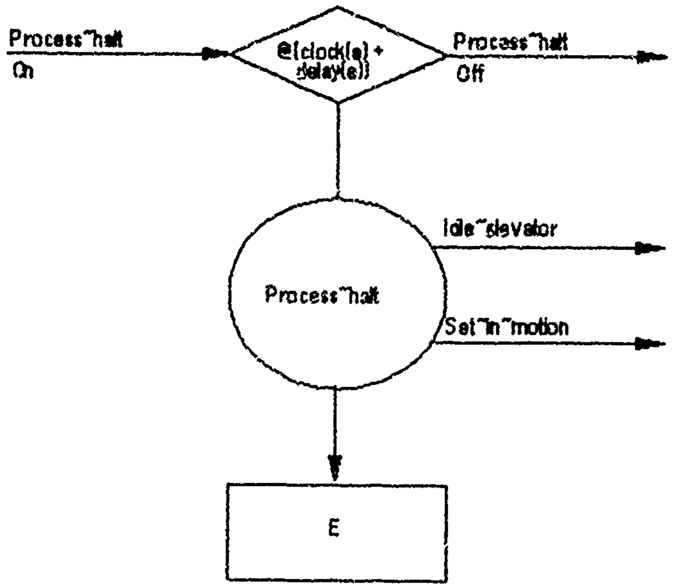


Figure C.19. Graphical Specification of Transaction 5 (Process halt)

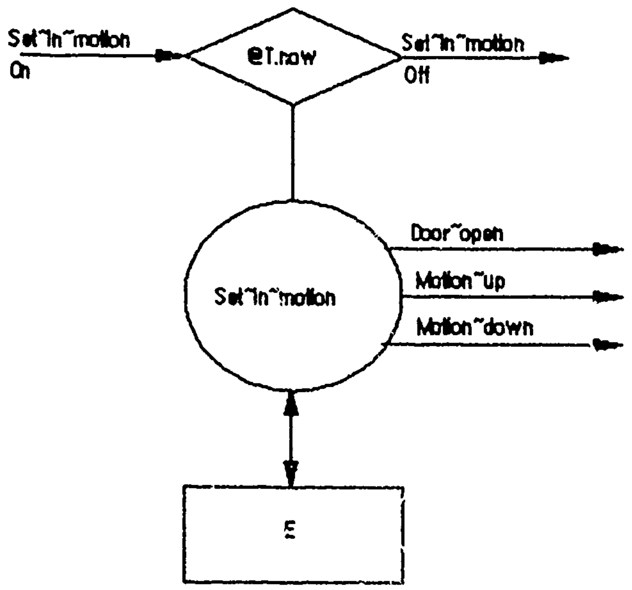


Figure C.20. Graphical Specification of Transaction 6 (Set in motion)

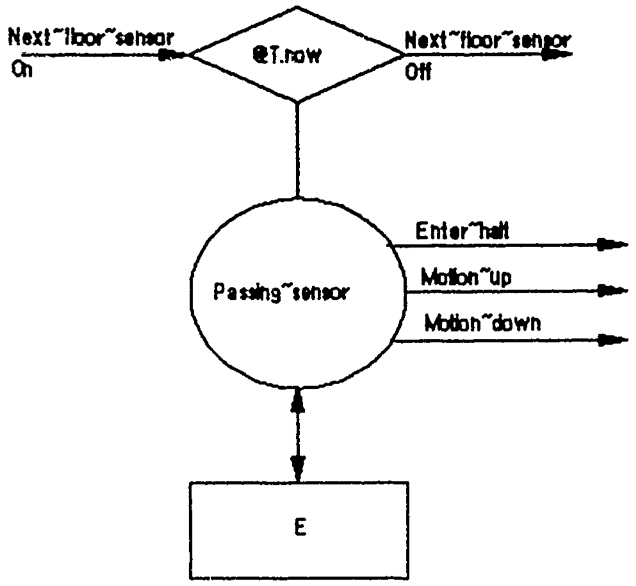


Figure C.21. Graphical Specification of Transaction 7 (Passing sensor)

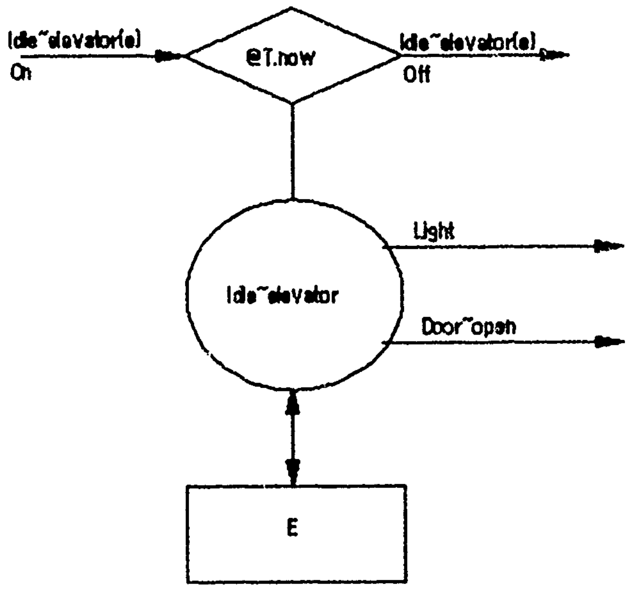


Figure C.22. Graphical Specification of Transaction 8 (Idle elevator)

Appendix D. *Refine Primitive Operations Categorized by Operand Data Type*

This appendix contains Refine's primitive operations categorized by the operand types of the operations.

- Numbers
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Integer Division
  - Integer Remainder (Modulo)
  - Integer to Real Coercion
  - Equality
  - Greater Than
  - Greater Than or Equal To
  - Less Than
  - Less Than or Equal To
- Characters
  - Equality
  - Greater Than

  - Greater Than or Equal To
  - Less Than
  - Less Than or Equal To
- Booleans
  - Negation
  - Conjunction
  - Disjunction
  - Implication
  - Ordered Conjunction
  - Ordered Disjunction
  - Universal Quantification
  - Existential Quantification
  - Nondeterministic Choice
  - Equality
- Symbols
  - Symbol To String Coercion
  - Equality
- Sets
  - Size

  - Arbitrary Element
  - Element Addition
  - Element Deletion
  - Union
  - Intersection
  - Set Difference
  - Filter by a Predicate
  - Reduction by an Operation
  - Set to Sequence Coercion
  - Empty
  - Membership
  - Subset
  - Equality

- Sequences
  - Size
  - n-th Element
  - First Element
  - Last Element
  - Subsequence
  - Following Subsequence

- Assignment of n-th Element
- Insertion as n-th Element
- Append an Element
- Prepend an Element
- Delete the n-th Element
- Reverse
- Image Under a Map
- Domain
- Range
- Concatenate
- Filter by a Predicate
- Reduction by an Operation
- Sequence to Set Coercion
- Sequence to Map Coercion
- Empty
- Membership
- Equality

• Strings

- Greater Than
- Greater Than or Equal To

- Less Than
- Less Than or Equal To

- Tuples
  - Field Retrieval
  - Field Assignment
  - Equality
- Maps
  - Size
  - Filter by a Map
  - Image
  - Domain
  - Range
  - Closure
  - Composition
  - Inverse
  - Map tɔ Binary Relation Coercion
  - Empty
  - Equality
- Binary Relations

  - Image
  - Domain
  - Range
  - Closure
  - Composition
  - Transitive Closure

- Objects
  - Creation
  - Destruction
  - Attribute Assignment
  - Attribute Retrieval

Appendix E. *Refine Primitive Operations Categorized by Operation Characteristics*

This appendix contains a listing of Refine's primitive operations categorized conceptual similarities. If multiple primitive operations are grouped together under a single operation heading, a listing of the possible operand types follows the operation heading.

- Simple Assignment
  - Numbers
  - Characters
  - Booleans
  - Symbols
  - Sets
  - Sequences
  - Strings
  - Tuples
  - Maps
  - Binary Relations
  - Objects
- Addition
  - Numbers
  - Sets (Element Addition)

  - Sequences (Concatenation)
- Subtraction
  - Numbers
  - Sets (Element Deletion)
  - Sets (Set Difference)
- Multiplication
- Division
- Integer Division
- Integer Remainder (Modulo)
- Integer to Real Coercion
- Equality
  - Numbers
  - Characters
  - Booleans
  - Symbols
  - Sets
  - Sequences
  - Tuples
  - Maps

- Greater Than
  - Numbers
  - Characters
  - Strings
- Greater Than or Equal To
  - Numbers
  - Characters
  - Strings
- Less Than
  - Numbers
  - Characters
  - Strings
- Less Than or Equal To
  - Numbers
  - Characters
  - Strings
- Symbol to String Coercion
- Negation
- Conjunction

- Ordered Conjunction
- Disjunction
- Ordered Disjunction
- Implication
- Universal Quantification
- Existential Quantification
- Size
  - Sets
  - Sequences
  - Maps
- Arbitrary Element
  - Booleans
  - Sets
- Union
- Intersection
- Filter by a Predicate
  - Sets
  - Sequences
  - Maps

- Set to Sequence Coercion
- Subset Test
- Empty Test
  - Sets
  - Sequences
  - Maps
- Membership
- n-th Element
- First Element
- Last Element
- Subsequence
- Rest of Sequence
- Assign n-th Element
- Insert at n-th Position
- Append Element
- Prepend Element
- Delete n-th Element
- Reverse
- Image

  - Sequences
  - Maps
  - Binary Relations

- Domain
  - Sequences
  - Maps
  - Binary Relations
- Range
  - Sequences
  - Maps
  - Binary Relations
- Sequence to Set Coercion
- Map Coercion
  - Sequences
  - Binary Relations
- Field Retrieval
  - Tuples
  - Objects

- Field Assignment
  - Tuples
  - Objects
- Closure
  - Maps
  - Binary Relations
- Composition
  - Maps
  - Binary Relations
- Inverse
- Map to Binary Relation Coercion
- Transitive Coercion

Appendix F. *Refine*^{TM} *Specification for the Library Control System*

This appendix contains the Refine specification for the Library Control System as described in (18).

```
#||
---------------------------------------------------------------
      Berztiss's Library System in Refine

This file implements Alf Berztiss's library system
as described in SEI document #SEI-SM-8-1.0, "SF Specification:
A Library System"

---------------------------------------------------------------
||#

% These forms should appear at the beginning of files containing
% REFINE source code

!! in-package("RU")
!! in-grammar('user)

% Establish the foundation for the library world

var LIBRARY-WORLD-OBJECT: object-class subtype-of user-object

% Now define the objects in the library's world
var BOOK: object-class subtype-of library-world-object
var USER: object-class subtype-of library-world-object

% Require that every non-identical instance of any of the classes have
% a unique name
form DECLARE-LIBRARY-UNIQUE-NAMES-CLASSES
   unique-names-class ( 'book, true ) &
   unique-names-class ( 'user, true )

% Define the attributes of a book
var BOOK-OUT: map( book, boolean ) = {||}
var ON-SHELF: map( book, boolean ) = {||}
var TITLE-OF-BOOK: map( book, string ) = {||}
var AUTHOR-OF-BOOK: map( book, string ) = {||}
var SUBJECT-OF-BOOK: map( book, set( string ) ) = {||}
var LAST-CHECKED-OUT-BY: map( book, string ) = {||}

% Define the attributes of a user
var CUSTOMER: map( user, boolean ) = {||}
var STAFF: map( user, boolean ) = {||}
var USER-NAME: map( user, string ) = {||}

% Library maintenance operations
rule Add-Book-To-Library( author : string,
                          title : string,
                          subject : set( string ) )
    % If there's not already a book with this title
    % (to guarantee unique titles)
    empty({ b | (b: book) book(b) & title-of-book(b) = title }) -->
            % Create a new book object
            let ( new-book: book = make-object( 'book ) )
              % And set the attributes of the book
                author-of-book(new-book) <- author;
                title-of-book(new-book) <- title;
                subject-of-book(new-book) <- subject;
                on-shelf(new-book) <- true;
                book-out(new-book) <- false

% Assumes that every book has a unique title
rule Remove-Book-From-Library( book-title : string )
   % If a book of this title exists
   ~empty({ b | (b: book) book(b) & title-of-book(b) = book-title }) -->
    % Find the book with this title
    let ( book-to-delete: book = arb({ b | (b:book) book(b) &
                                    title-of-book(b) = book-title }))
      % And delete it
      erase-kb-object( book-to-delete )

rule Add-User( users-name : string, on-staff : boolean )
   % If there does not already exist a user with this name
   % (to guarantee unique names)
   empty({ u | (u: user) user(u) & user-name(u) = users-name }) -->
            % Then create a new user
            let ( new-user: user = make-object( 'user ) )
              % And set the user's attributes
              set-attrs( new-user, 'user-name, users-name,
                                   'staff, on-staff,
                                   'customer, ~on-staff )
```

```
rule Remove-User( users-name : string )
   % If there really does exist a user with this name
   ~empty({ u | (u: user) user(u) & user-name(u) = users-name }) -->
       % Then find the user with this name
       let( user-to-delete: user =
         arb({ u | (u: user) user(u) & user-name(u) = users-name}))
       % And delete that user
       erase-kb-object( user-to-delete )

rule Check-Out-Book( whos-asking: string,
                     users-name: string,
                     which-book: string )
   % If there is a user with the name of whos-asking
   ~empty({ u | (u: user) user(u) & user-name(u) = whos-asking }) &
   % And if there is a user with the name of users-name
   ~empty({ u | (u: user) user(u) & user-name(u) = users-name }) &
   % And if there is a book of the title which-book
   ~empty({ b | (b: book) book(b) & title-of-book(b) = which-book }) -->
      % Then if whos-asking is on the staff
      staff( arb({u | (u: user) user(u) & user-name(u) = whos-asking}) ) &
      % And if the requested book is on the shelf
      on-shelf( arb({ b | (b: book) book(b) &
                              title-of-book(b) = which-book })) -->
      % Then find the book with the title of which book
      let( book-to-check-out: book = arb({ b | (b: book) book(b) &
                                      title-of-book(b) = which-book}))
        % And check the book out
        set-attrs( book-to-check-out, 'on-shelf, false,
                                      'book-out, true,
                                      'last-checked-out-by, users-name )

rule Return-Book( whos-asking: string, which-book: string )
   % If there is a user with the name of whos-asking
   ~empty({ u | (u: user) user(u) & user-name(u) = whos-asking }) &
   % And if there is a book with the title of which-book
   ~empty({ b | (b: book) book(b) & title-of-book(b) = which-book }) -->
      % Then if whos-asking is on the staff
      staff( arb({ u | (u: user) user(u) & user-name(u) = whos-asking })) &
      % And if the book is checked out
      book-out( arb({ b | (b: book) book(b) &
                          title-of-book(b) = which-book })) -->
         % Then find the book with the title which-book
         let ( book-to-be-returned: book = arb({ b | (b: book) book(b) &
                          title-of-book(b) = which-book }))
```

```
            % And check the book back in
            set-attrs( book-to-be-returned,
                           'on-shelf, true,
                           'book-out, false )

% Library printing functions

function PRINT-BOOK-SET( set-of-books: set(book) ) =
    if empty(set-of-books)
      then format( true, "This set of books is empty")
      else enumerate b over set-of-books do
        format( true, title-of-book(b) )

function PRINT-USER-SET( set-of-users: set(user) ) =
   if empty(set-of-users)
     then format( true, "This set of users is empty")
     else enumerate u over set-of-users do
       format( true, user-name(u) )

% Library queries
rule Books-On-Subject( which-subject: string )
   % Return all books with subjects matching
   % which-subject
   true --> print-book-set({ b | ( b: book) book( b ) &
                              which-subject in subject-of-book( b ) })

function Books-By-Author( which-author: string ) =
   % Return all books with an author matching
   % which-author
   print-book-set( { b | ( b: book ) book( b ) &
                                    author-of-book( b ) = which-author })

function Books-Checked-Out-By-User( whos-asking: string
                 users-name: string ) =
   % If there is a user with the name of whos-asking
   if ~empty({ u | (u: user) user(u) & user-name(u) = whos-asking }) &
   % And if there is a user with the name of users-name
   ~empty({ u | (u: user) user(u) & user-name(u) = users-name }) &
   % And if whos-asking is on the staff
   ( staff( arb({ u | (u: user) user(u) & user-name(u) = whos-asking })) or
   % Or if the user is asking about him or herself
   whos-asking = users-name )
      % Then return all books that the user currently has checked out
      then print-book-set( { b | (b: book) book(b) & book-out(b) &
              users-name = last-checked-out-by(b) })
      else format( true, "You are not authorized access to that information")


form CREATE-LIBRARY-USERS
   add-user( "gene", true );
   add-user( "mark", false );
   add-user( "rob", false )

form CREATE-LIBRARY-BOOKS
   add-book-to-library( "ritchie", "the c programming language",
       {"programming", "computer science"});
   add-book-to-library( "silberschatz", "operating system concepts",
       {"operating systems", "computer science"});
   add-book-to-library( "korth", "database system concepts",
       {"database", "computer science"})
```

Appendix G. *Graphical Specification for the Library Control System*

This appendix contains the formal specification for the Library Control System as described in (18) using the Refine-based graphicai specification language developed in Chapter 4.

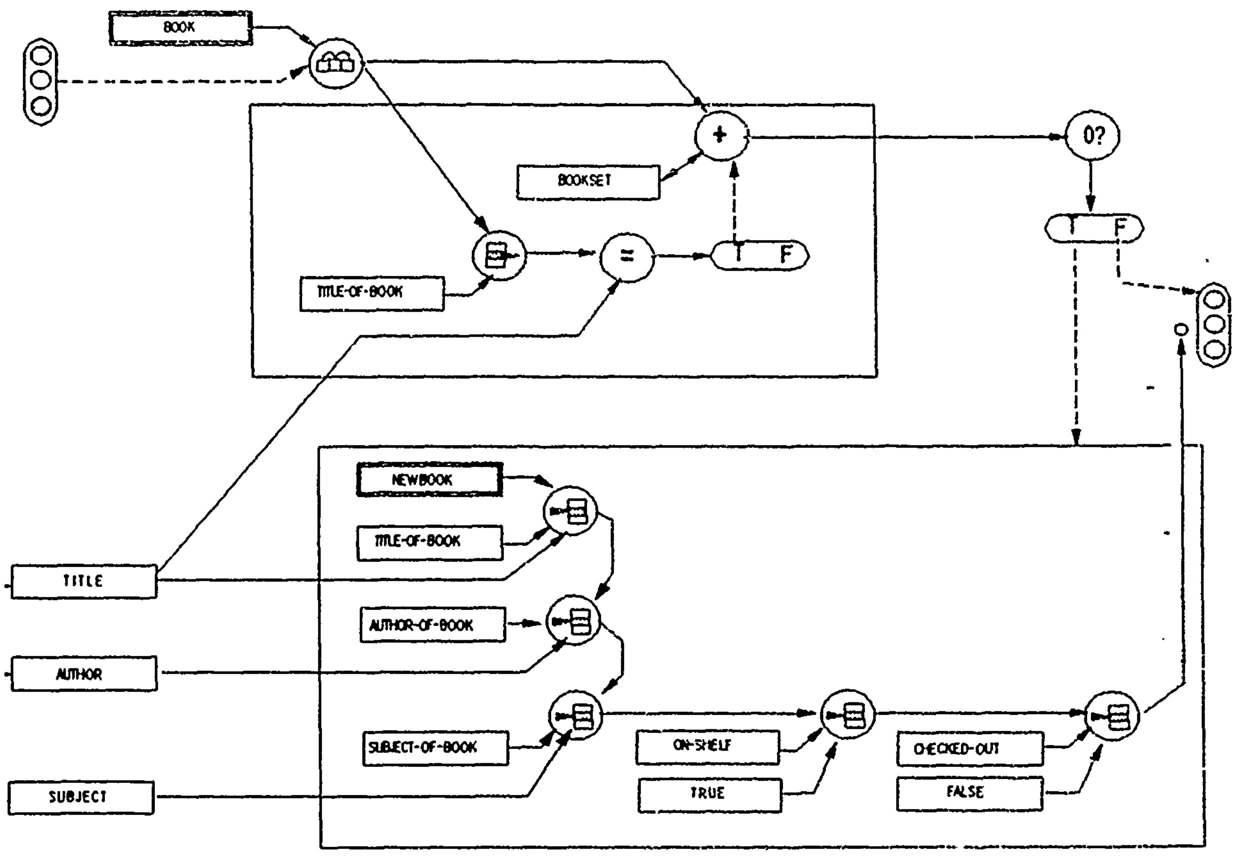


Figure G.1. Graphical Formulation of the Add-Book Function

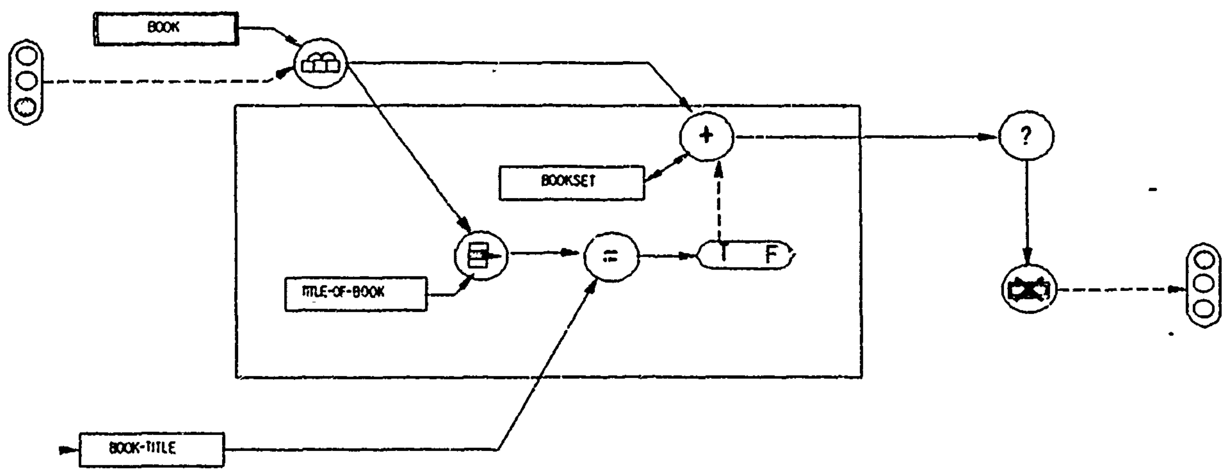


Figure G.2. Graphical Formulation of the Remove-Book Function

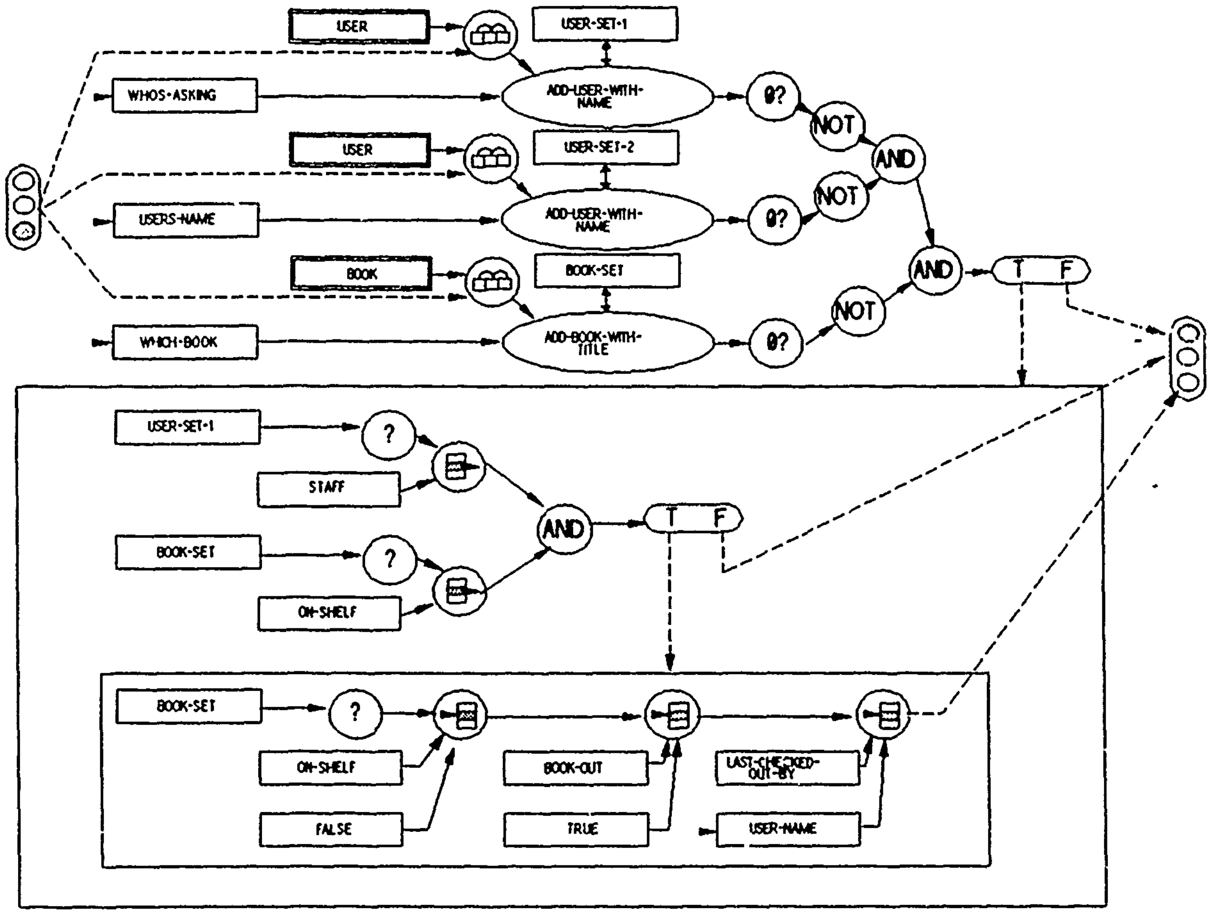


Figure G.3. Graphical Formulation of the Check-Out-Book Function

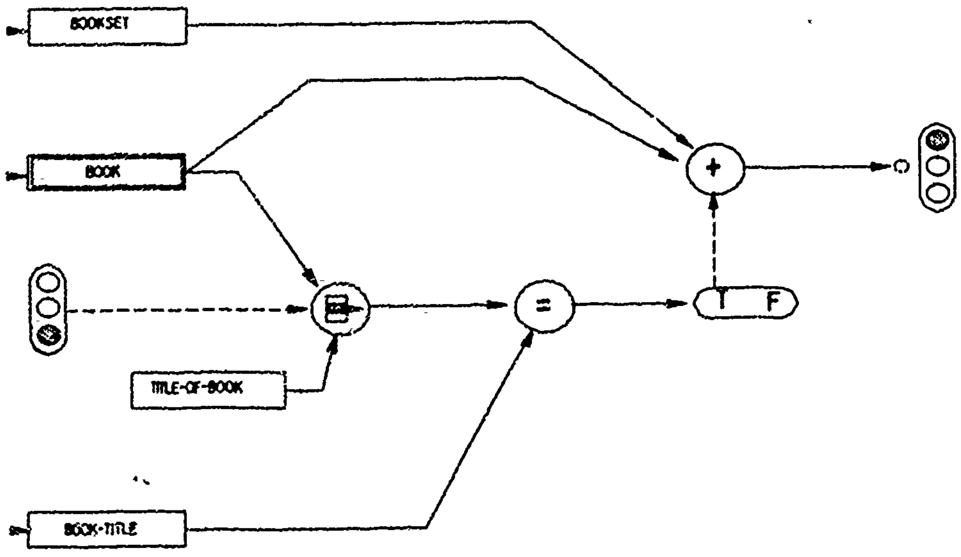


Figure G.4. Graphical Formulation of the Add-Book-With-Title Function

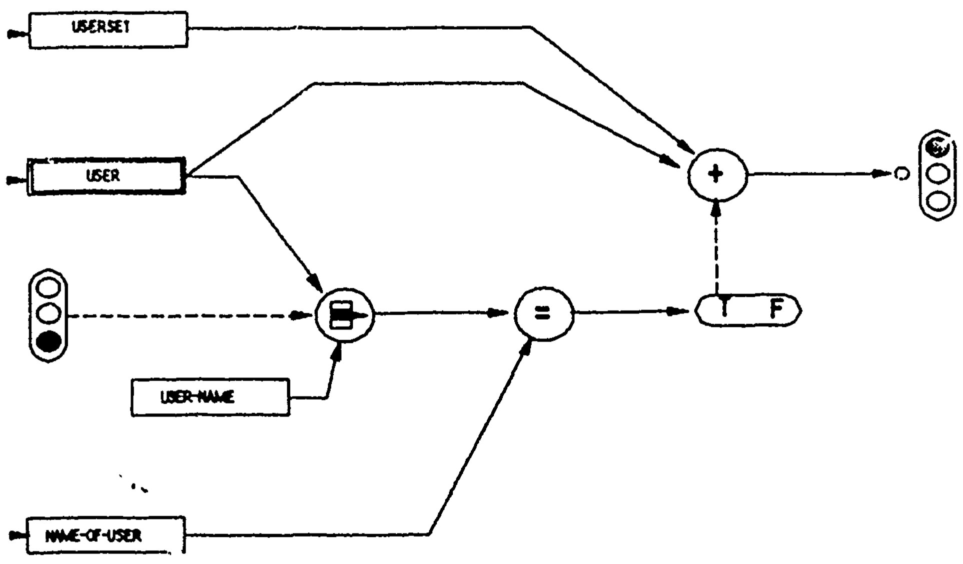


Figure G.5. Graphical Formulation of the Add-User-With-Name Function

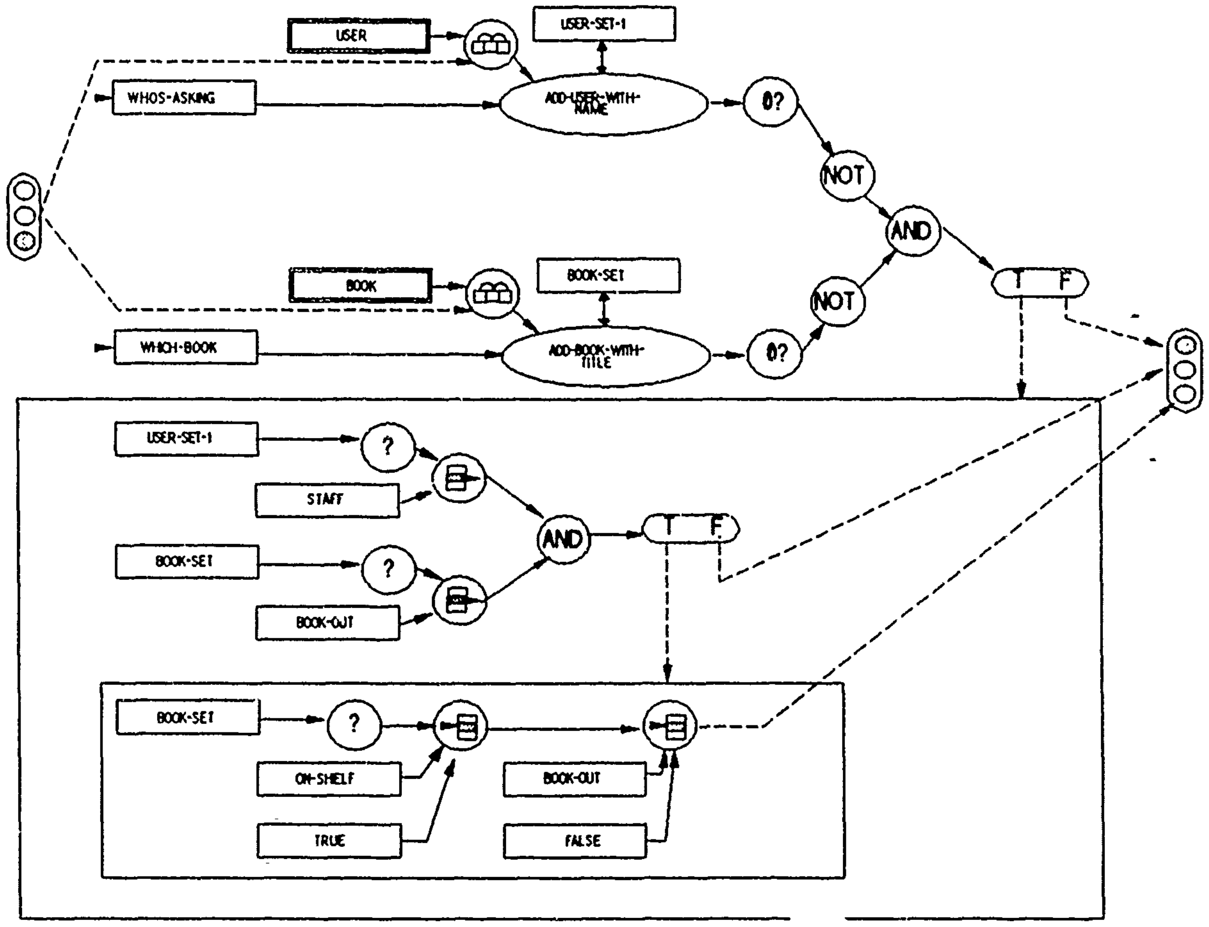


Figure G.6. Graphical Formulation of the Return-Book Function

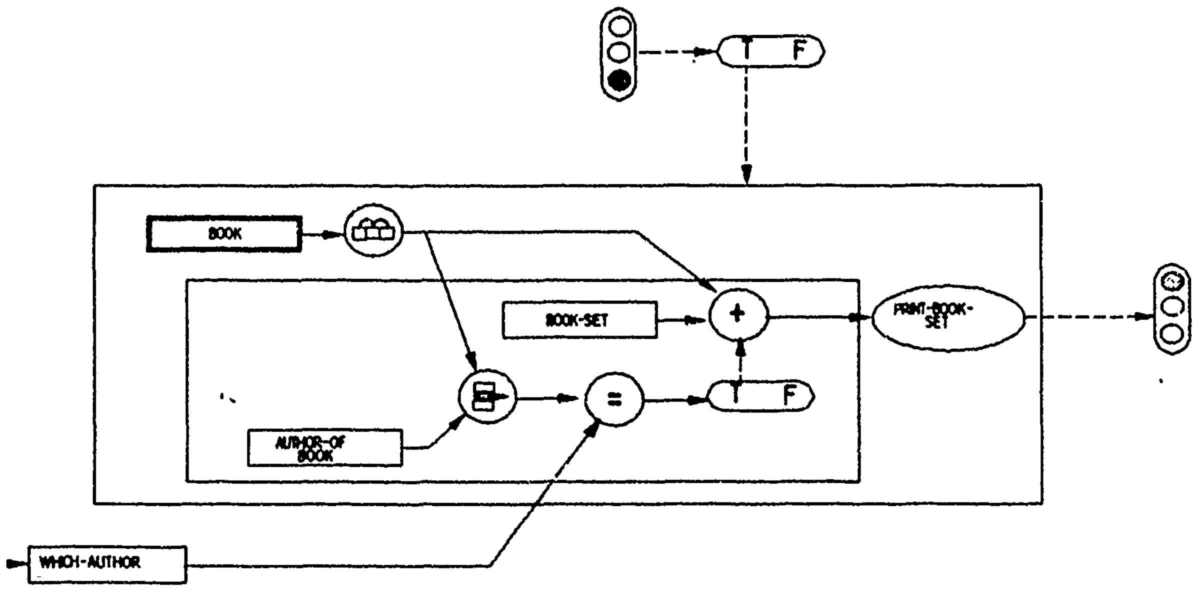


Figure G.7. Graphical Formulation of the Books-by-Author Function

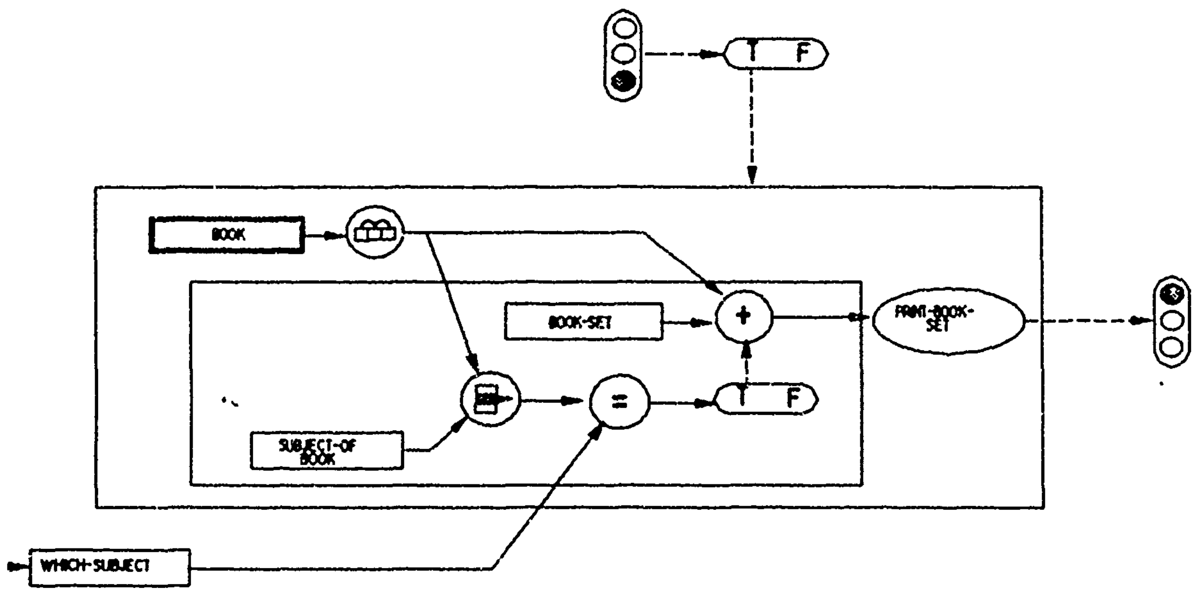


Figure G.8. Graphical Formulation of the Books-on-Subject Function

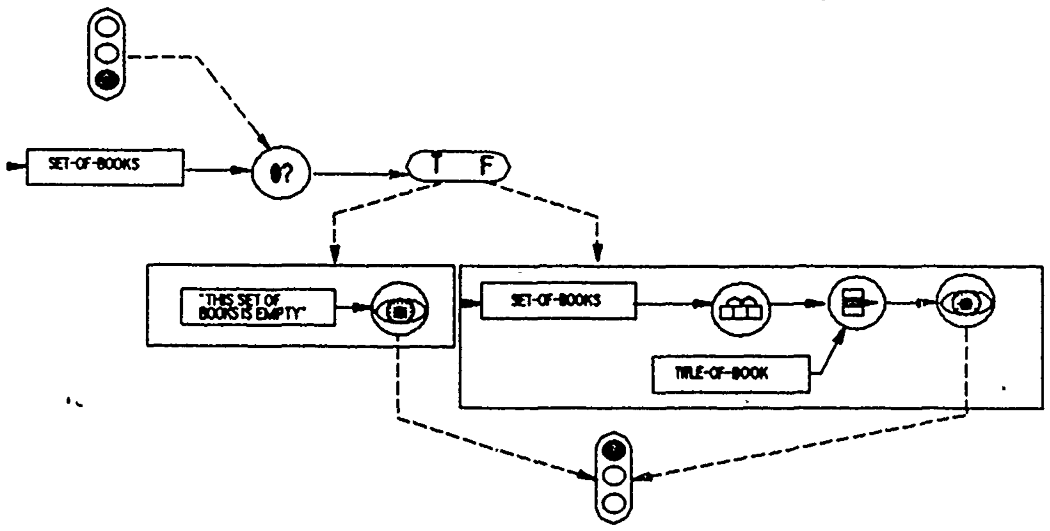


Figure G.9. Graphical Formulation of the Print-Book-Set Function

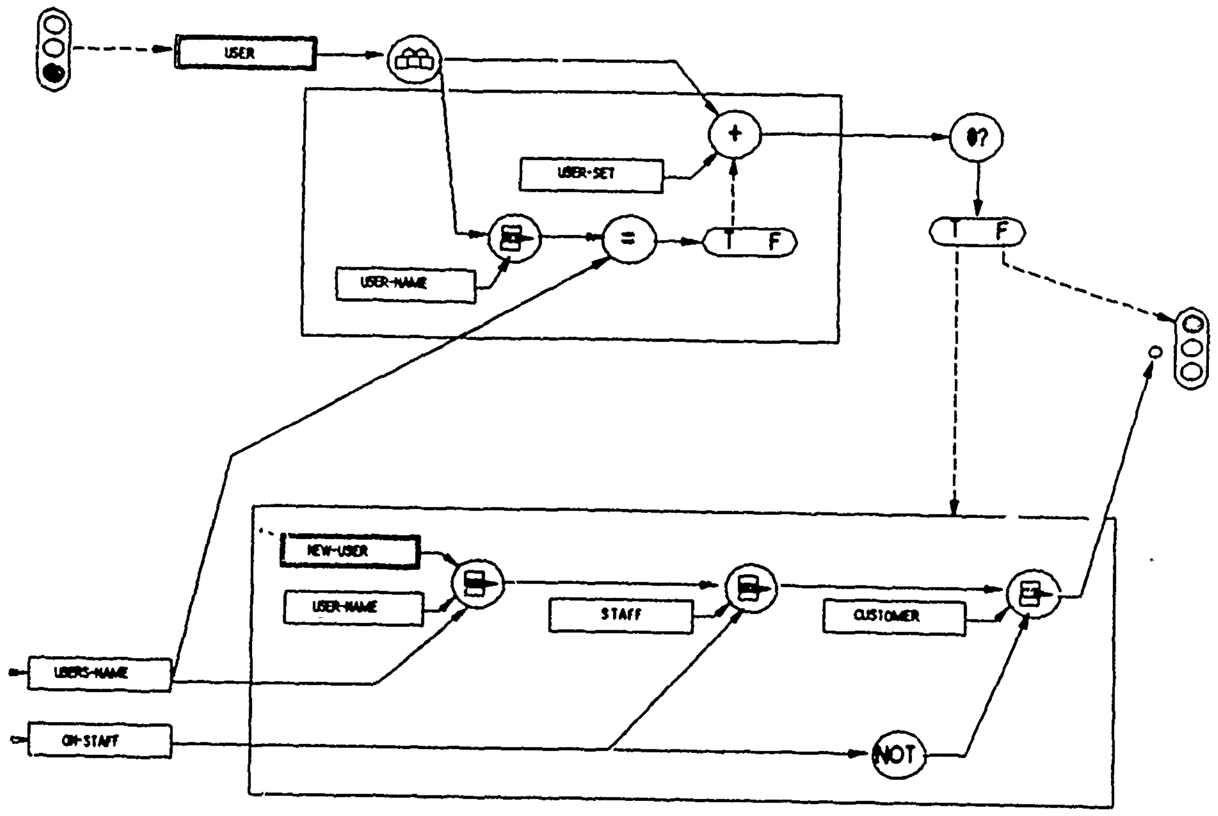


Figure G.10. Graphical Formulation of the Add-User Function

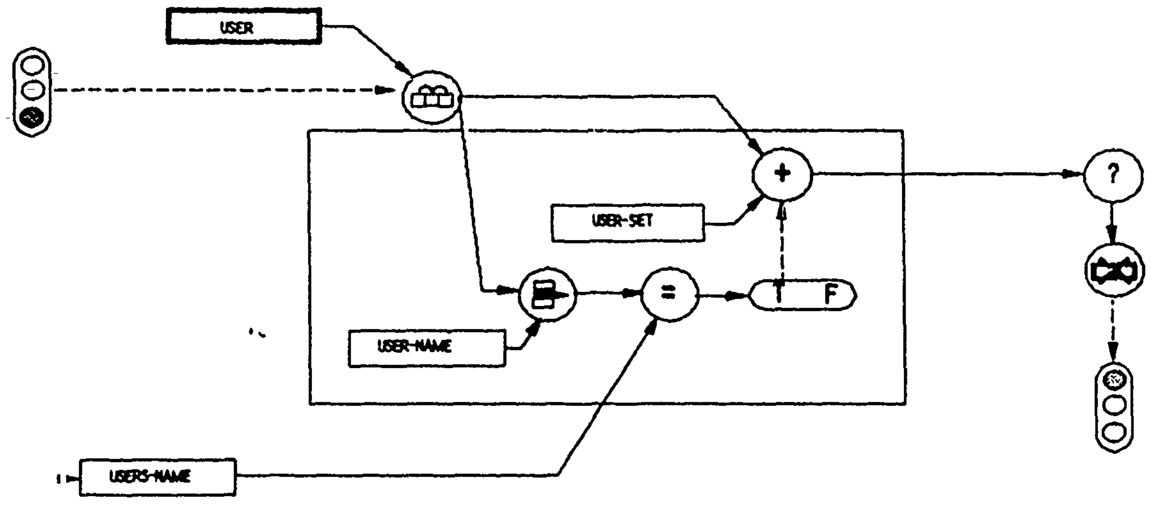


Figure G.11. Graphical Formulation of the Remove-User Function

Appendix H. *Refine^{TM} Specification for the Elevator System*

This appendix contains the Refine specification for the Elevator Control System as described in (11).

```
#||
----------------------------------------------------------------
             Berztiss's Elevator System in Refine
This file implements Alf Berztiss's elevator system
as described in SEI document #SEI-SM-8-1.0, "SF Specification:
An Elevator Controller"

----------------------------------------------------------------
||#

% These forms should appear at the beginning of files containing
% REFINE source code

!! in-package("RU")
!! in-grammar('user)


% Establish the foundation for the elevator world
var ELEVATOR-WORLD-OBJECT: object-class subtype-of user-object

% Now define the objects in the elevator's world
var ELEVATOR: object-class subtype-of elevator-world-object

% Require that every non-identical instance of any of the classes have
% a unique name
form DECLARE-ELEVATOR-UNIQUE-NAMES-CLASSES
   unique-names-class( 'elevator, true )

% Declare the attributes of an elevator
var STATE: map( elevator, symbol ) = {||}
var CURRENT-FLOOR: map( elevator, integer ) = {||}
var SCHEDULE: map( elevator, set( integer ) ) = {||}
var ALARM: map( elevator, boolean ) = {||}
var RESET: map( elevator, boolean ) = {||}
```

```
% Establish the data structures for the elevator
% driver
var event: object-class subtype-of elevator-world-object
var *events*: set( event ) = {}

var EVENT-TIME: map( event, integer ) = {||}
var EVENT-TYPE: map( event, symbol ) = {||}
var ELEVATOR-ID: map( event, symbol ) = {||}
var FLOOR-ID: map( event, symbol ) = {||}
var WHICH-FLOOR: map( event, integer ) = {||}
var DIRECTION: map( event, symbol ) = {||}

var clock: object-class subtype-of elevator-world-object
constant wall-clock: clock = make-object( 'clock )
var CLOCK-TIME: map( clock, integer ) = {| wall-clock -> 0 |}

% Establish the rules for transitioning between elevator
% states
% The states for the elevator system are described in
% Berztiss, p. 35

% Move from the idle state to the uphalt state
rule Start-Up( the-elevator : elevator )
   ~empty({ f | (f : integer) f in schedule( the-elevator ) &
    f > current-floor( the-elevator ) }) &
   state( the-elevator ) = 'idle -->
   state( the-elevator ) <- 'uphalt

% Move from the uphalt state to the up state
rule Move-Up( the-elevator: elevator )
   ~empty({ f | (f: integer) f in schedule( the-elevator ) &
    f > current-floor( the-elevator ) }) &
   state( the-elevator ) = 'uphalt -->
     ( state( the-elevator ) <- 'up ) &
     ( add-event( clock-time( wall-clock ) + 2,
  'floor-reached,
  name( the-elevator ),
  0, 'dummy ) )

% Move from the uphalt state to the idle state
rule Up-To-Idle( the-elevator: elevator )
   empty({ f | (f: integer) f in schedule( the-elevator ) &
   f > current-floor( the-elevator ) }) &
   state( the-elevator ) = 'uphalt -->
   state( the-elevator ) <- 'idle
```

```
% Move from the up state to the uphalt state
rule Up-To-Halt( the-elevator: elevator )
   ~empty({ f | (f: integer) f in schedule( the-elevator ) &
   f = current-floor( the-elevator ) }) &
   state( the-elevator ) = 'up -->
   ( state( the-elevator ) <- 'uphalt ) &
   ( schedule( the-elevator ) <- schedule( the-elevator ) less
                                 current-floor( the-elevator ) )

% Move from either the up or uphalt states to the upstop state
rule Move-To-Upstop( the-elevator: elevator )
   alarm( the-elevator ) &
   ( ( state( the-elevator ) = 'uphalt ) or
     ( state( the-elevator ) = 'up ) ) -->
   state( the-elevator ) <- 'upstop

% Move from the upstop state to the uphalt state
rule Up-Reset( the-elevator: elevator )
   reset( the-elevator ) &
   state( the-elevator ) = 'upstop -->
   ( alarm( the-elevator ) <- false ) &
   ( reset( the-elevator ) <- false ) &
   ( state( the-elevator ) <- 'uphalt )

% Move from the idle state to the dhalt state
rule Start-Down( the-elevator: elevator )
   ~empty({ f | (f: integer) f in schedule( the-elevator ) &
    f < current-floor( the-elevator )}) &
   state( the-elevator ) = 'idle -->
   state( the-elevator ) <- 'dhalt

% Move from the dhalt state to the idle state
rule Down-To-Idle( the-elevator: elevator )
   empty({ f | (f: integer ) f in schedule( the-elevator ) &
   f < current-floor( the-elevator )}) &
   state( the-elevator ) = 'dhalt -->
   state( the-elevator ) <- 'idle

% Move from the dhalt state to the down state
rule Move-Down( the-elevator: elevator )
   ~empty({ f | (f: integer) f in schedule( the-elevator ) &
    f < current-floor( the-elevator )}) &
   state( the-elevator ) = 'dhalt -->
     ( state( the-elevator ) <- 'down ) &
     ( add-event( clock-time( wall-clock ) + 2,
  'floor-reached,
  name( the-elevator ),
  0, 'dummy ) )

% Move from the down state to the dhalt state
rule Down-To-Halt( the-elevator: elevator )
   ~empty({ f | (f : integer) f in schedule( the-elevator ) &
    f = current-floor( the-elevator )}) &
   state( the-elevator ) = 'down -->
   ( state( the-elevator ) <- 'dhalt ) &
   ( schedule( the-elevator ) <- schedule( the-elevator ) less
                                 current-floor( the-elevator ) )

% Move from either the down or dhalt states to the dstop state
rule Move-To-Dstop( the-elevator: elevator )
   alarm( the-elevator ) &
   (( state( the-elevator ) = 'down ) or
    ( state( the-elevator ) = 'dhalt )) -->
   state( the-elevator ) <- 'dstop

% Move from the dstop state to the dhalt state
rule Down-Reset( the-elevator: elevator )
   reset( the-elevator ) &
   state( the-elevator ) = 'dstop -->
   ( reset( the-elevator ) <- false  ) &
   ( alarm( the-elevator ) <- false ) &
   ( state( the-elevator ) <- 'dhalt )

% Now establish the mechanism to move the elevator from state
% state
rule Check-State( the-elevator: elevator )
   true -->
   ( state( the-elevator ) = 'idle -->
       ( start-up( the-elevator ) ) &
       ( start-down( the-elevator ) ) ) &
   ( state( the-elevator ) = 'uphalt -->
       ( move-up( the-elevator ) ) &
       ( up-to-idle( the-elevator ) ) ) &
   ( state( the-elevator ) = 'up -->
       ( up-to-halt( the-elevator ) ) &
       ( move-to-upstop( the-elevator ) ) ) &
   ( state( the-elevator ) = 'upstop -->
       ( up-reset( the-elevator ) ) ) &
   ( state( the-elevator ) = 'dhalt -->
        ( move-down( the-elevator ) ) &
        ( down-to-idle( the-elevator ) ) ) &
    ( state( the-elevator ) = 'down -->
        ( down-to-halt( the-elevator ) ) &
        ( move-to-dstop( the-elevator ) ) ) &
    ( state( the-elevator ) = 'dstop -->
        ( down-reset( the-elevator ) ) )

% Establish a method of creating and initializing an elevator
rule Add-Elevator( elevator-id: symbol )
   true -->
     ( let ( new-elevator: elevator = make-object( 'elevator ) )
         name( new-elevator ) <- elevator-id;
         state( new-elevator ) <- 'idle;
         current-floor( new-elevator ) <- 1;
         schedule( new-elevator ) <- {};
         alarm( new-elevator ) <- false;
         reset( new-elevator ) <- false )

% Establish a method of adding events to the system event list
rule Add-Event( the-time: integer,
        the-type: symbol,
        the-elevator: symbol,
        the-button: integer,
        the-direction: symbol )
   true -->
       let ( new-event: event = make-object( 'event ) )
         event-time( new-event ) <- the-time;
         event-type( new-event ) <- the-type;
         elevator-id( new-event ) <- the-elevator;
         which-floor( new-event ) <- the-button;
         direction( new-event ) <- the-direction;
         *events* <- *events* with new-event

rule Schedule-Summons( the-floor: integer,
       the-direction: symbol )
   true -->
     ( the-direction = 'up -->
        let (candidates: set( elevator ) = { e | (e: elevator) elevator(e) &
     (( state(e) = 'up &
         the-floor > current-floor(e) )
                                                          or state(e) = 'idle ) })
          if ~empty( candidates ) then
            let (min: elevator = arb(candidates))
              candidates <- candidates less min;
              ( enumerate e: elevator over candidates do
                if (the-floor - current-floor(e) <
    the-floor - current-floor(min)) then
                  min <- e );
              schedule(min) <- schedule(min) with the-floor
          else
            let ( tempset: set(elevator) = { e | (e:elevator) elevator(e) } )
              let ( min: elevator = arb( tempset ) )
                tempset <- tempset less min;
                ( enumerate e: elevator over candidates do
           if size(schedule(e)) < size(schedule(min)) then
             min <- e );
                schedule(min) <- schedule(min) with the-floor ) &
   ( the-direction = 'down -->
       let (candidates: set( elevator ) = { e | (e: elevator) elevator(e) &
    (( state(e) = 'down &
        the-floor < current-floor(e) )
                                            or state(e) = 'idle ) } )
         if ~empty( candidates ) then
   let( min: elevator = arb( candidates ) )
             candidates <- candidates less min;
             (enumerate e: elevator over candidates do
                if (current-floor(e) - the-floor <
    current-floor(min) - the-floor) then
                  min <- e);
             schedule(min) <- schedule(min) with the-floor
         else
           let( tempset: set(elevator) = { e | (e: elevator) elevator(e) } )
     let ( min: elevator = arb( tempset ) )
               tempset <- tempset less min;
               (enumerate e: elevator over tempset do
          if (size(schedule(e)) < size(schedule(min))) then
            min <- e);
               schedule(min) <- schedule(min) with the-floor )


function Handle-Events( event-set: set(event) ) =
     while ~empty( event-set ) do
       let ( the-event: event = arb( event-set ) )
         ( event-type( the-event ) = 'dest-pressed -->
           let ( the-elevator: elevator =
   arb({e | (e: elevator) elevator(e) &
        name(e) = elevator-id( the-event )}))
             schedule( the-elevator ) <- schedule( the-elevator ) with
                                 which-floor( the-event ) );
         ( event-type( the-event ) = 'floor-reached -->
   let ( the-elevator: elevator =
   arb({e | (e: elevator) elevator(e) &
name(e) = elevator-id( the-event)}))
     ( if state( the-elevator ) = 'down then
         current-floor( the-elevator ) <-
           current-floor( the-elevator ) - 1
       elseif state( the-elevator ) = 'up then
         current-floor( the-elevator ) <-
           current-floor( the-elevator ) + 1 ));
         ( event-type( the-event ) = 'summons-pressed -->
    schedule-summons( which-floor( the-event ),
      direction( the-event ) ) );
         ( event-type( the-event ) = 'alarm-rung -->
    let( the-elevator: elevator = arb({ e | (e: elevator) elevator(e) &
   name(e) = elevator-id(the-event)}))
      alarm( the-elevator ) <- true );
         ( event-type( the-event ) = 'elevator-reset -->
    let (the-elevator: elevator = arb({ e | (e: elevator) elevator(e) &
   name(e) = elevator-id(the-event)}))
      reset( the-elevator ) <- true );
         event-set <- event-set less the-event

function Schedule-New-Events() =
   let ( moving-set: set( elevator ) = filter( lambda( x: elevator )
       ~ ex( v: event )
      ( v in *events* &
       elevator-id( v ) = name(x) &
       event-type(v) = 'floor-reached),
      { e | (e: elevator)
       elevator(e) &
       (state(e) = 'up or
state(e) = 'down)}))
     enumerate e: elevator over moving-set do
       add-event( clock-time( wall-clock ) + 2,
  'floor-reached, name(e),
  0, 'dummy )

function Do-Elevator() =
     % While there's still things left to do...
     while ~empty( *events* ) or ~empty({ e | (e: elevator) elevator(e) &
 (state(e) ~= 'idle or
 ~empty( schedule(e) ))}) do
       % Find the events happening now
       ( let ( current-events: set(event) = filter( lambda( x: event )
event-time( x ) <=
clock-time( wall-clock ),
*events* ) )
        % Handle all current events
        handle-events( current-events );
        % Check the new states of all of the elevators
        ( enumerate e: elevator over
     { elev | (elev: elevator) elevator( elev ) } do
          check-state(e) ) );
        % Remove the old events from the current event list
        *events* <- setdiff( *events*, filter( lambda( x: event )
      event-time( x ) <=
      clock-time( wall-clock ),
      *events* ) );
        schedule-new-events();
        % Increment the clock
        clock-time( wall-clock ) <- clock-time( wall-clock ) + 1;
        print("==============================================");
        print("Elevator states at time");
        print(clock-time( wall-clock ) );
        print("==============================================");
        (enumerate e: elevator over { elev | (elev: elevator) elevator(elev)}
        do
          print( name(e));
          print("Current floor");
  print( current-floor(e));
  print("Current state");
  print( state(e) );
  print("Current schedule");
  print( schedule(e));
          print("=============================================="));
        print("==============================================");
        print("Events pending at time");
        print(clock-time( wall-clock));
        print("==============================================");
        (enumerate v: event over { ev | (ev : event) event(ev) &
   ev in *events* } do
           print( "Event time" );
   print( event-time( v ) );
   print( "Event type" );
   print( event-type( v ) ):
   print("Elevator ");
   print( elevator-id( v ) );
   print( "Floor" );
   print( which-floor( v ) );
```

```
    print( "Direction" );
    print( direction( v ) );
    print("==============================================") );
            print("///////////////////////////////////////////////")

form CREATE-ELEVATORS
   add-elevator( 'elevator-1 ) &
   add-elevator( 'elevator-2 )

form CREATE-INITIAL-EVENTS
   add-event( 1, 'dest-pressed, 'elevator-1, 3, 'dummy ) &
   add-event( 2, 'summons-pressed, 'dummy, 2, 'up ) &
   add-event( 3, 'summons-pressed, 'dummy, 4, 'down ) &
   add-event( 5, 'dest-pressed, 'elevator-2, 1, 'dummy )
```

Appendix I. *Graphical Specification of Elevator System*

This appendix contains the formal specification for the Elevator Control System as described in (11) using the Refine-based graphical specification language developed in Chapter 4.

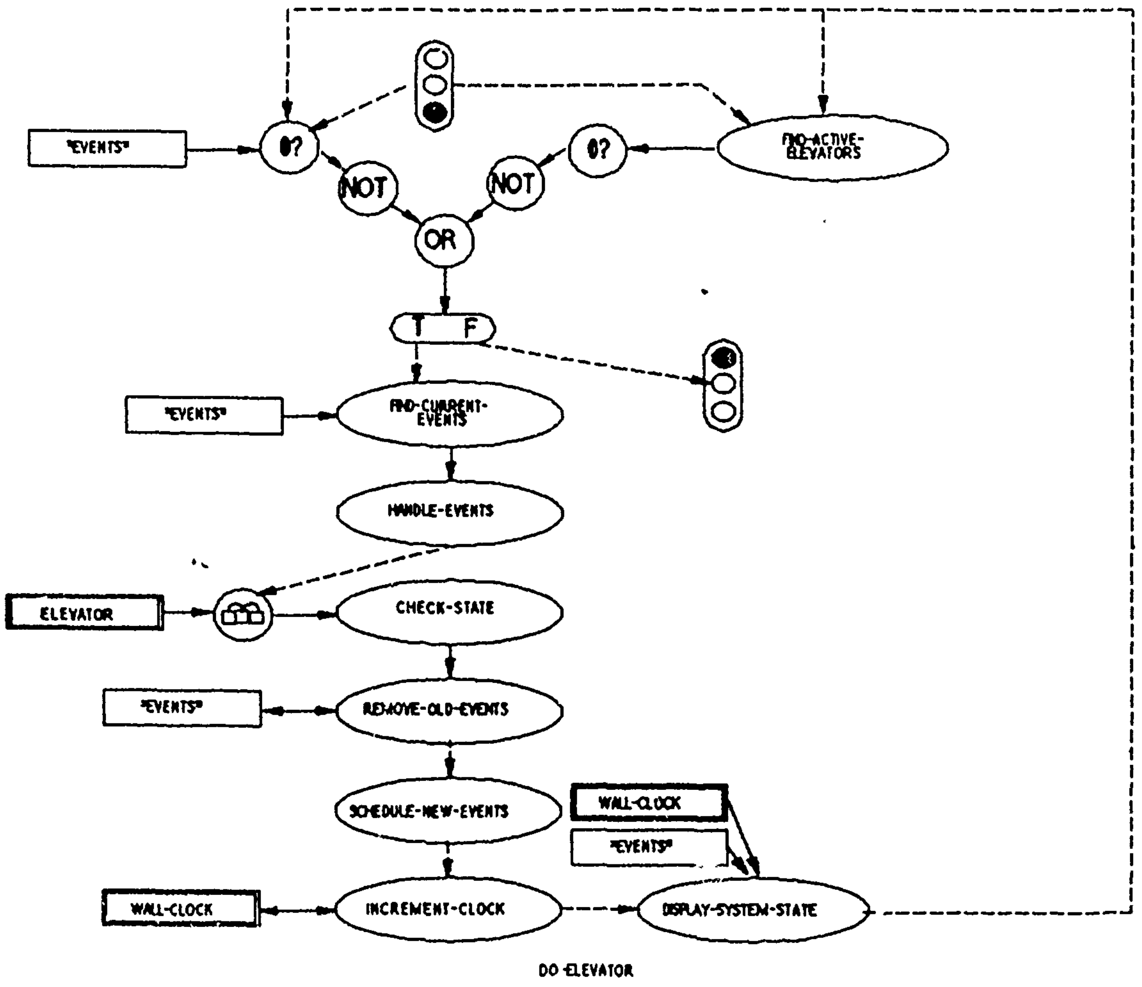


Figure I.1. Graphical Formulation of the Do-Elevator Function

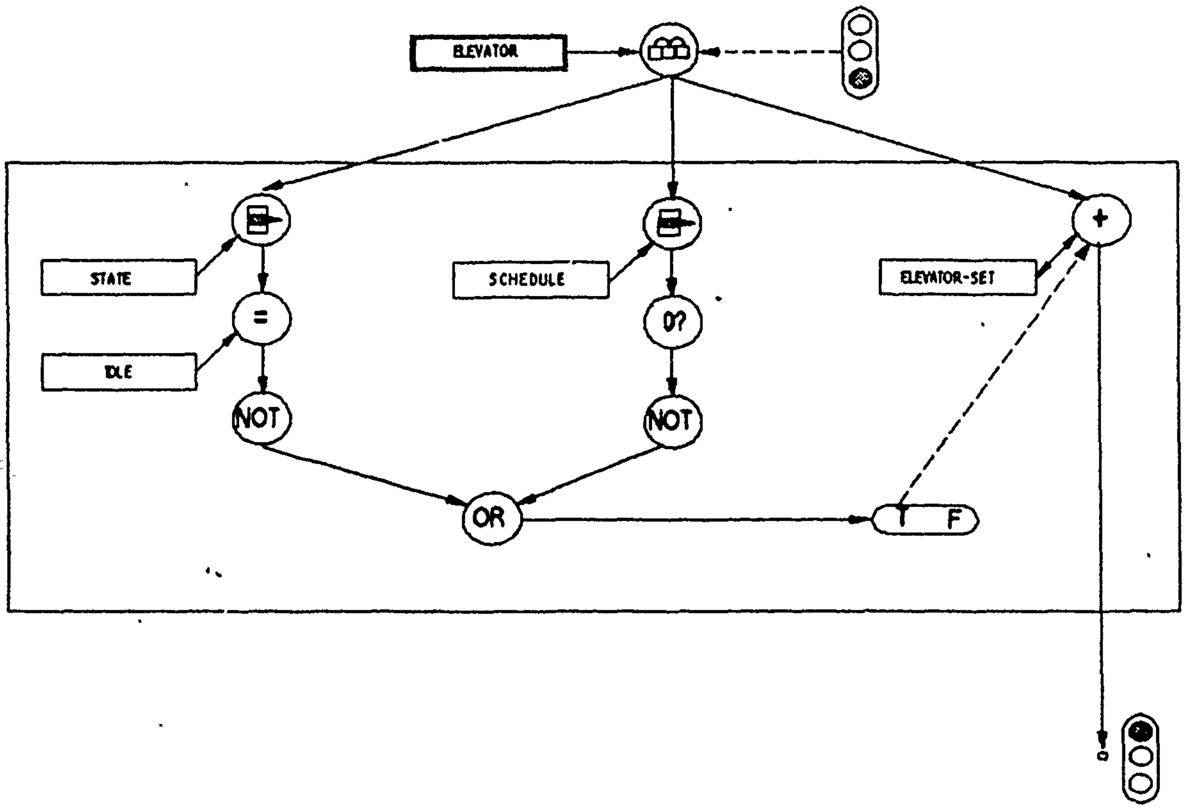


Figure I.2. Graphical Formulation of the Find-Active-Elevators Function

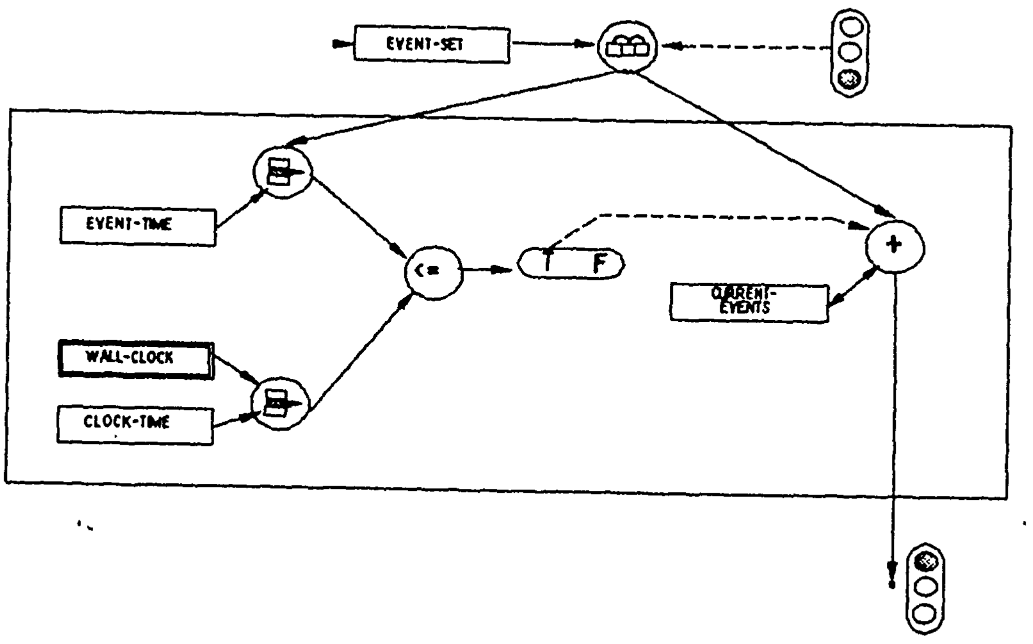


Figure I.3. Graphical Formulation of the Find-Current-Events Function

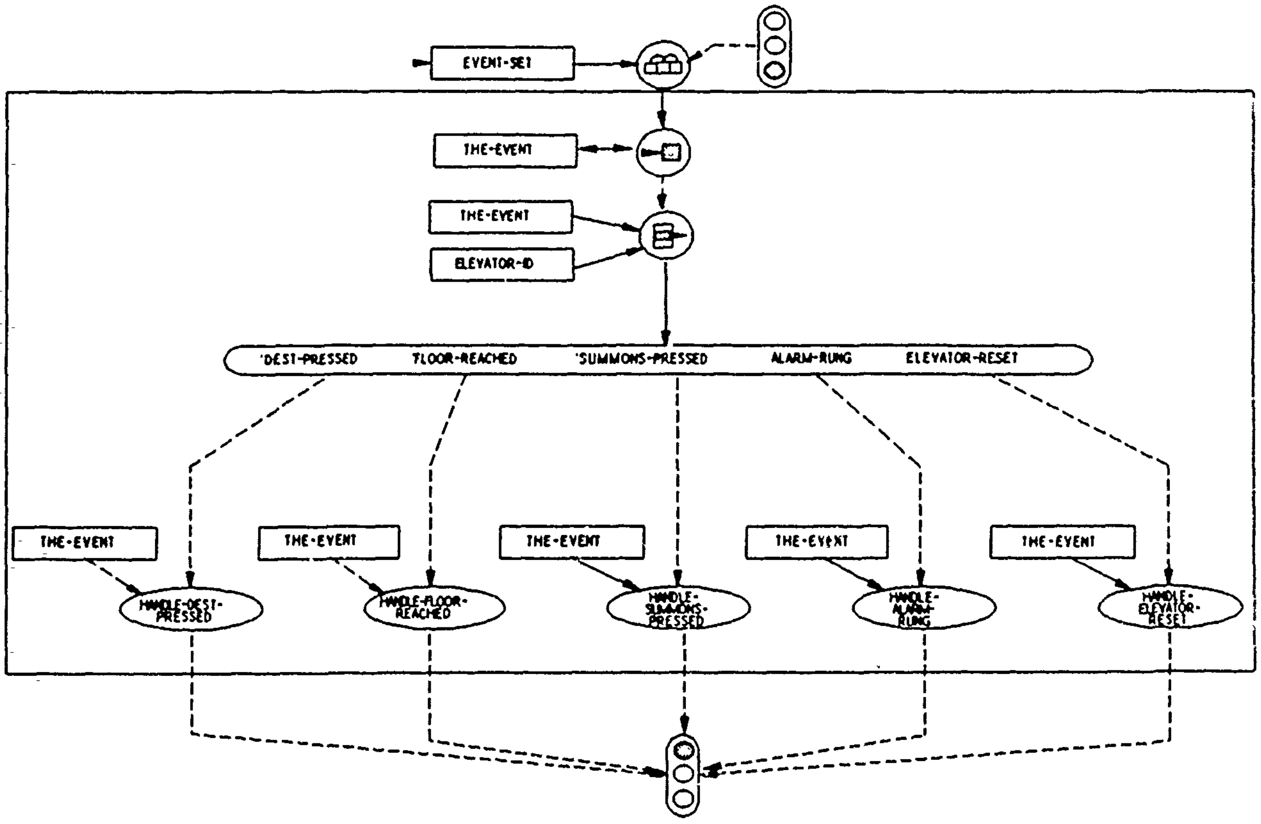


Figure I.4. Graphical Formulation of the Handle-Events Function

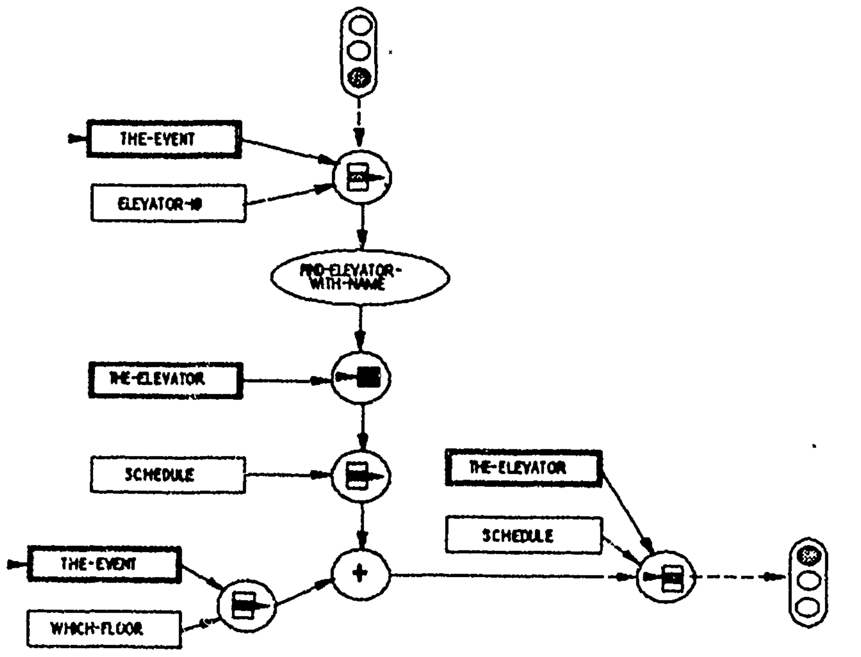


Figure I.5. Graphical Formulation of the Handle-Dest-Pressed Function

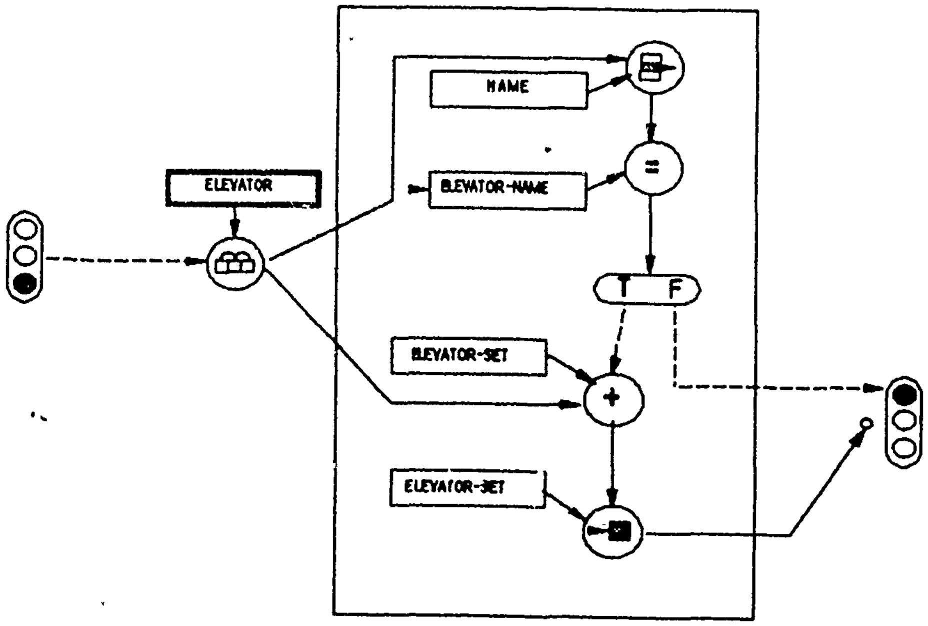


Figure I.6. Graphical Formulation of the Find-Elevator-with-Name Function

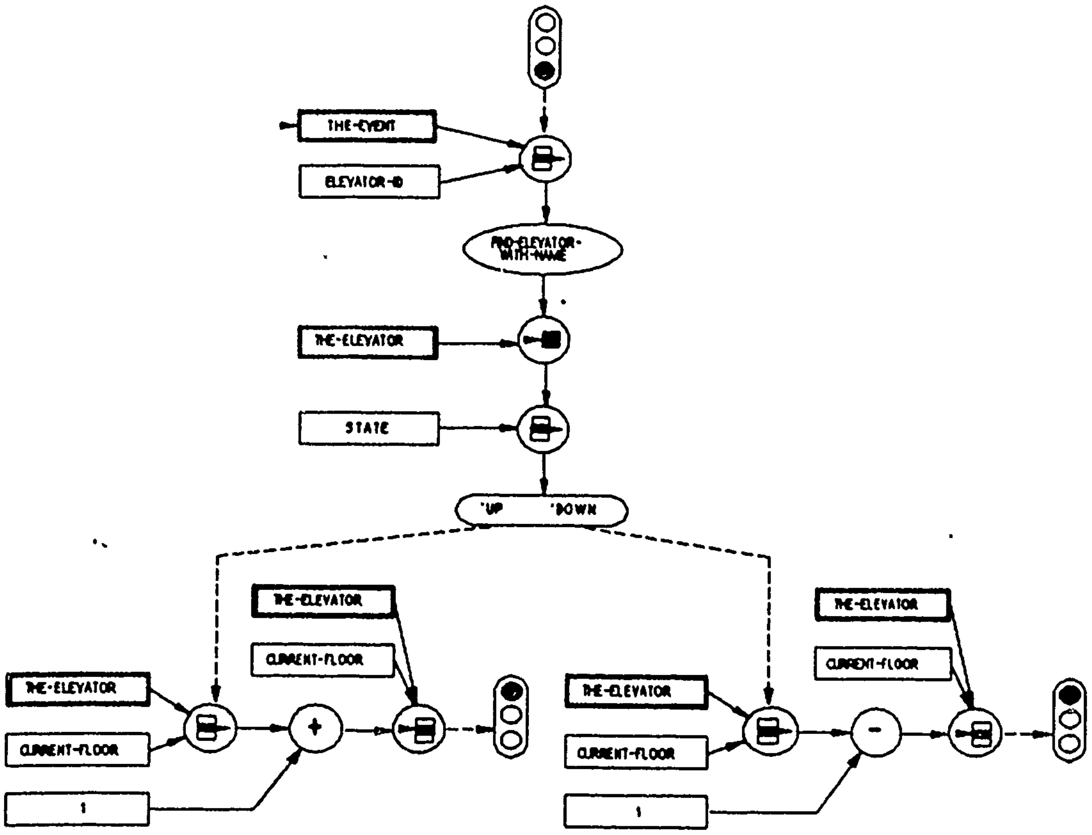


Figure I.7. Graphical Formulation of the Handle-Floor-Reached Function

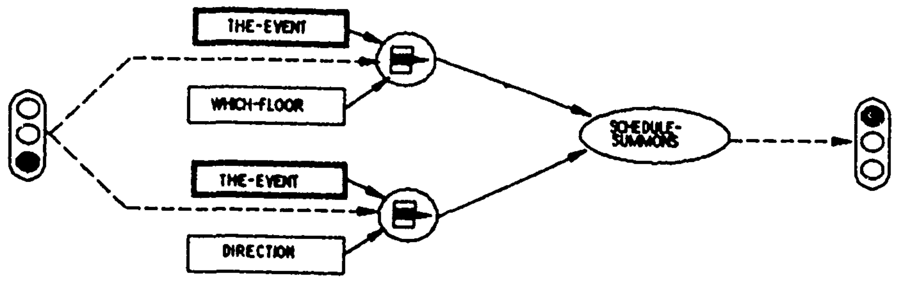


Figure I.8. Graphical Formulation of the Handle-Summons-Pressed Function

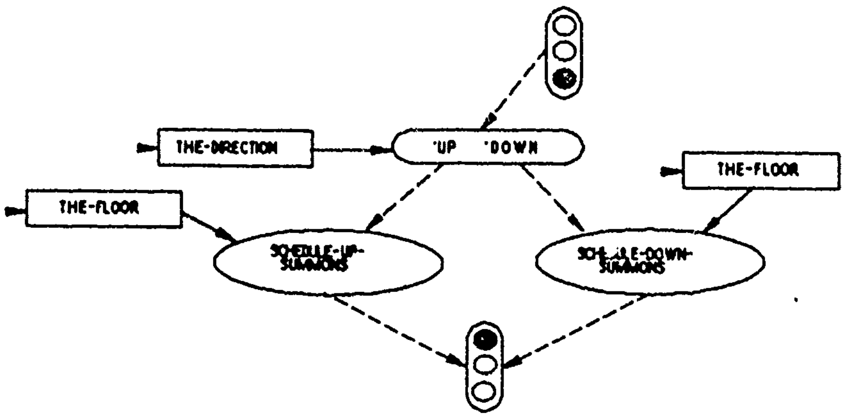


Figure I.9. Graphical Formulation of the Schedule-Summons Function

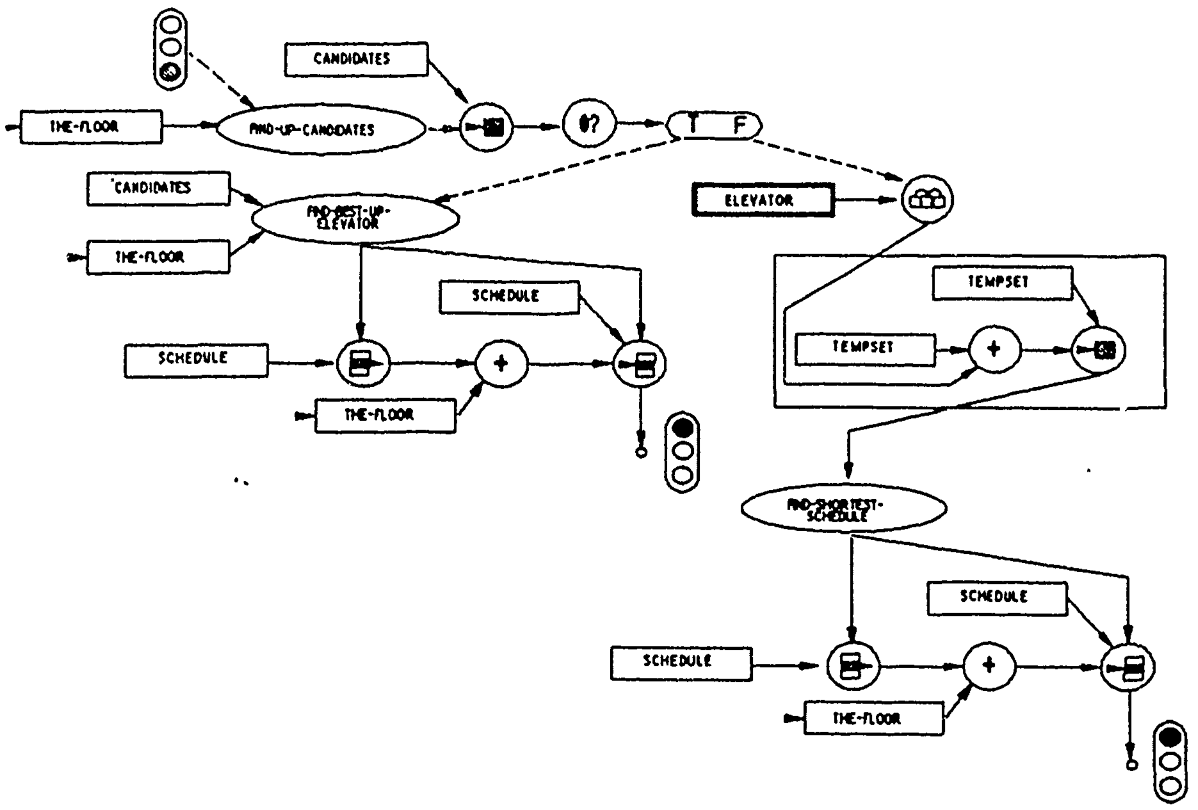


Figure I.10. Graphical Formulation of the Schedule-Up-Summons Function

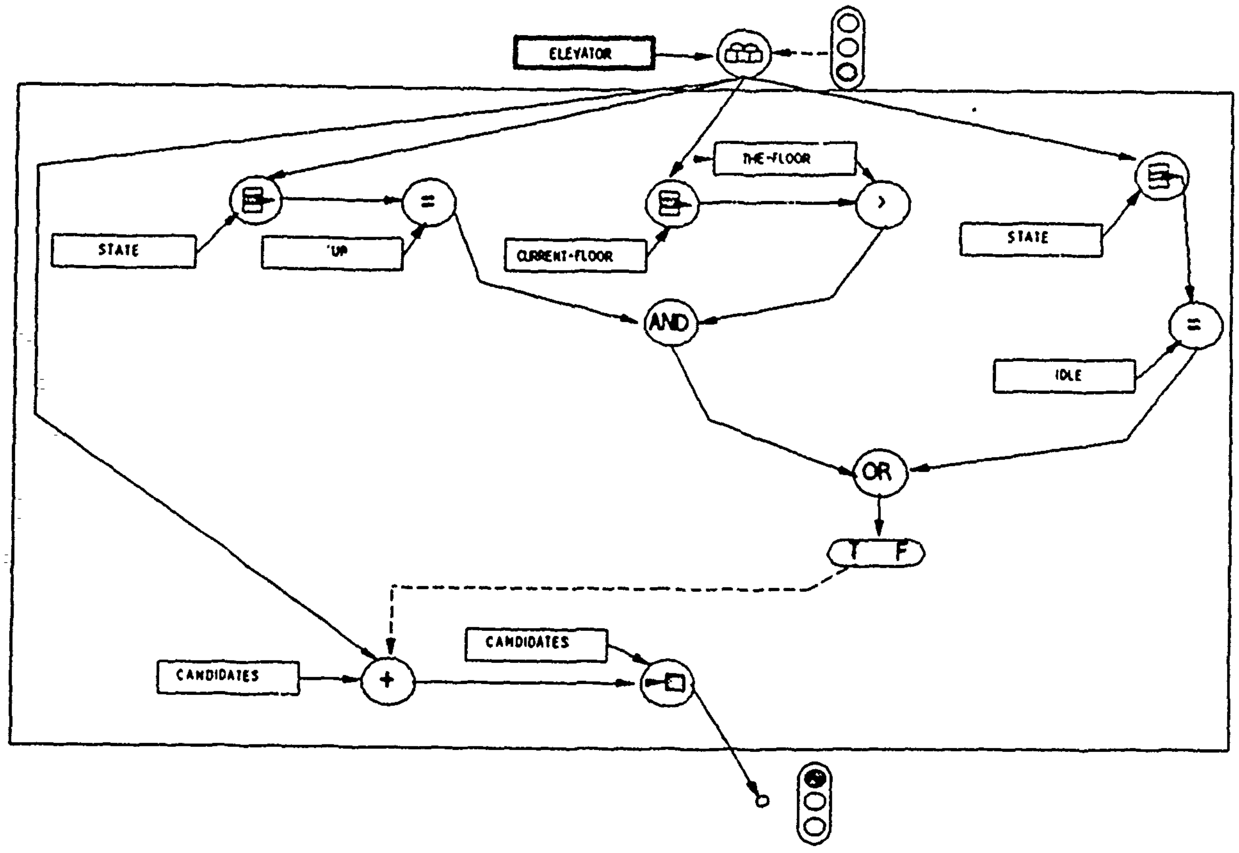


Figure I.11. Graphical Formulation of the Find-Up-Candidates Function

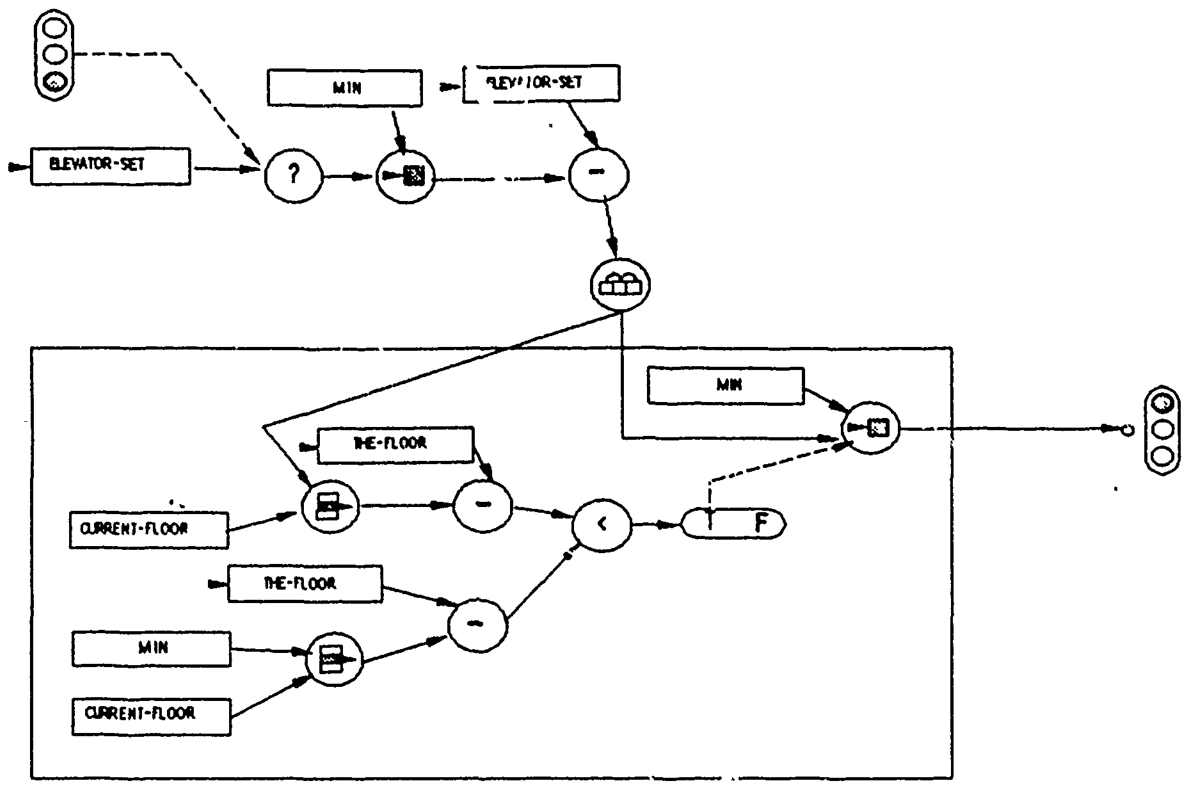


Figure I.12. Graphical Formulation of the Find-Best-Up-Elevator Function

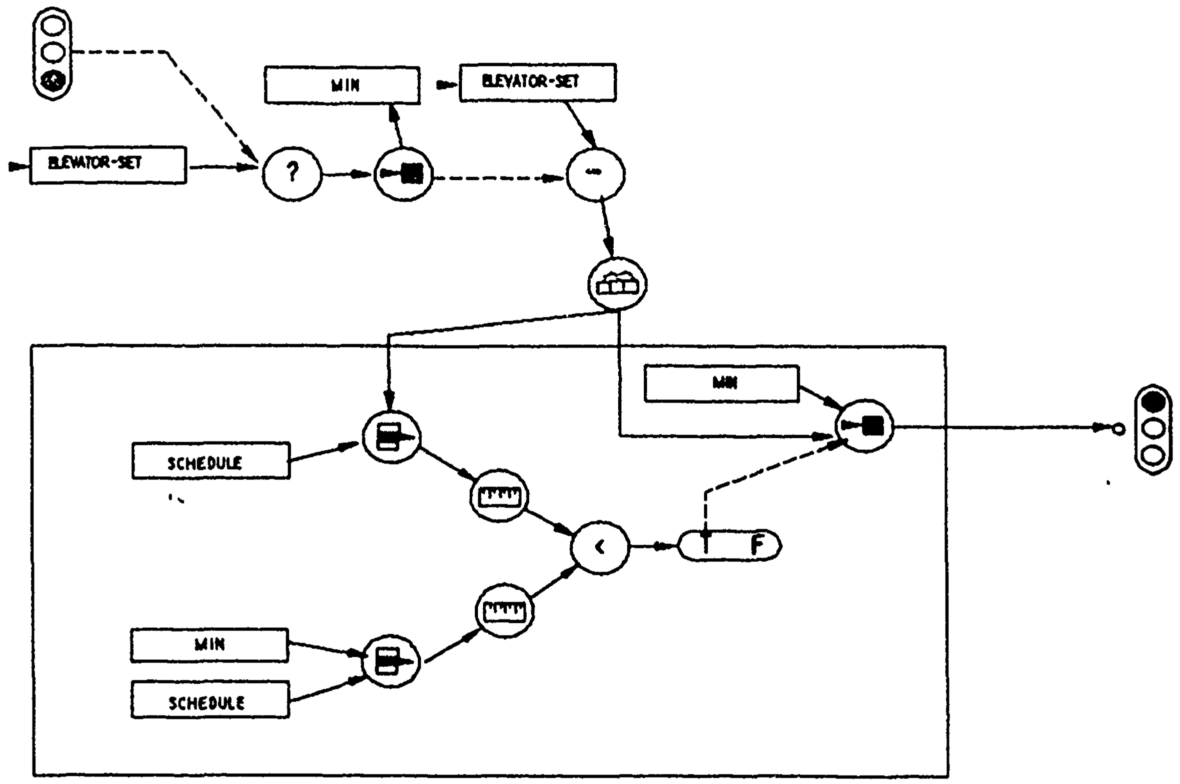


Figure I.13. Graphical Formulation of the Find-Shortest-Schedule Function

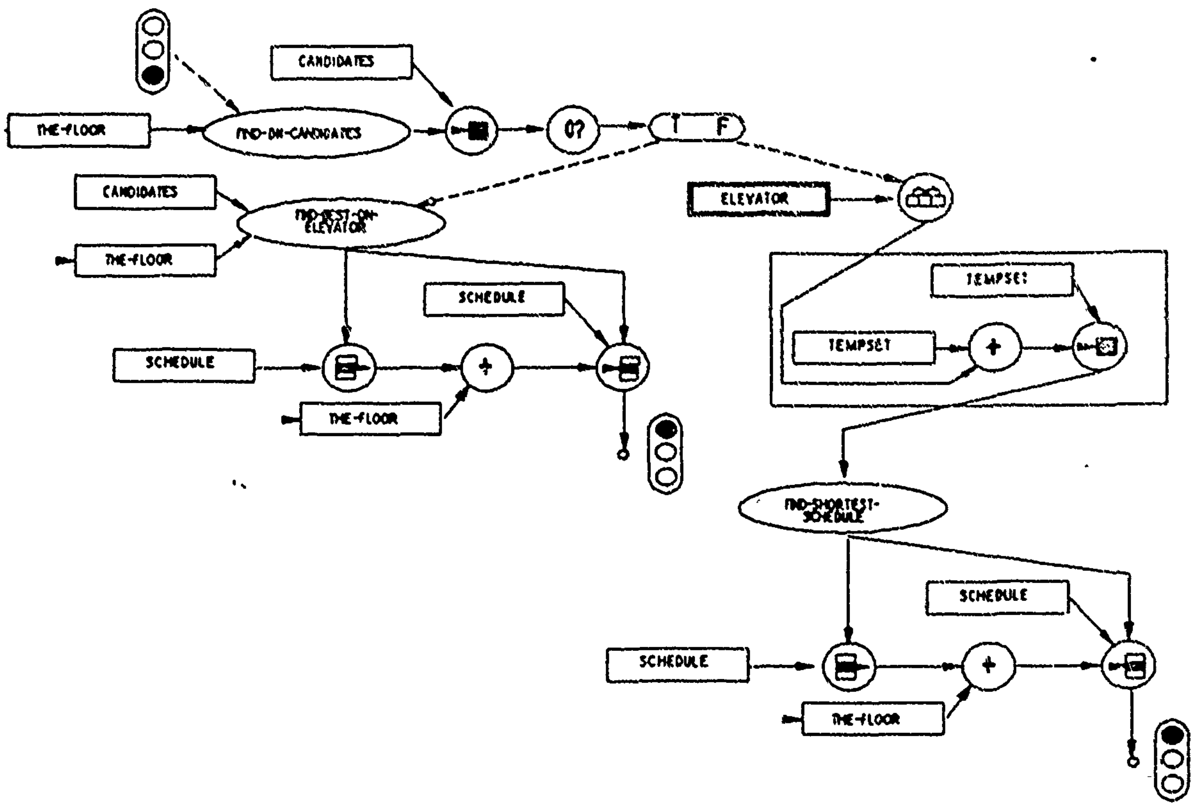


Figure I.14. Graphical Formulation of the Schedule-Down-Summons Function

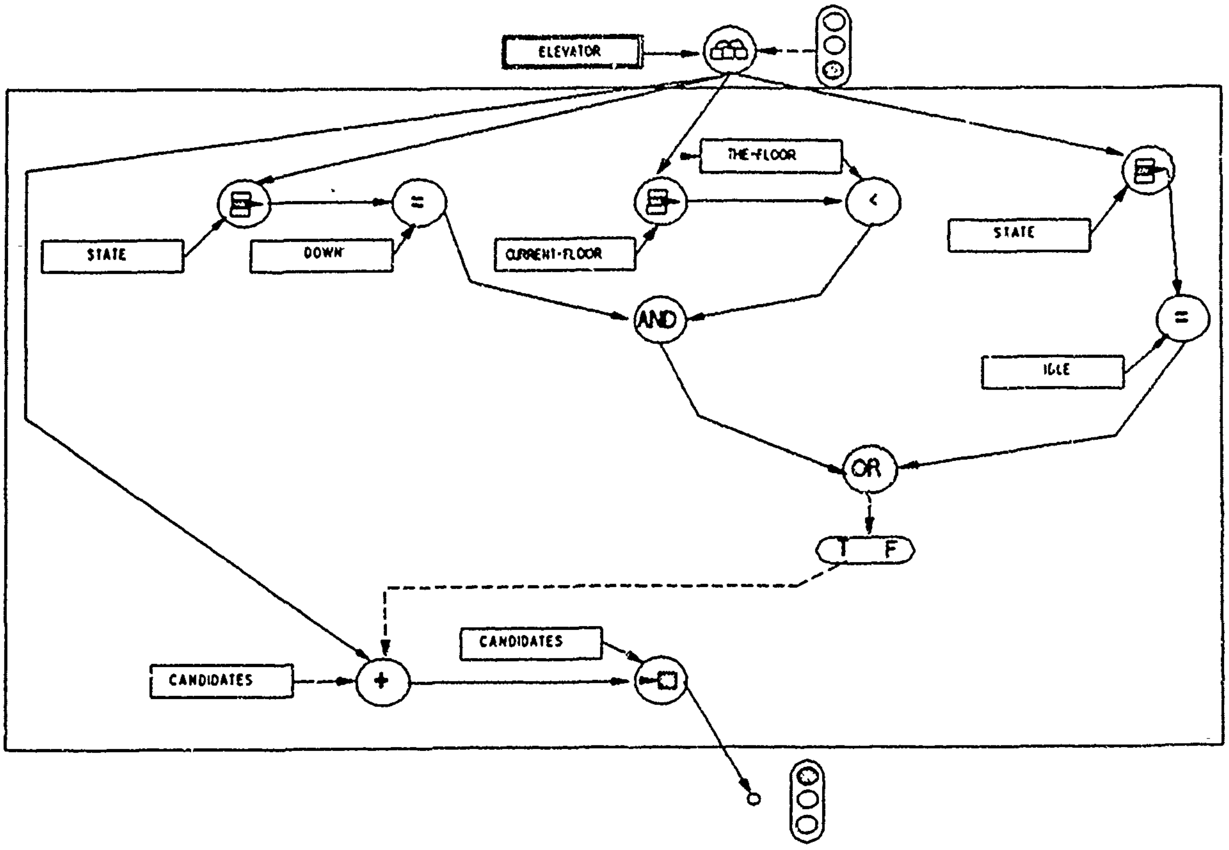


Figure I.15. Graphical Formulation of the Find-Dn-Candidates Function

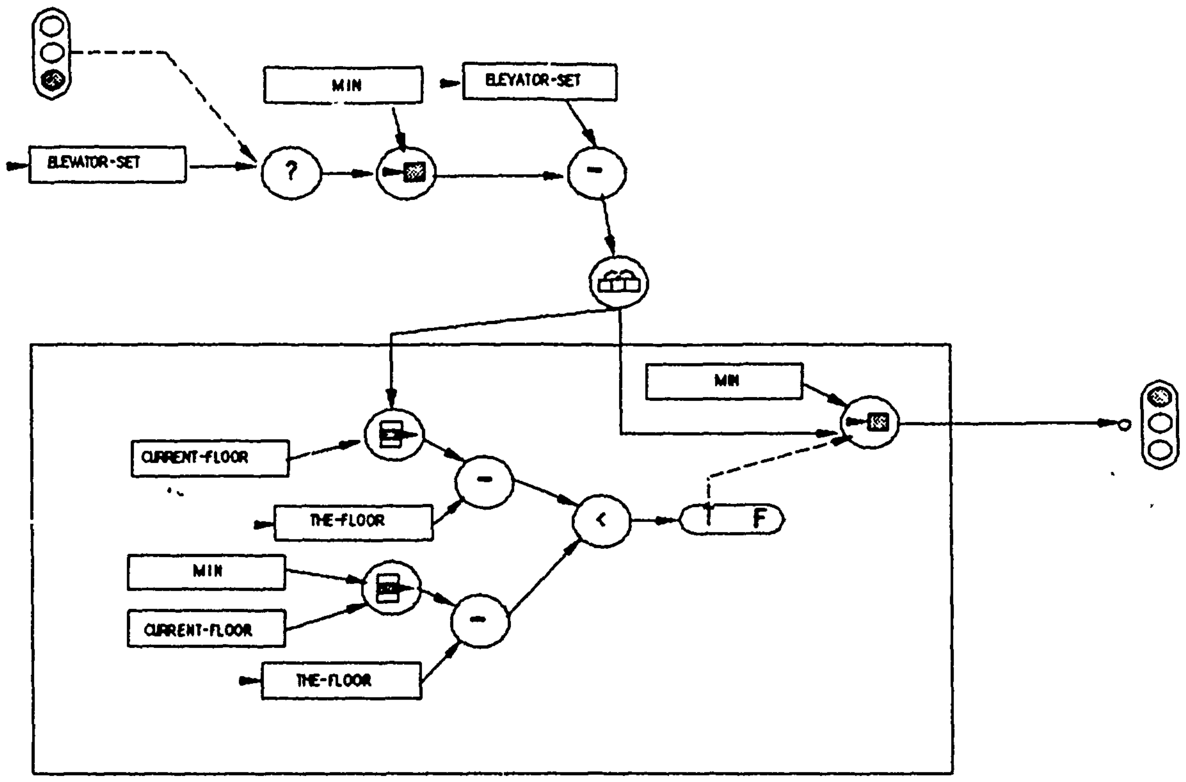


Figure I.16. Graphical Formulation of the Find-Best-Dn-Elevator Function

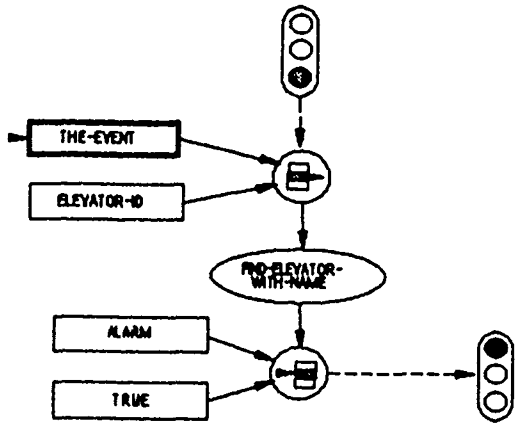


Figure I.17. Graphical Formulation of the Handle-Alarm-Rung Function

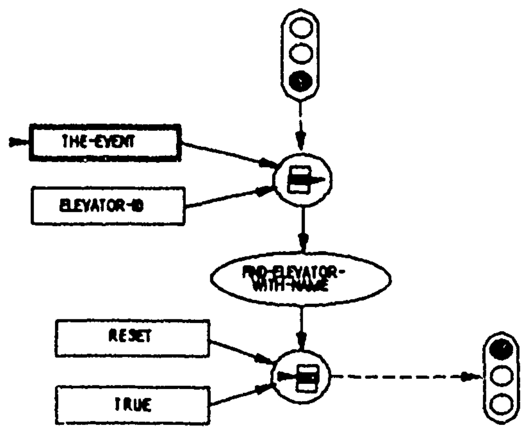


Figure I.18. Graphical Formulation of the Handle-Elevator-Reset Function

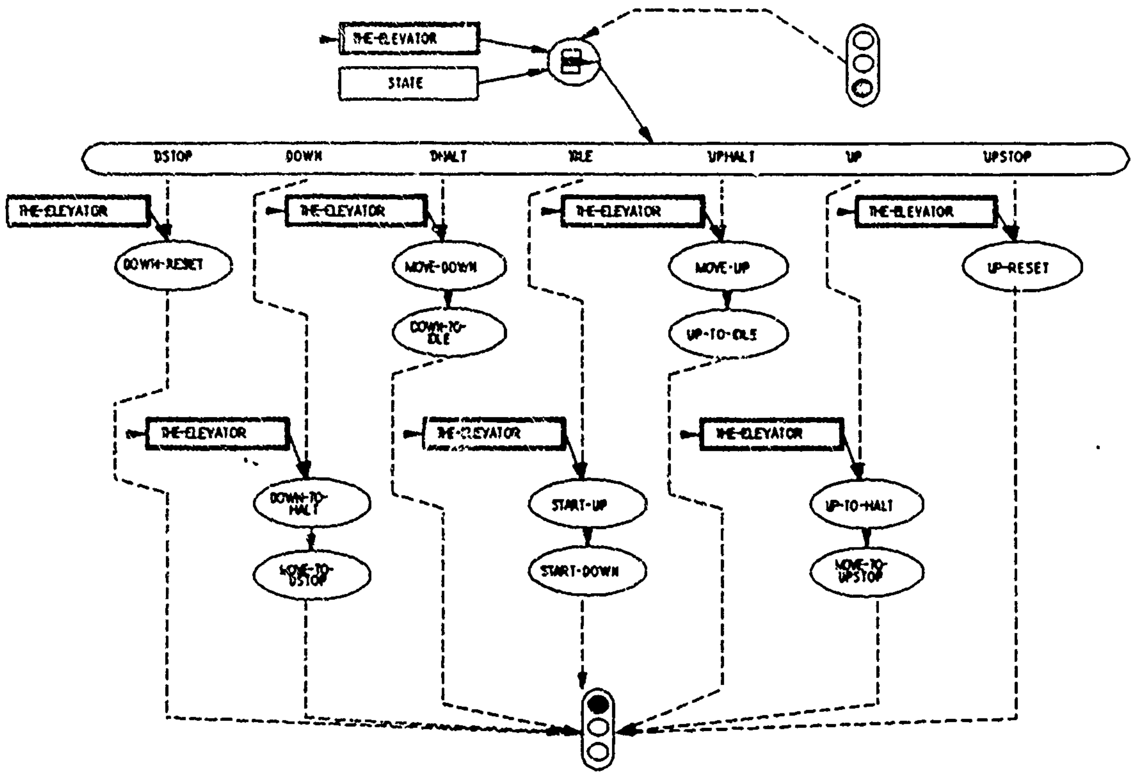


Figure I.19. Graphical Formulation of the Check-State Function

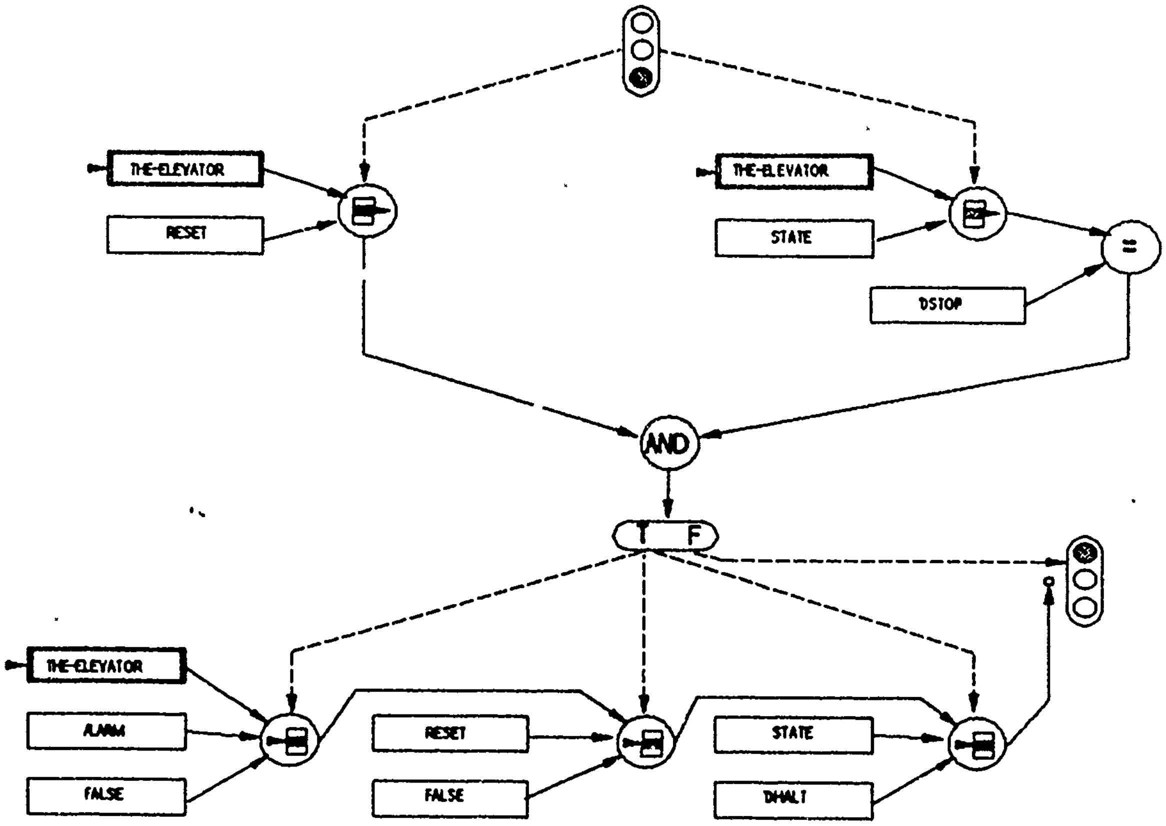


Figure I.20. Graphical Formulation of the Down-Reset Function

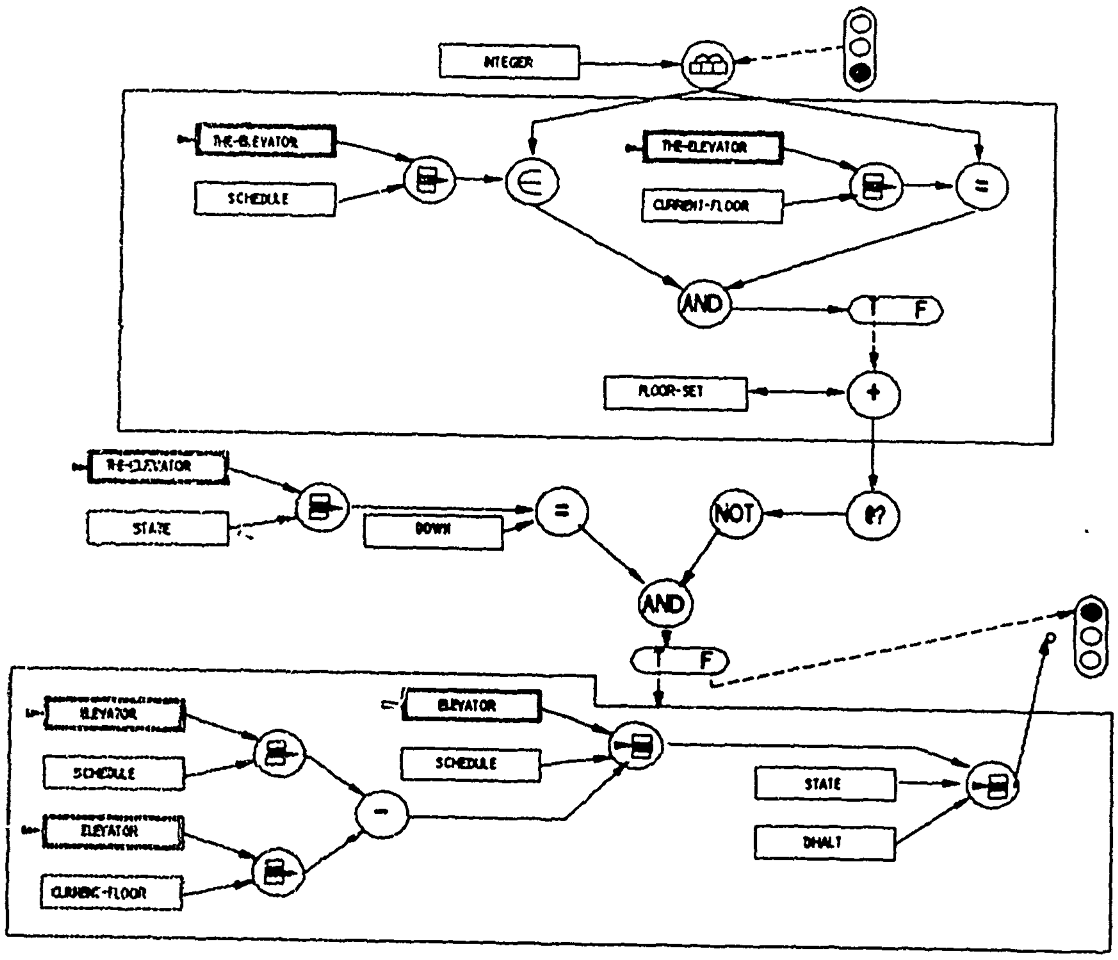


Figure I.21. Graphical Formulation of the Down-to-Halt Function

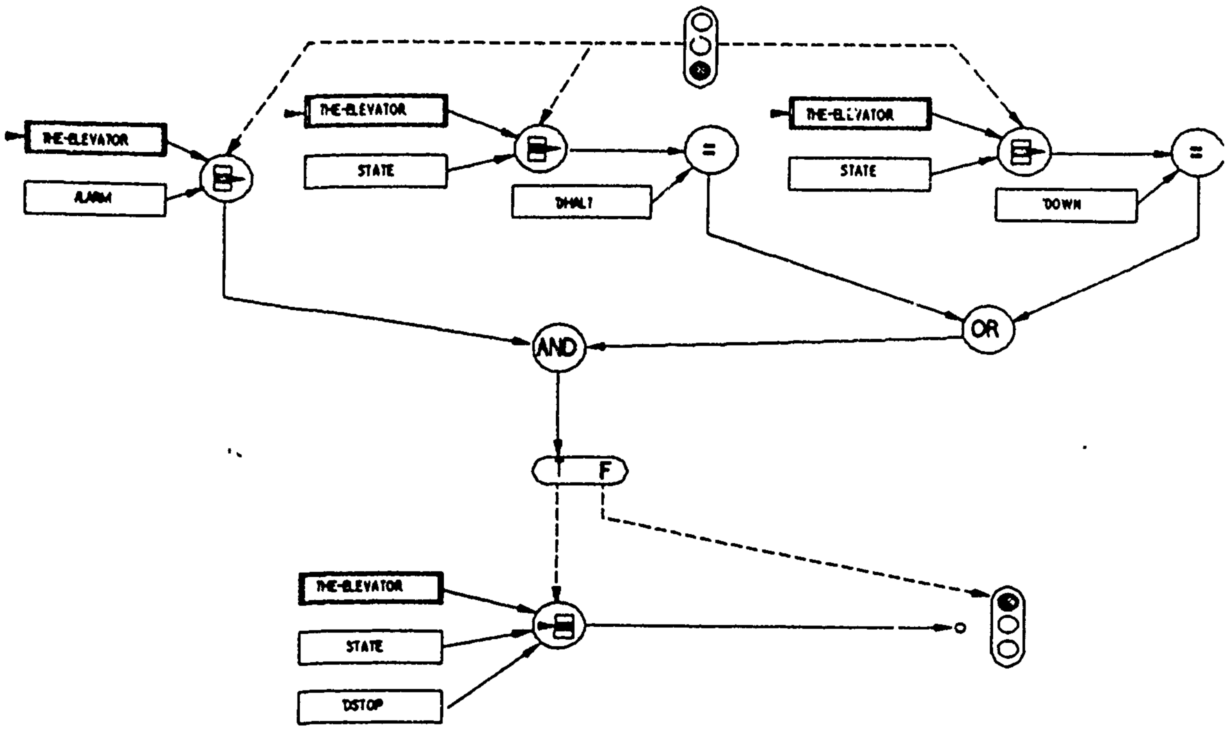


Figure I.22. Graphical Formulation of the Move-to-DStop Function

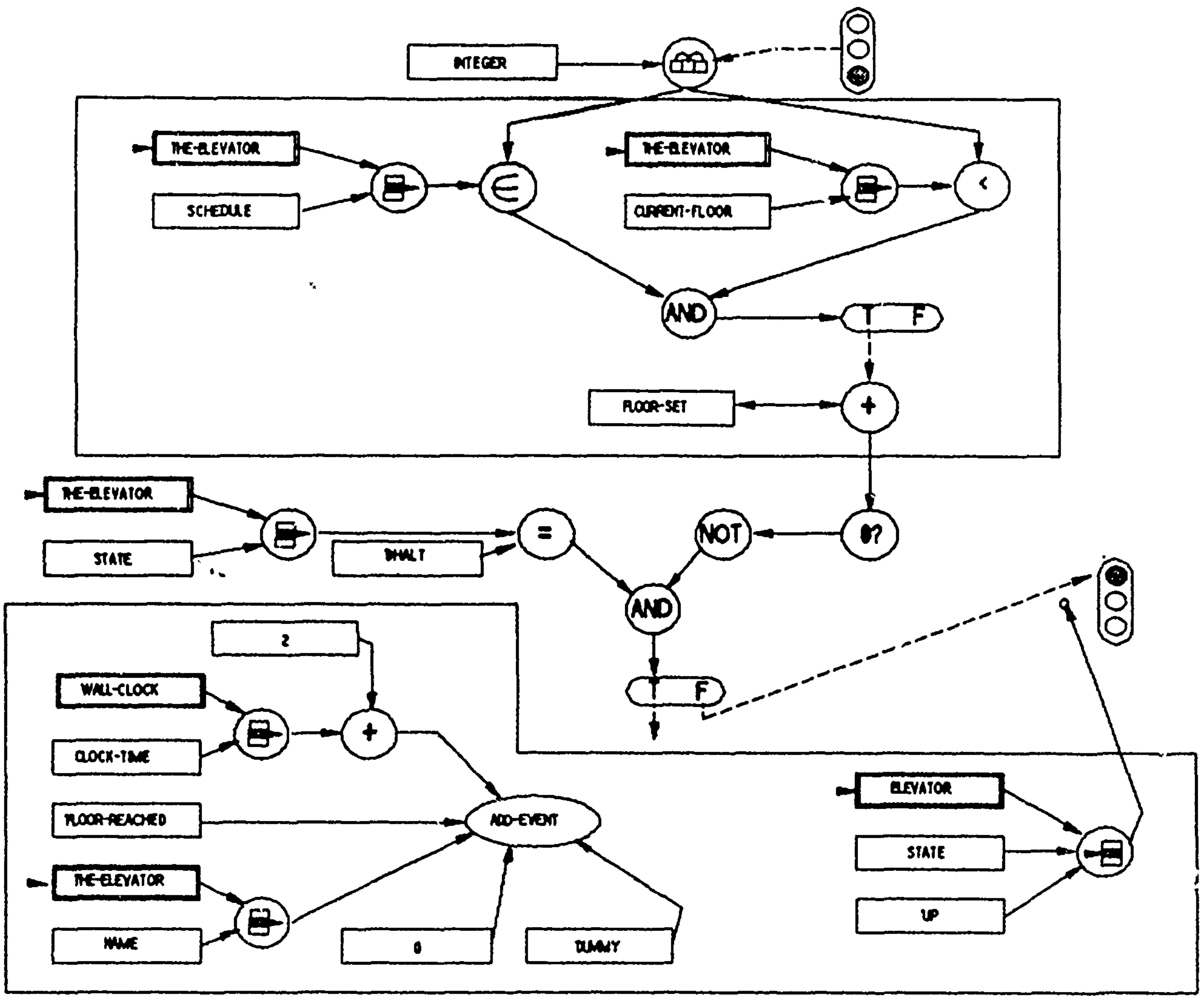


Figure I.23. Graphical Formulation of the Move-Down Function

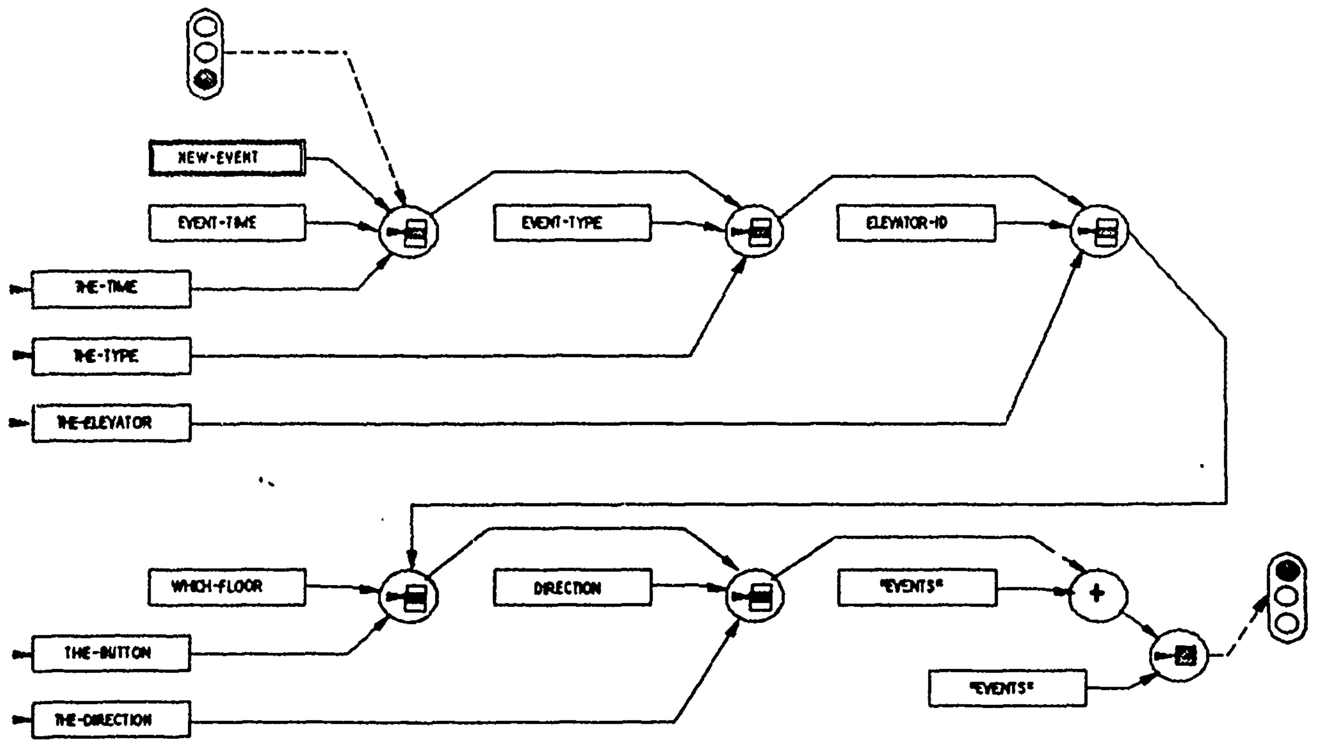


Figure I.24. Graphical Formulation of the Add-Event Function

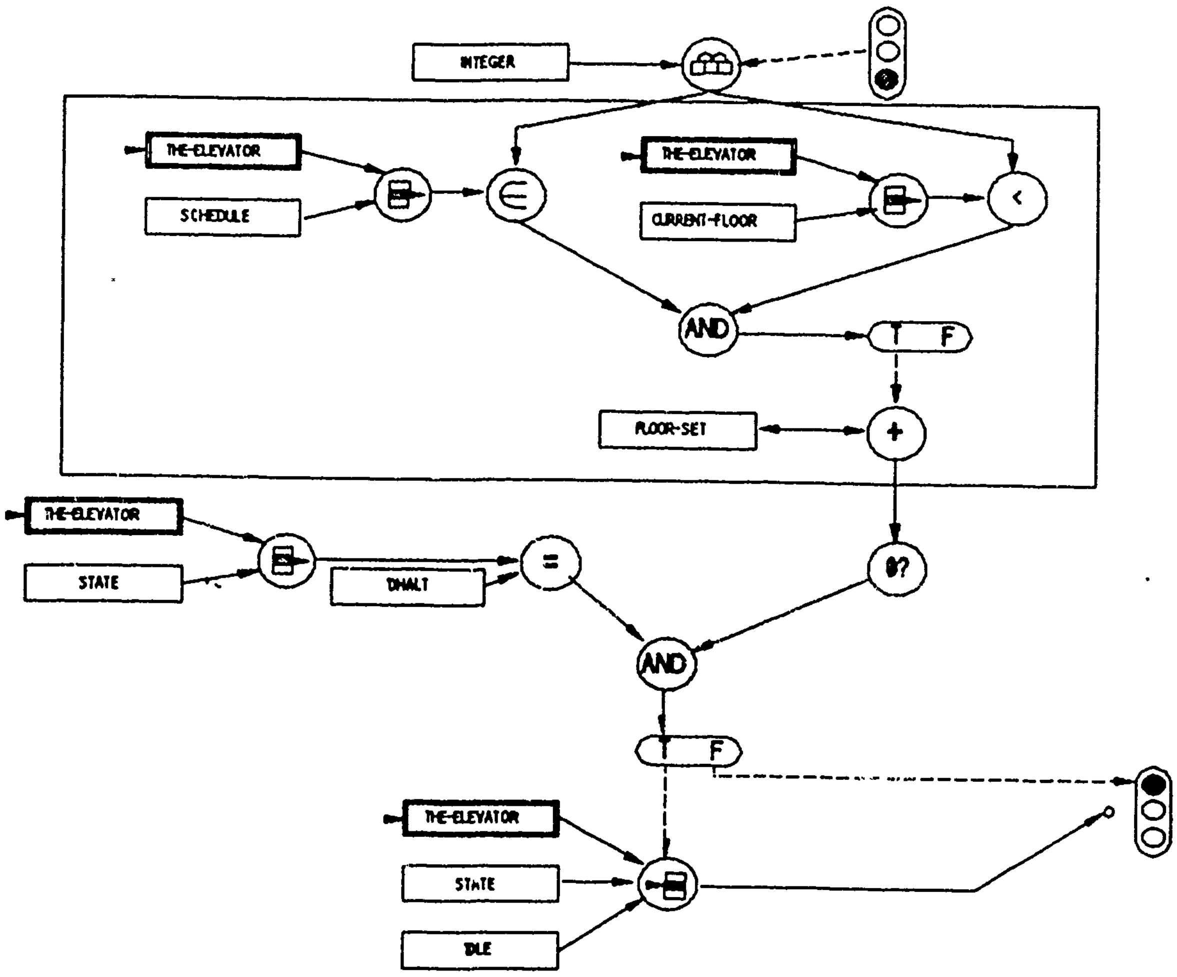


Figure I.25. Graphical Formulation of the Down-to-Idle Function

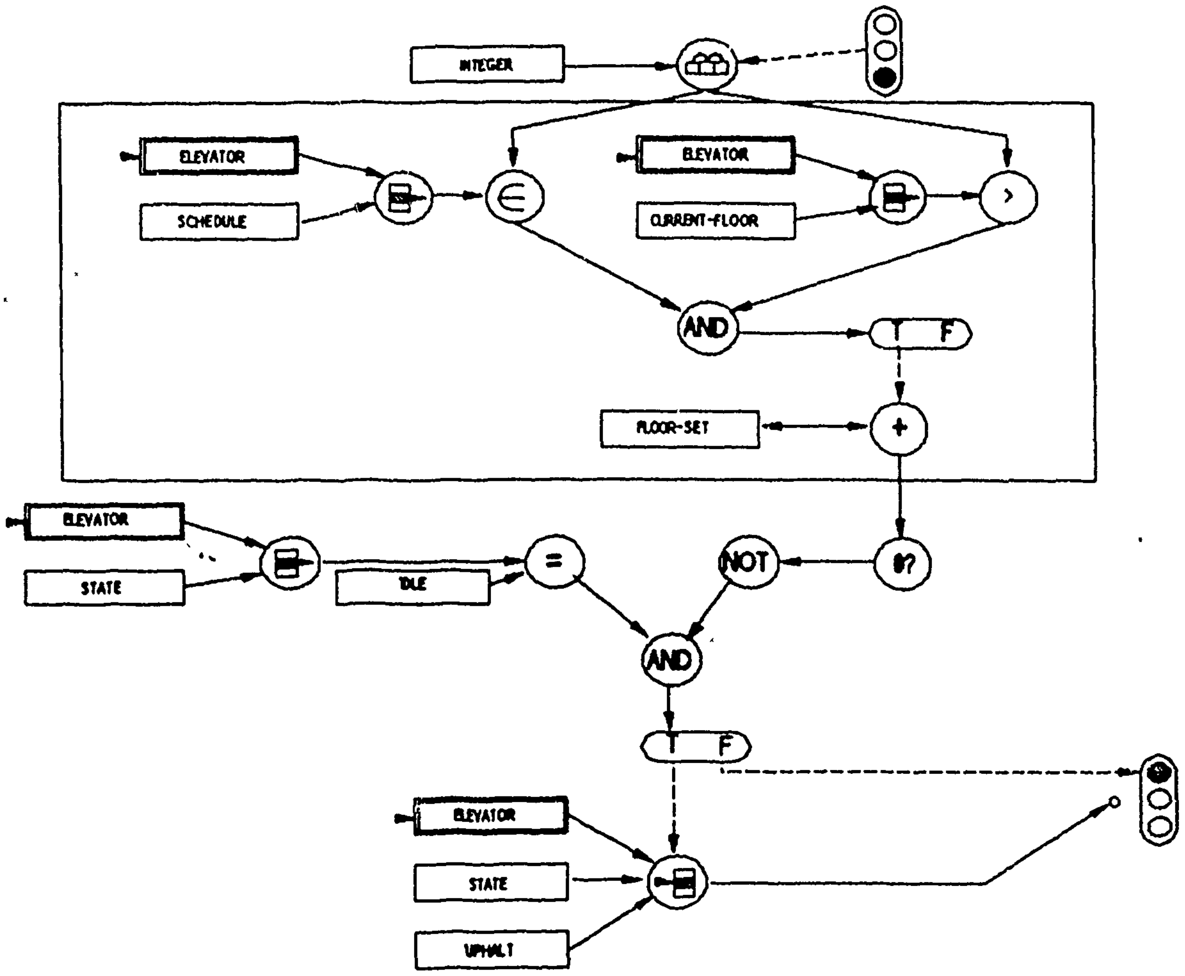


Figure I.26. Graphical Formulation of the Start-Up Function

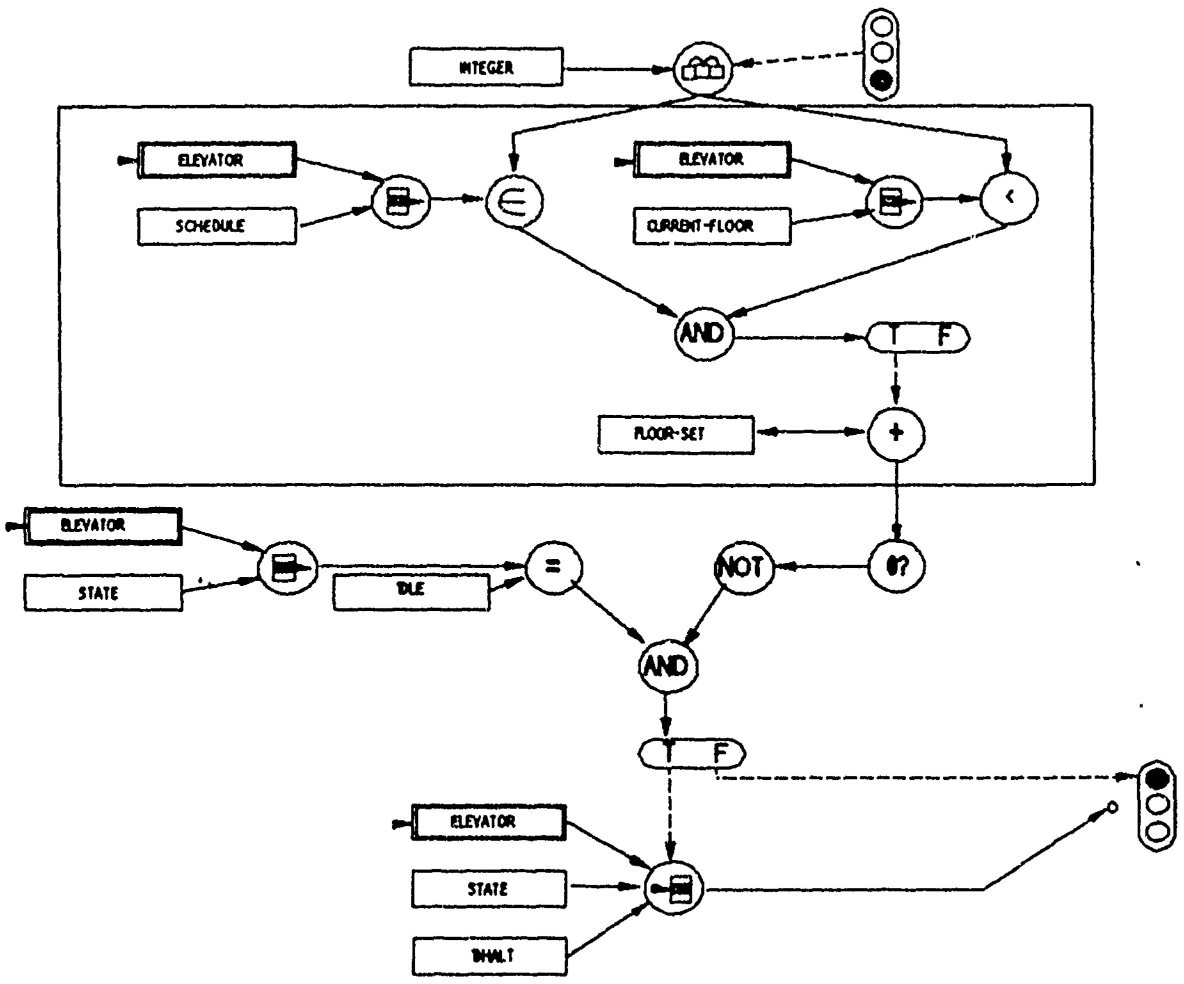


Figure I.27. Graphical Formulation of the Start-Down Function

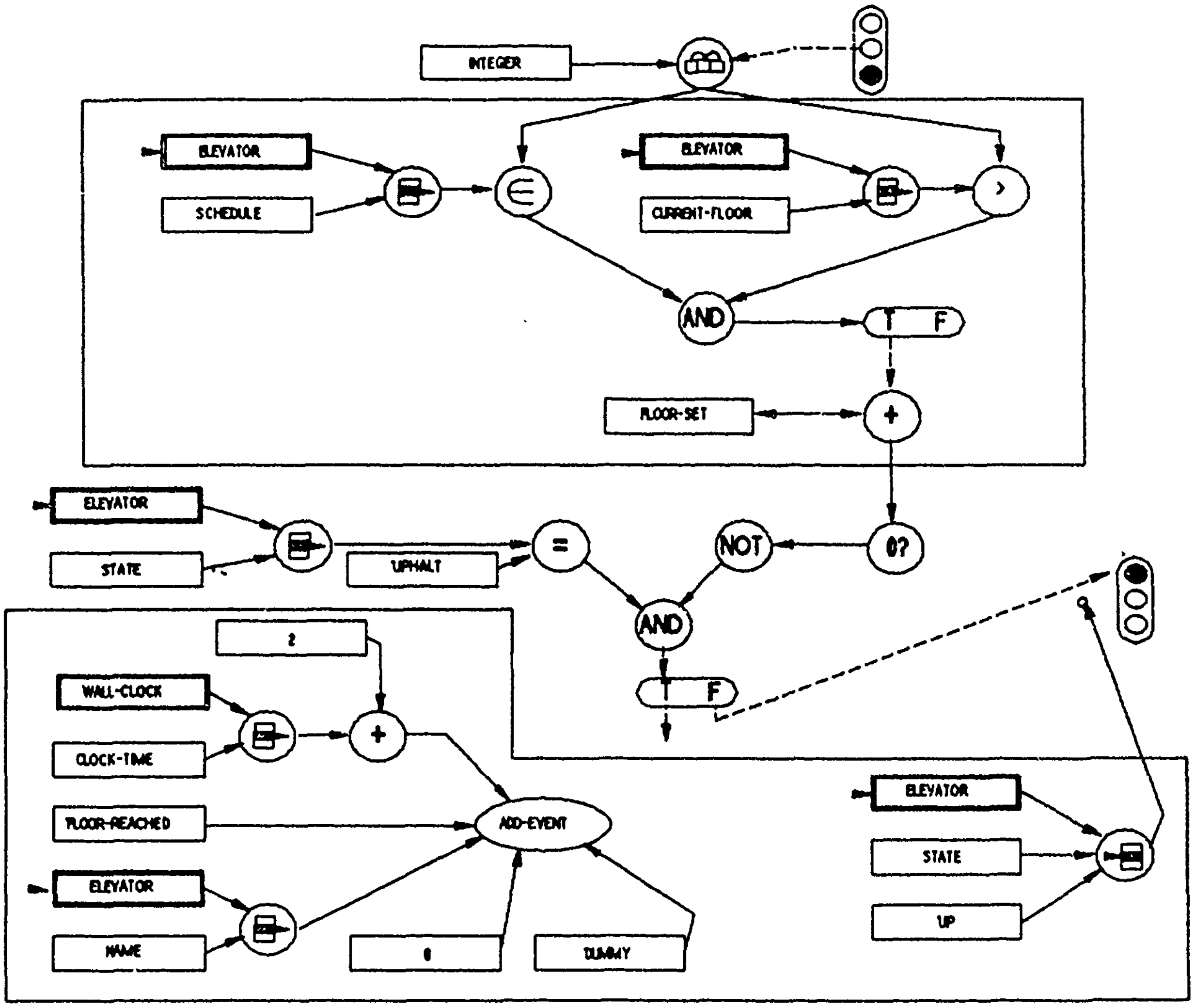


Figure I.28. Graphical Formulation of the Move-Up Function

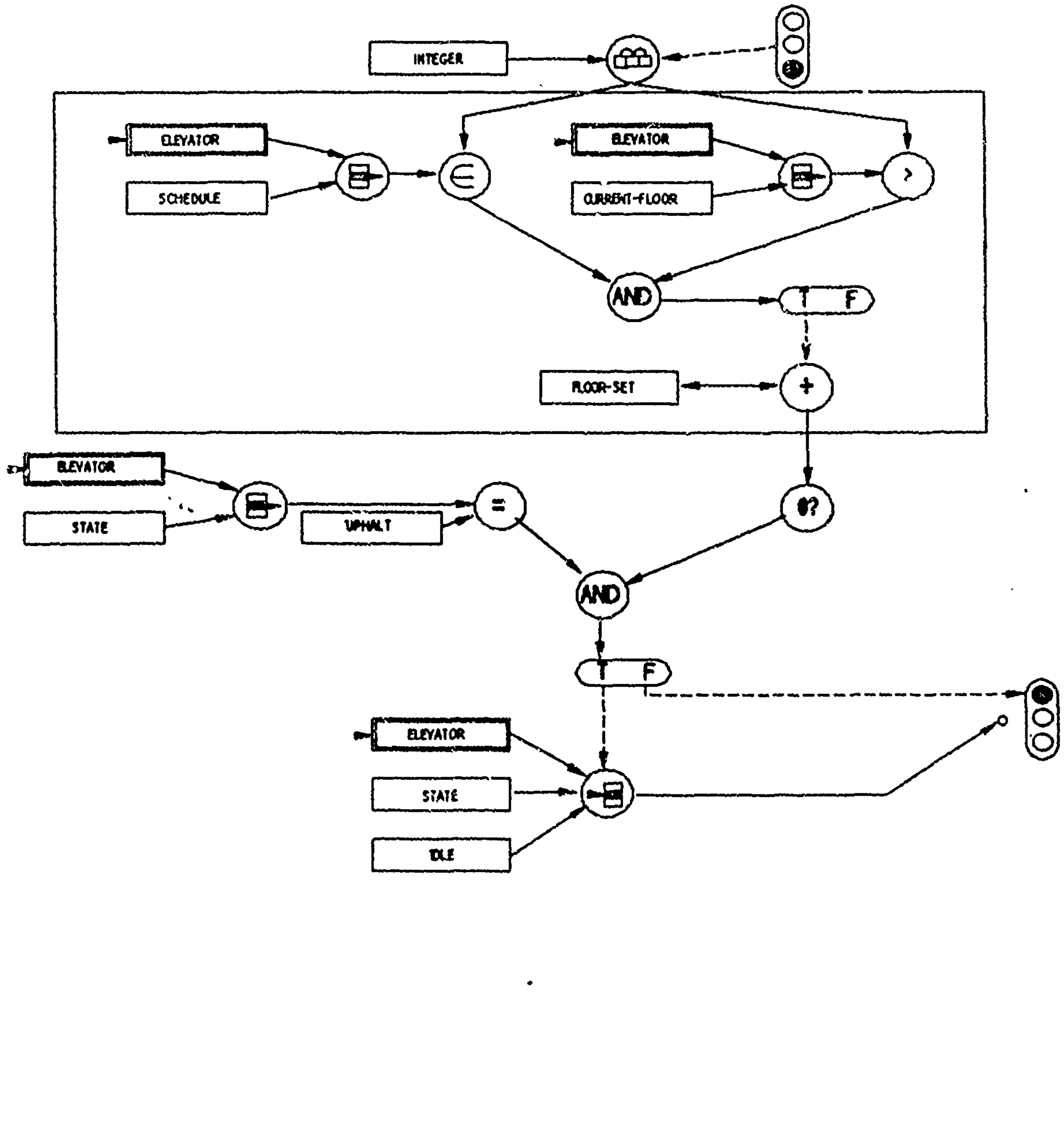


Figure I.29. Graphical Formulation of the Up-to-Idle Function

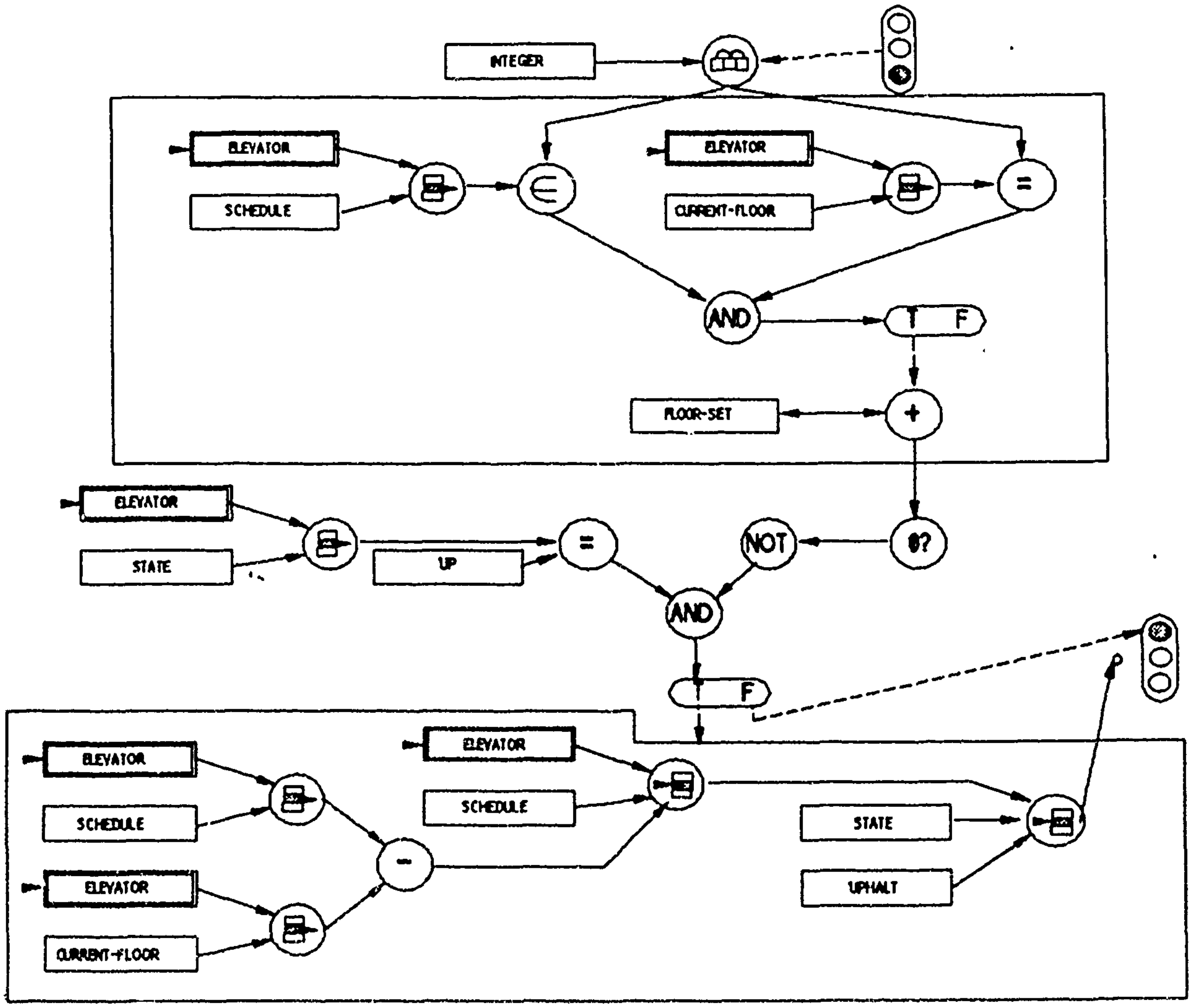


Figure I.30. Graphical Formulation of the Up-to-Halt Function

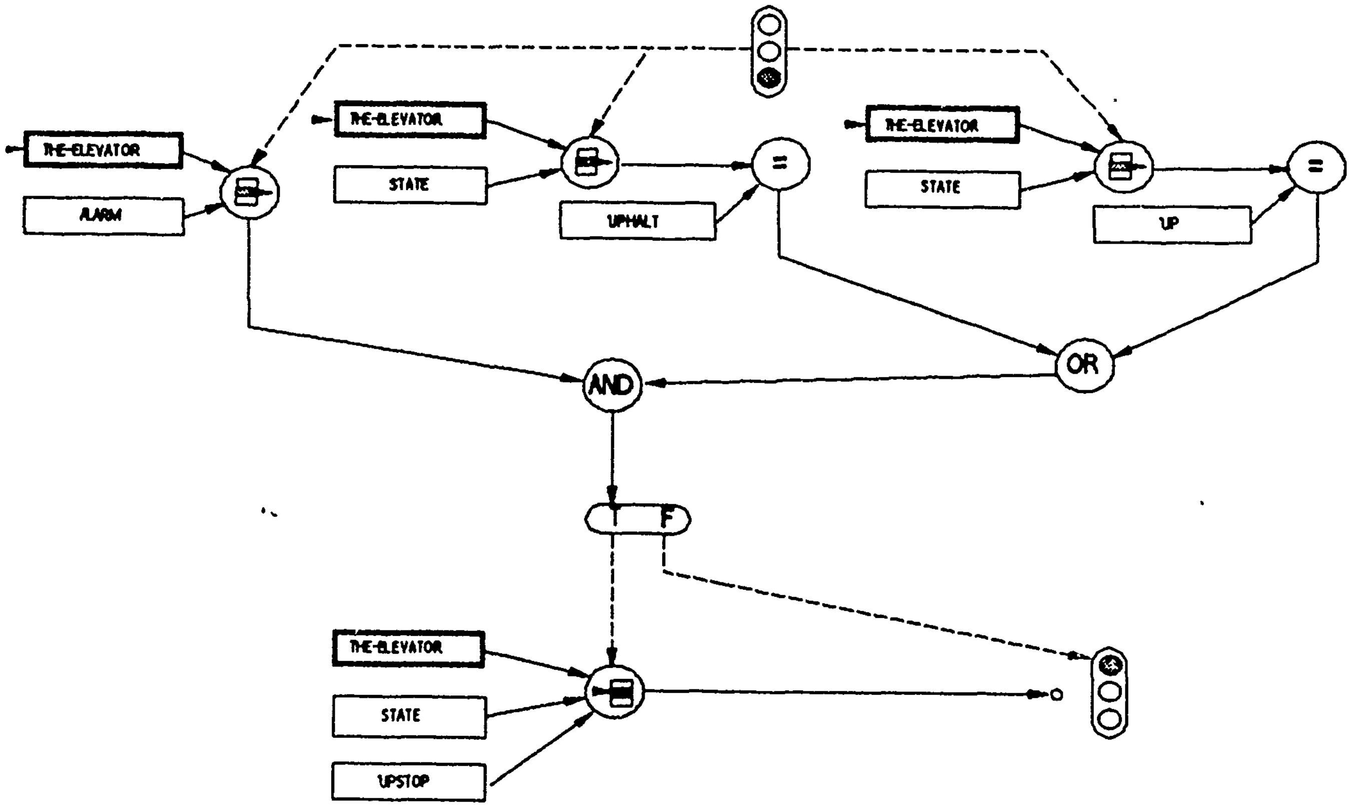


Figure I.31. Graphical Formulation of the Move-to-Upstop Function

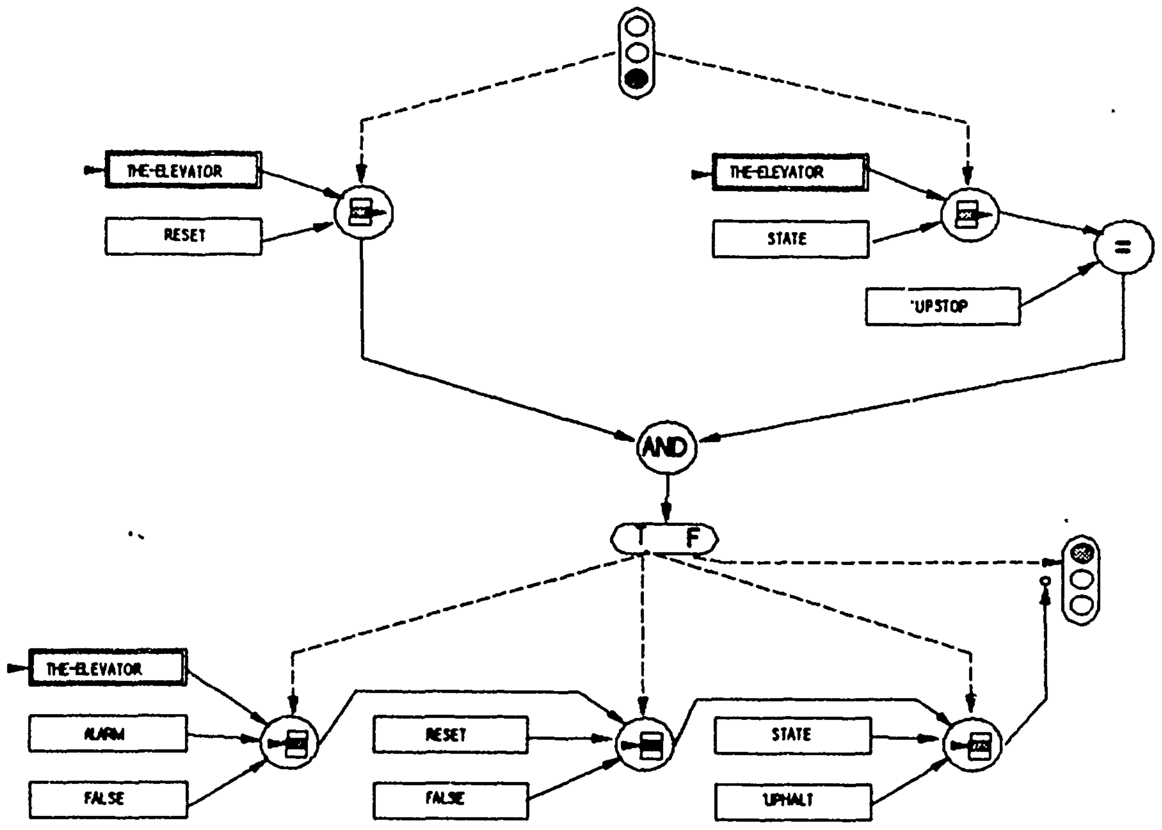


Figure I.32. Graphical Formulation of the UpReset Function

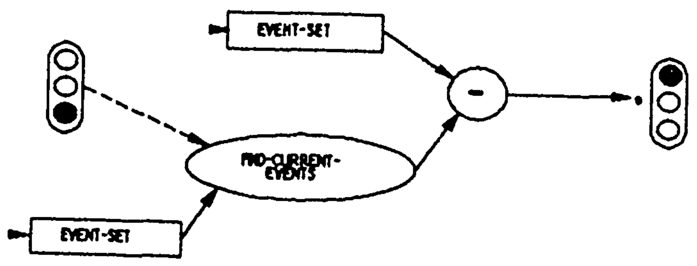


Figure I.33. Graphical Formulation of the Remove-Old-Events Function

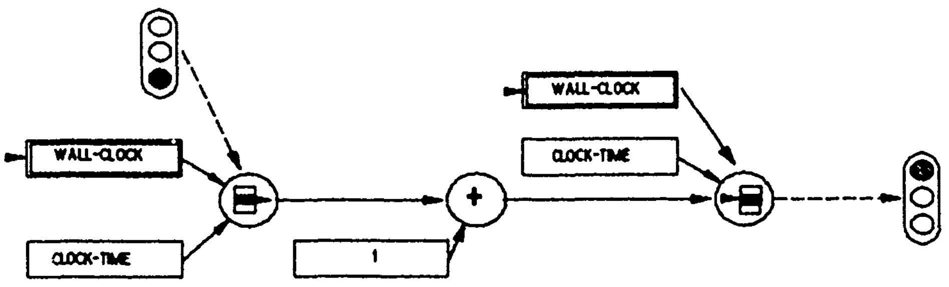


Figure I.34. Graphical Formulation of the Increment-Clock Function

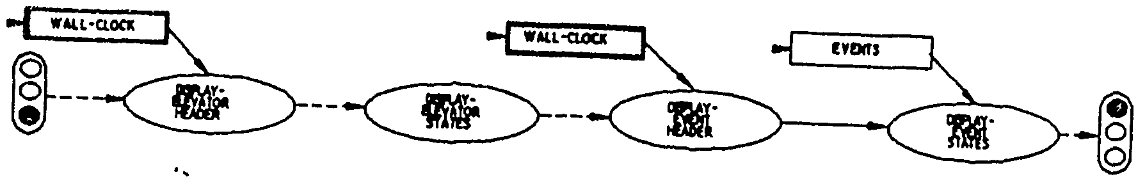


Figure I.35. Graphical Formulation of the Display-System-State Function

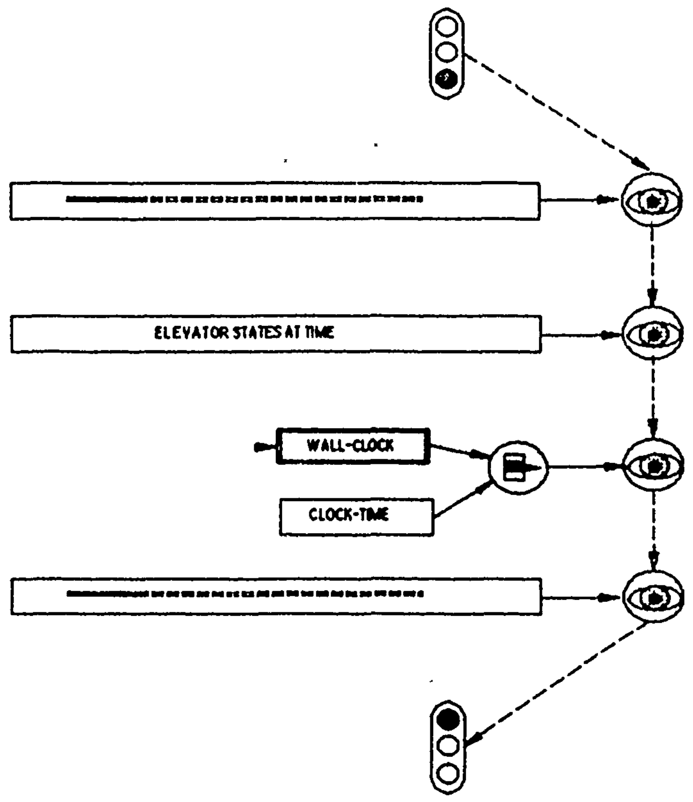


Figure I.36. Graphical Formulation of the Display-Elevator-Header Function

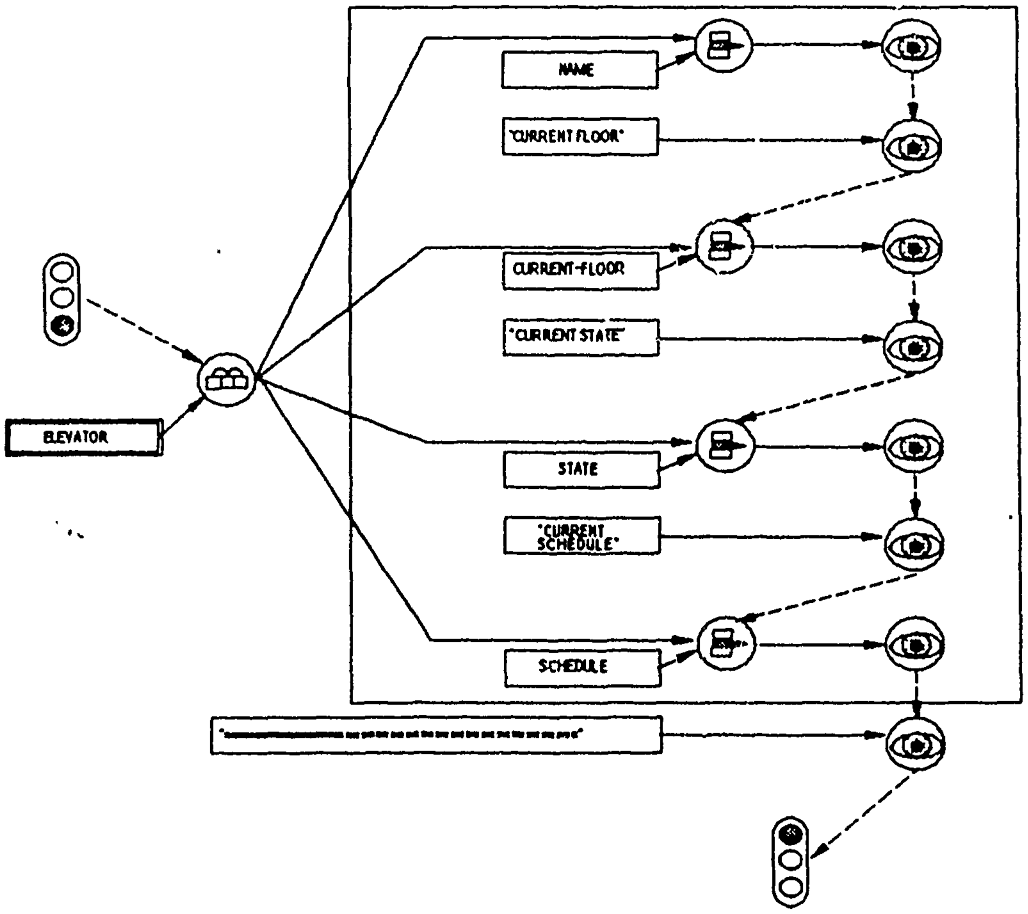


Figure I.37. Graphical Formulation of the Display-Elevator-State Function

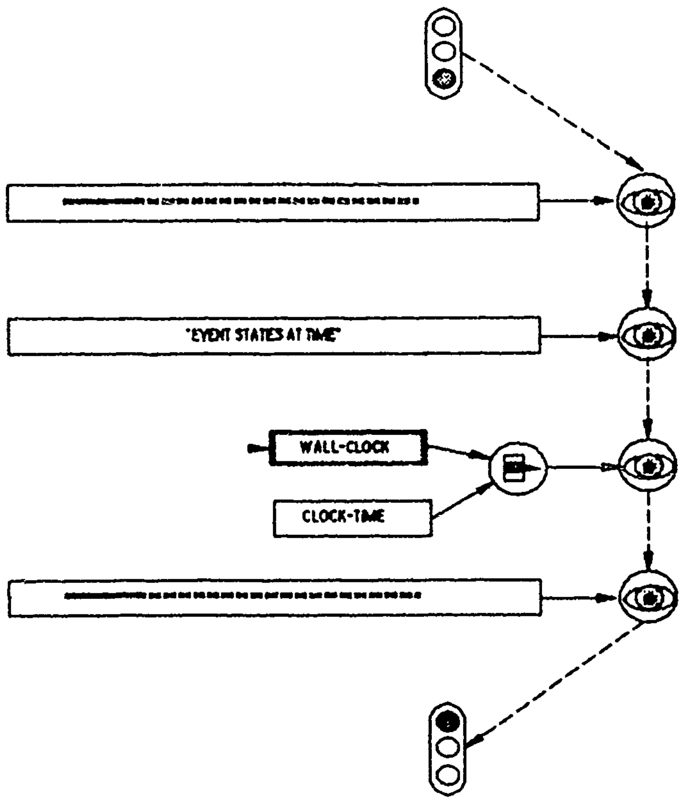


Figure I.38. Graphical Formulation of the Display-Event-Header Function

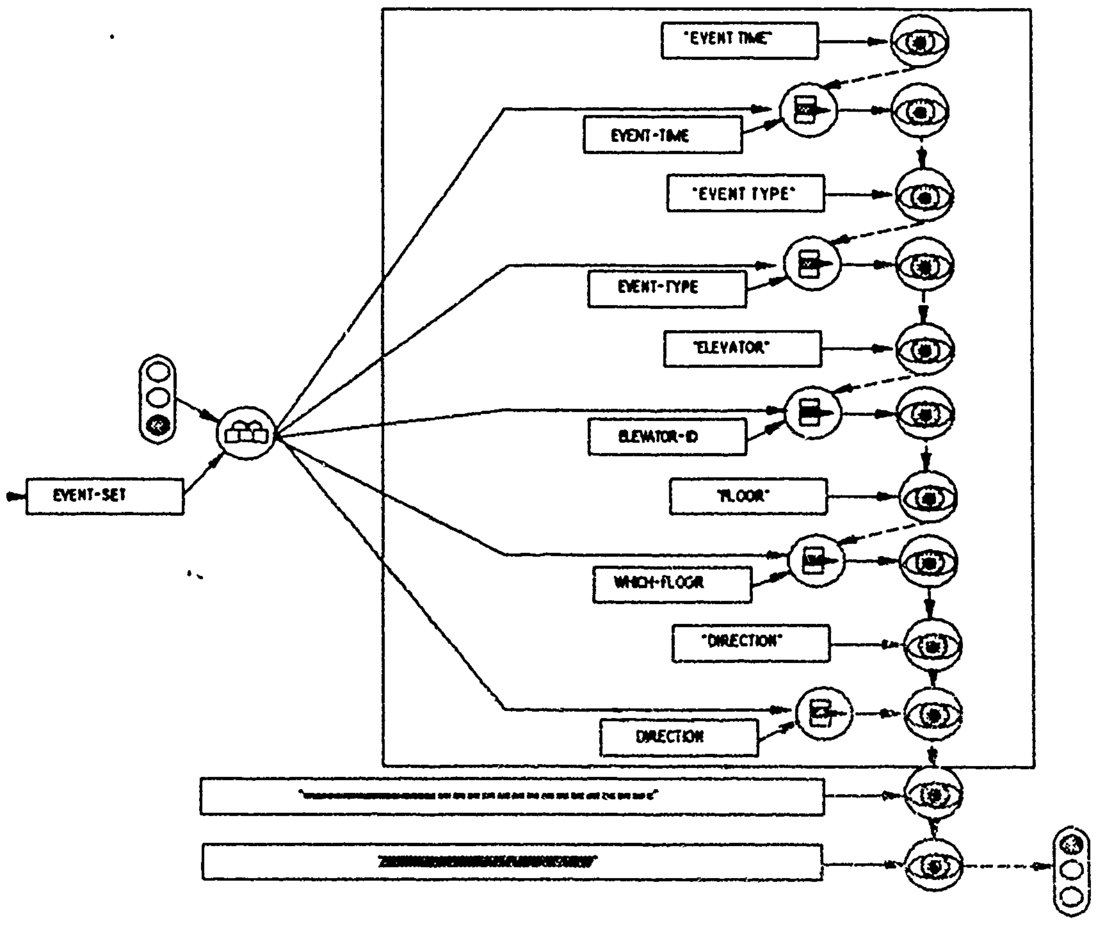


Figure I.39. Graphical Formulation of the Display-Event-States Function

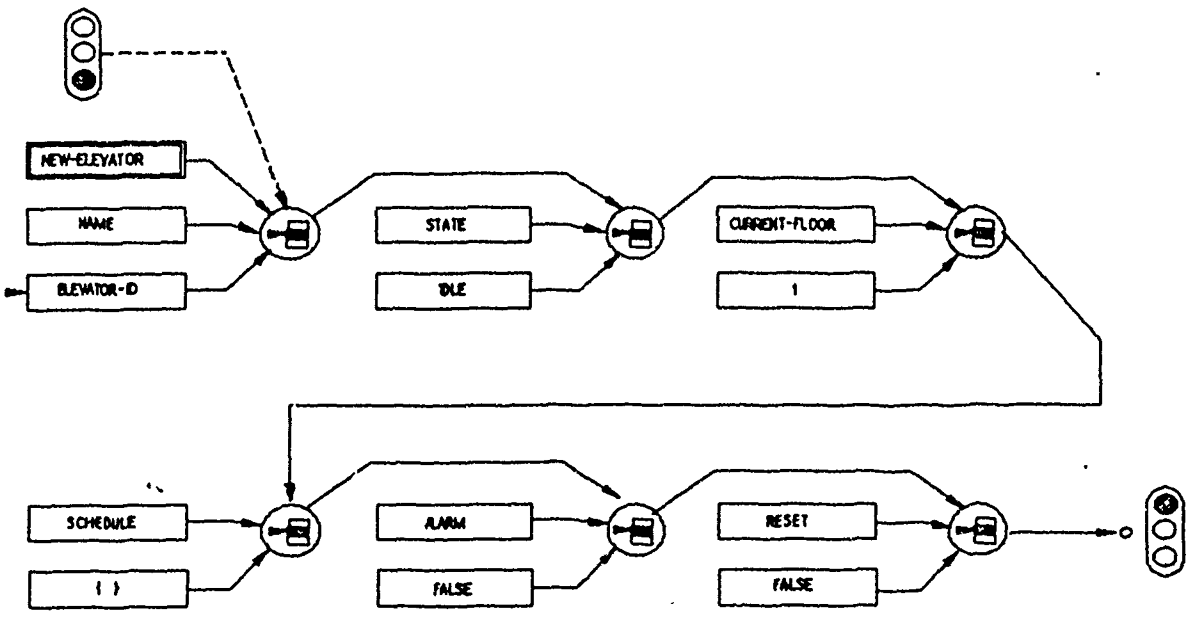


Figure I.40. Graphical Formulation of the Add-Elevator Function

Bibliography

1. Ambler, Allen L. and Margaret M. Bernett. "Influence of Visual Technology on the Evolution of Language Environments," *IEEE Computer*, *22*(10):9-22 (October 1989).
2. Balzer, Robert, et al. "Software Technology in the 1900's: Using a New Paradigm," *IEEE Computer*, pages 39-45 (November 1983).
3. Berztiss, Alfs. *SF Specification: A Library System*. SEI Curriculum Module SEI-SM-8-1.0, Software Engineering Institute, Carnegie Mellon University.
4. Berztiss, Alfs. *SF Specification: An Elevator Controller*. SEI Curriculum Module SEI-SM-8-1.0, Software Engineering Institute, Carnegie Mellon University.
5. Berztiss, Alfs. *Formal Specification of Software*. SEI Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, October 1987.
6. Berztiss, Alfs. "Formal Specification Methods and Visualization." In Chang, Shi-Kuo, editor, *Principles of Visual Programming Systems*, pages 231-290, Englewood Cliffs:Prentice Hall, 1990.
7. Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs:Prentice-Hall, 1981.
8. Boehm, Barry W. "Improving Software Productivity," *IEEE Computer*, *20*(9):43-57 (September 1987).
9. Bustard, D. M., et al. "A Pictorial Approach to the Animation of Process Oriented Formal Specifications," *Software Engineering Journal*, pages 114-118 (July 1988).
10. Davis, Alan L. and Robert M. Keller. "Data Flow Graphs," *IEEE Computer*, pages 26-41 (February 1982).
11. Davis, N. "Lift." In *Fourth International Workshop on Software Specification and Design*, page x, Washington, D.C.:Computer Society Press of the IEEE, April 3-4 1987.
12. Elefante, Donald M. *Knowledge-Based Software Assistant Technology Transfer Consortium: Status Report No. 1*. Technical Report, Rome Air Development Center, 12 September 1990.
13. Fischer, Charles N. and Jr. Richard J. LeBlanc. *Crafting a Compiler*. Menlo Park:Benjamin Cummings, 1988.
14. Fitter, M. and T. R. G. Green. "When Do Diagrams Make Good Computer Languages?," *International Journal of Man-Machine Studies*, pages 235-261 (November 1979).
15. Glinert, Ephraim P. "Towards Software Metrics for Visual Programming," *International Journal of Man-Machine Studies*, *30*(4):425-445 (April 1989).
16. Ingalls, Dan, et al. "Fabrik: A Visual Programming Environment." In *OOPSLA '88 Proceedings*, pages 176-190.

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE THE DEVELOPMENT OF A GRAPHICAL NOTATION FOR THE FORMAL SPECIFICATION OF SOFTWARE	5. FUNDING NUMBERS
6. AUTHOR(S) Gene A. Place, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology WPAFB, OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-10
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This research developed a graphical formal specification language based on the Refine wide-spectrum language using a graph-based iconic representation to present formal specifications in a format that is much easier to create and manipulate than the equivalent textual formal specifications. The development of the graphical formal specification language proceeded in two steps: the Refine language was first decomposed into its primitive data types and operations, then iconic representations were developed for each of the primitive data types and operations that were identified. In addition to the development of a graphical formal specification language, this research also proposed a number of heuristics to convert existing graphical program design notations, such as data flow diagrams, state transition diagrams, and entity relationship diagrams, into graphical formal specification syntax. The architecture for a graphical formal specification environment was proposed in this research that would support the creation and revision of formal specifications using only their graphical representations.

14. SUBJECT TERMS software engineering, programming languages, specifications, graphics, mathematical logic	15. NUMBER OF PAGES 259
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL