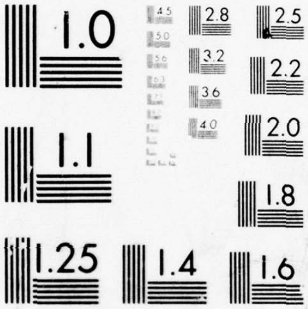
AD-A051 672 MITRE CORP BEDFORD MASS F/G 9/2
PROGRAM DESIGN LANGUAGES-AN INTRODUCTION.(U)
JAN 78 L L CHENG F19628-77-C-0001
UNCLASSIFIED MTR-3446 ESD-TR-77-324 NL

1 OF 2
AD
A051672


1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A051672

ESD-TR-77-324 ② MTR-3446

# PROGRAM DESIGN LANGUAGES
# AN INTRODUCTION

BY LORNA L. CHENG

JANUARY 1978

Prepared for

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

AD No. ___
DDC FILE COPY

DDC
RECEIVED
MAR 22 1978
A

Project No. 522M
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-77-C-0001

Approved for public release;
distribution unlimited.

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

William E Rzepka

WILLIAM E. RZEPKA
Project Engineer

John L McNamara

JOHN L. MCNAMARA
Ch, Info Mgt Sciences Section
Info Processing Branch

FOR THE COMMANDER

Alan R Barnum

ALAN R. BARNUM
Asst Chief, Info Sciences Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

## REPORT DOCUMENTATION PAGE

(19)

READ INSTRUCTIONS BEFORE COMPLETING FORM

|  |  |  |
|---|---|---|
| 1. REPORT NUMBER<br>(18) ESD-TR-77-324 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>(6) PROGRAM DESIGN LANGUAGES — AN INTRODUCTION, |  | 5. TYPE OF REPORT & PERIOD COVERED<br>(9) Technical rept., |
|  |  | 6. PERFORMING ORG. REPORT NUMBER<br>(14) MTR-3446 |
| 7. AUTHOR(s)<br>L. L. Cheng (10) Loina L. Cheng |  | 8. CONTRACT OR GRANT NUMBER(s)<br>(15) F19628-77-C-0001 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>The MITRE Corporation<br>Box 208<br>Bedford, MA 01730 |  | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>Project No. 522M |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Electronic Systems Division (AFSC)<br>Hanscom Air Force Base, MA 01731 |  | 12. REPORT DATE<br>(11) JANUARY 1978 |
|  |  | 13. NUMBER OF PAGES<br>123 (12) 127p. |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* |  | 15. SECURITY CLASS. *(of this report)*<br>UNCLASSIFIED |
|  |  | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

COMPUTER PROGRAMS
HIGH LEVEL PROGRAMMING LANGUAGES
PREPROCESSORS
PROGRAM DESIGN LANGUAGES (PDLs)

(over)

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report is an introduction to program design languages (PDLs). A definition is given, and general functions of PDLs are discussed. A set of characteristics is defined to (1) indicate types of design information representable in PDLs and (2) to provide a framework for comparing various PDLs. Each PDL reviewed in a brief survey is discussed within the framework of these characteristics. The set → next page

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

235 050

TC

19. Key Words (Continued)

REQUIREMENTS LANGUAGES
SOFTWARE DESIGN LANGUAGE
SOFTWARE SPECIFICATION LANGUAGE
STRUCTURED PROGRAMMING

20. Abstract (Continued)

cont.

of characteristics set forth when expanded, curtailed, or amended according to special needs may constitute the specification for a PDL desirable for a particular class of application.

## ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project No. 522M. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

The author wishes to thank three people who have reviewed drafts of this paper and gave helpful, constructive comments: Marlene Hazle, Dave James, and William Rzepka of RADC, project officer of this project. Special thanks are due to Marlene Hazle for our many discussions that helped shape the ideas presented in this report. Thanks are also due to Jane McCarthy for maintaining drafts of this report on ATMS and for her help in some editorial changes.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION ..........
..........
BY ..........
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Preceding Page Blank - Filmed

## TABLE OF CONTENTS

Page

SECTION I PROGRAM DESIGN LANGUAGES 7

INTRODUCTION 7

- Techniques and Notations 7
- Design 8
- Definition of a Program Design Language 8

PURPOSE OF PROGRAM DESIGN LANGUAGES 9

- Basic Functions 9
  - To Represent Design 9
  - To Record Design Decisions 9
  - To Enforce Built-in Rules 9
  - To Communicate 10
- Auxiliary Functions 10

DESIGN VERSUS REQUIREMENTS 11

DESIGN VERSUS CODING 11

SECTION II CHARACTERISTICS OF PROGRAM DESIGN LANGUAGES 13

BASIC CONTENT 16

- External Interfaces 17
- Static Structure 17
  - Composition and Organization 18
  - Calling Dependencies and Sequence 18
  - Data Dependencies 18
  - Ownership of Resources 19
- Dynamic Structure 19
- Data 20
  - Organization or Structure 21
  - Scope or Access 21
  - Operations 21
  - Size 21
  - Flow 22
  - Derivation 22
- Control Flow 22
- Levels of Abstraction 22

TABLE OF CONTENTS (Continued)

| | | Page |
|---|---|---|
| | AUXILIARY CONTENT | 23 |
| | Decision Details | 23 |
| | Error Handling | 24 |
| | Performance Estimates | 24 |
| | Verification Information | 25 |
| | CONSTRAINTS | 25 |
| | Hierarchy | 25 |
| | Construction Rules | 26 |
| | FORMAT | 26 |
| | APPLICATION CONSIDERATIONS | 26 |
| | Size of the Language | 27 |
| | Syntax | 27 |
| | Similarity to Familiar Languages | 27 |
| | Orientation toward Implementation Languages | 27 |
| | Ease of Transformation into Code | 27 |
| | Types of Applications Suitable | 27 |
| | Status | 28 |
| | AUTOMATIC TOOL SUPPORT | 28 |
| SECTION III | SURVEY OF PROGRAM DESIGN LANGUAGES | 29 |
| | CAINE, FARBER & GORDON (CFG PDL) | 30 |
| | FLOWCHARTS | 36 |
| | HIERARCHY PLUS INPUT-PROCESS-OUTPUT (HIPO) | 41 |
| | HIGHER ORDER SOFTWARE SPECIFICATION LANGUAGE AXES | 47 |
| | IBM PDL | 53 |
| | MODULE INTERCONNECTION LANGUAGE (MIL75) | 59 |
| | P-NOTATION AND V-NOTATION | 66 |

TABLE OF CONTENTS (Concluded)

| | | Page |
|---|---|---|
| | PROBLEM STATEMENT LANGUAGE (PSL) | 72 |
| | PROCESS DESIGN LANGUAGE (PDL2) | 78 |
| | PROGRAM DESIGN AND DOCUMENTATION LANGUAGE (PDDL) | 84 |
| | SOFTWARE DESIGN LANGUAGE (SDL-1) | 90 |
| | SOFTWARE SPECIFICATION LANGUAGE (SSL) | 96 |
| | STRUCTURED CHARTS | 102 |
| | UMTA SPECIFICATION LANGUAGE (USL) | 108 |
| SECTION IV | CONCLUSIONS | 114 |
| | SELECTION | 114 |
| | OBSERVATIONS | 117 |
| REFERENCES | | 119 |

## LIST OF TABLES

| Table No. | | Page |
|---|---|---|
| I | Set of Characteristics of Program Design Languages | 14 |
| II | Characteristics of CFG PDL | 31 |
| III | Characteristics of Flowcharts | 37 |
| IV | Characteristics of HIPO | 42 |
| V | Characteristics of HOS AXES | 49 |
| VI | Characteristics of IBM PDL | 54 |
| VII | Characteristics of MIL75 | 61 |
| VIII | Characteristics of P-NOTATION and V-NOTATION | 67 |
| IX | Characteristics of PSL | 73 |
| X | Characteristics of PDL2 | 79 |
| XI | Characteristics of PDDL | 85 |
| XII | Characteristics of SDL-1 | 91 |
| XIII | Characteristics of SSL | 97 |
| XIV | Characteristics of Structured Charts | 103 |
| XV | Characteristics of USL | 109 |
| XVI | Summary of Some Characteristics of PDL's | 115 |

# SECTION I

# PROGRAM DESIGN LANGUAGES

## INTRODUCTION

This paper is intended to be a general introduction to program design languages (PDL's). The need for program design languages and their functions will be discussed first. A brief literature survey has been performed and a set of characteristics of PDL's has been defined (see Section II). Individual PDL's are examined within the framework of these characteristics. No actual application of these PDL's was performed for the purpose of this task, although some prior experience, directly or indirectly, has contributed to the results presented in Section III. It is hoped that the reader will obtain a general idea of the purposes of PDL's, some important aspects of their properties, and the types of PDL's currently available. The set of characteristics set forth when expanded, curtailed, or amended according to special needs, can evolve into the requirements for a PDL desirable for a particular situation.

### Techniques and Notations

Concomitant with the advent of recent software engineering techniques is the emergence of many proposed or implemented languages to facilitate the application of these techniques. Recent software engineering techniques are methods and procedures all aimed at improving the software development process and its final product, whether it be to achieve higher reliability, to provide a sounder basis for testing and verification, to ensure that computer programs are more comprehensible for maintenance purposes, to provide better communication among the development team, or to enable more intelligent management control of resources. It would be belaboring the point to restress the merits or the importance of these goals. Suffice it to say that awareness of the need for a better software engineering discipline permeates the research, educational, governmental and commercial sectors of the computer field.

Some of the new techniques embody new concepts or new approaches to various stages of software development. Through their application it is discovered that the old notations no longer suffice and new notations are required to reflect the new concepts involved. In the requirements area, the realization that inconsistencies in requirements can be sieved out early to reduce the number of errors made later in the design and coding stages has led to proposed requirements languages as a more rigorous means of specifying user requirements [Teic74]. Another example lies in the effects of structured programming. Existing programming languages have been found to be inadequate in support of structured programming principles. Numerous preprocessors were therefore written to extend existing languages, to restrict the use of unwanted constructs, and to enforce certain rules. For FORTRAN alone, one survey shows the existence of 56 preprocessors [Reif75]; another covers 20 preprocessors [Horo75], some of which duplicate the first survey. Structured programming principles have affected the design of languages such as BLISS and PASCAL. As a side effect of structured programming, a new language CLU [Lisk74] has been proposed to handle the abstraction of data, a capability not normally found within the confines of conventional programming languages.

## Design

Of the areas addressed by recent software engineering techniques, the requirements and design areas have been receiving increasing attention. This is a result of the continuing emphasis on eliminating errors as early as possible in the software development process. Work in the requirements area is relatively new and fewer in number, but much has been said about design in the last few years. Many design methods, sometimes mistakenly called methodologies (a collection of methods with rules for applying them), have been proposed to apply new concepts and consequently brought about new languages or other types of notations to supplement the design activity. These languages are labeled by many different names, such as program design language, software specification language, high level programming language, and software design language. This paper is concerned with those languages that can be used to aid software design and has taken a rather broad view of the term 'program design languages.'

## Definition of a Program Design Language

Within the context of this paper, a Program Design Language (PDL) refers to any notation, whether a textual language or a visual chart, that serves as:

1. A form of representation necessary to capture the design of a software system as it evolves so that the current status of the design is reflected in tangible form (requisite); and

2. A documentation aid to record other aspects of software design including verification, performance, trade-offs between alternatives, and rationale of decisions (optional).

P. Freeman views the design activity as one of "fitting form to function," and distinguishes between representation and documentation: "A design is a representation of an object; documentation describes some aspects of an object." [Free76]

## PURPOSE OF PROGRAM DESIGN LANGUAGES

### Basic Functions

#### To Represent Design

The primary purpose of a program design language is to represent the design of a software system at various stages of development. Considerations should include global issues such as overall structure and module interdependency as well as local issues such as control flow within a module and internal data. Not all this information is available at once. As the design evolves, more details are known to better depict what the ultimate system is like. For a PDL to be useful throughout the design stage, its scope must encompass the abstract as well as the concrete, and representation of a system should progress from general global concepts through varying levels of abstraction all the way to detailed specification of each module and data structure until the whole system is well defined for coding in a programming language.

#### To Record Design Decisions

Another purpose of PDL's is to record design decisions made. Too often design decisions made are implicitly embedded in the code. A PDL will hopefully record some design decisions more explicitly, especially when it is used conscientiously at various stages throughout the gradual evolvement of a design.

#### To Enforce Built-in Rules

The relation between design techniques and program design languages has already been discussed. There is usually an underlying philosophy in the design of any language. The set of constructs available in a language is significant both in the capabilities provided and in the capabilities not provided. Program design languages can be used to: (1) promote certain concepts in software design, such as viewing a system in varying levels of abstraction; (2) enforce certain approaches such as constructing software in a top-down hierarchical manner; and (3) ensure that the ultimate design obeys certain rules that, for example, apply to the control of modules over one another and the scope of data structures. The purpose of such enforcement is usually to allow for better management of the complexity of a system and to promote correctness in the design.

### To Communicate

Program design languages can serve as an important communication tool. In the case where design of a system is undertaken by more than one person, PDL's act as a communication tool among the designers by representing the current design in tangible form.

The design, as represented by a PDL, serves as a means of communication between requirement definers and designers in the verification of design against requirements. Requirement definers can be an acquiring agency or a user of the target system, or both.

On the other side of the spectrum, PDL's serve as a means of communication between designers and coders, the ultimate form of the design being the specifications for coding.

PDL's facilitate the generation of test plans by communicating the design from designers to people responsible for testing the ultimate system. PDL's also communicate the design to maintenance personnel.

### Auxiliary Functions

The author considers the above functions to be primary functions that a PDL must fulfill. Other uses have been suggested for PDL's, namely: (1) to facilitate the prediction of system performance [Grah73], and (2) to include test plans and procedures in design specifications [Reif76]. Though worthwhile goals, they are not primary functions of a PDL, especially when the feasibility of incorporating such functions effectively into a PDL has not yet been proven.

## DESIGN VERSUS REQUIREMENTS

To understand in greater detail the functions of program design languages, the design activity has to be considered in proper perspective in the software life cycle so that it can be contrasted with requirements definition on the one hand and coding on the other. Characteristics of PDL's can then be distinguished from those of requirement languages and programming languages.

Theoretically, requirements should specify needs of the user independently of implementation methods. For instance, data in the requirements stage are generally dealt with only in terms of the information transferred. In some stage of design, the format of the data and the internal representation of the data in the computer become of interest. In communication applications, requirements may specify that incoming messages be accepted and routed to the correct destination according to information contained in a header within the message. A certain level of performance may be required, but the routing algorithm is not specified. Design will then specify which part of the message (e.g., bits 1 to 10) will contain the header and how the routing is accomplished. In general, requirements are concerned with what is to be done and acceptable performance levels; design is concerned with how things are done.

In practical situations, however, there are times when some design will be specified as a requirement, although this should always be done consciously and with good reason. In cases where the software to be specified is planned to augment an existing system, the requirements may be especially stringent to ensure correct interface. For some reason, the algorithm may have to be specified. Practices within the military and industry have borne out the fact that these deviations do exist. As a result, it is difficult to draw a sharp line between requirements and design. Rather, each successive description of the system can be viewed as a specification for the next lower level and a design for the previous level, progressing from an abstract representation to a concrete set of code in a programming language executable on a computer.

## DESIGN VERSUS CODING

On the other end of the spectrum, the role of PDL's is matched against that of high level programming languages. Translating user requirements into an overall design of the system is called global design. At this stage, most system functions and data remain in abstract form. As each successive step refines the description of the system, more details are specified and design activities close to the coding stage are referred to as detailed design. Design is said to be complete when the system can be coded from it. Again, it is not always easy to draw the line between detailed design and coding. When pseudo-code is used to represent design, partial coding has already taken place.

Judging from the relations among requirements, design and coding, global design is very different from detailed design. Program design languages have to accommodate a wide spectrum of levels of system description. The scope of the survey of existing PDL's (Section III of this paper) has been extended to include some requirements languages for the precise reason that their capabilities seem applicable to global design as well. Other PDL's surveyed are actually extensions of high level programming languages and naturally address issues closer to detailed design.

# SECTION II

## CHARACTERISTICS OF PROGRAM DESIGN LANGUAGES

In this section, some characteristics of program design languages will be discussed. This set of characteristics is limited by the effort expended for the task in the sense that it may not be complete and that some characteristics could be delved into more thoroughly. Table I contains a list of the characteristics that will be discussed. Each characteristic will be defined in the text of this section, and its related issues will be addressed. Specific properties of individual PDL's will be given in Section III within the framework of the characteristics discussed here.

The characteristics have been constructed and presented here with four purposes in mind:

1. To point out some directly observable properties of PDL's. Referring to Table I, characteristics such as Basic Content, Constraints, Format, and Automatic Tool Support are typically observable characteristics.

2. To suggest a few more commonly agreed upon elements to be included in a PDL, if possible. These are reflected by the Basic and Auxiliary Contents.

3. To raise some issues that have to be considered in selecting a PDL for use - Application Considerations. The size of the language, the syntax, and the similarity a PDL bears towards familiar languages all affect the amount of learning required of a designer with a particular background for application of a PDL. These characteristics are necessarily less objective than the directly observable characteristics.

4. To provide a framework for comparing various PDL's. Properties of the PDL's surveyed will be highlighted within the framework of these characteristics.

In characterizing the attributes of a PDL, care should be taken to discriminate between features of the language and features of automatic tools that support it. Automatic tools are usually developed to process the language and produce some analysis of the information represented by the language. It does not mean, however,

## Table I

### Set of Characteristics of Program Design Languages

- BASIC CONTENT
  - External Interfaces
  - Static Structure
    - Composition and Organization
    - Calling Dependencies and Sequence
    - Data Dependencies
    - Ownership of Resources
  - Dynamic Structure
  - Data
    - Organization or Structure
    - Scope or Access
    - Operations
    - Size
    - Flow
    - Derivation
  - Control Flow
  - Levels of Abstraction

- AUXILIARY CONTENT
  - Decision Details
    - Algorithms
    - Solution
    - Trade-off between Alternatives
    - Rationale for Decisions
  - Error Handling
  - Performance Estimates
  - Verification Information

- CONSTRAINTS
  - Hierarchy
  - Construction Rules

- FORMAT
  - Input Format
  - Internal Format
  - Output Format

Table I (Concluded)

APPLICATION CONSIDERATIONS
- Size of the Language
- Syntax
- Similarity to Familiar Languages
- Orientation toward Implementation Languages
- Ease of Transformation into Code
- Types of Applications Suitable
- Status

AUTOMATIC TOOL SUPPORT

that the PDL's currently not supported by automated analysis are not amenable to such support in the future.

In characterizing the information representable in a PDL (content) in this report, a PDL is considered independently of support tools as far as possible. In general, support tools provide analysis of information already in the PDL; rarely do they generate new information for which no basis is provided in the PDL, although they may make such information more apparent. A type of information is included in the content of a PDL if that information is representable in the PDL, regardless of how obscure it may be. Information is excluded from the content if the PDL is incapable of expressing such information. An attempt will also be made to indicate information obtainable from automatic tools to portray a PDL more completely.

Some PDL's have been developed in conjunction with a design method to support a particular design technique or at least with some underlying philosophy. Such techniques will be frequently referred to in the discussion of PDL characteristics, especially implicit and explicit construction rules. Though it is beyond the scope of this paper to define all these techniques, a list of design techniques, some more commonly known than others, is presented here together with associated references for readers who wish to pursue these concepts further.

- Levels of abstraction [Dijk68]
- Top-down design [Mill71]
- Structured programming [McCr73, McGo75]
- Parnas black-box approach and module specification [Parn72a, Parn72b]
- Stepwise refinement [Wirt71]
- Structured design [Stev74]
- Higher Order Software (HOS) [Hami76]
- Data abstraction [Lisk74]
- WELLMADE methodology [Boyd76]
- Information Automat [Wils75]
- Structured Analysis Design Technique [Soft76]
- Programming-in-the-large [DeRe76]
- Process design methodology [Gaul76]

## BASIC CONTENT

Foremost in the characteristics of any language is its ability to handle information. The content of a PDL refers to the information representable in that PDL. In this paper, external interfaces, static and dynamic structure, data, logic or control flow, and levels of abstraction are considered basic elements of design to be represented in a PDL. Other information, such as error handling, performance estimates, and verification information, is discussed separately under Auxiliary Content.

A PDL is said to possess a certain item in its information content if the item is part of the repertoire of that PDL in any form. In other words, substance is judged independently of format, and the same information content representable in two PDL's may be very apparent in one and very obscure in the other.

## External Interfaces

External interfaces define the external boundaries of the system being designed (the target system). A system can be viewed as an entity performing a certain function; it accepts certain inputs and transforms them into a set of outputs. From the point of view of the ultimate user of the system under development, this characteristic is most important. Without the definition of external interfaces, it is not known what information is required for the system to function and what information the system can generate. It can perhaps be argued that the distinction of what relates the system to the external environment is more crucial to requirements specification than design representation. External interfaces are indeed part of the requirements of the system, in that they represent the goals which the design should fulfill. If such interfaces are made more apparent in the design representation, they will not only delineate the external boundaries of the system but also make it easier to ensure that the design satisfies the requirements. If design is viewed as a more specific description of a system than requirements, it sometimes helps to clarify what the requirements are.

Some examples of information contained in external interfaces are identification of people or organizations who will use the target system, input provided by users, and reports generated by the system.

## Static Structure

The static structure describes the composition of a system in terms of its constituent parts and how they relate to one another.

### Composition and Organization

A PDL should be able to describe the static structure at various levels from a general overview to the smallest unit. Composition of a target system may be defined in terms of many units: a subsystem, which is a logical collection of related modules; a module, which we define as a conceptual unit performing a function that may eventually be implemented as one or more procedures or as a part of a procedure; or a procedure as understood in a programming language. This is related to the concept of levels of abstraction which will be discussed separately. A correspondence may also be established between a unit in the composition and that part of the requirements it satisfies.

Each constituent part may be successively decomposed into its own constituents until a desired level of refinement is achieved, showing how each part fits into the whole structure of the system. This constitutes organization information.

### Calling Dependencies and Sequence

Besides the organization of functional units of a system, there also exist relationships among these units. One important relationship is the dependency *of invocation* - which other modules does a module call on to perform its function and the order of such invocation. This characteristic is only concerned with the static aspect of the calling relationship - what modules a module may require to help it do its job. In real execution, not all these modules may be actually invoked, depending on which paths of the control logic are executed.

### Data Dependencies

Another important relationship is that of data dependencies among the modules, i.e., the information passed from module to module or shared among modules. This is an important aspect of the structure of a system because it represents the interfaces among the various parts of the system. The data that are shared may eventually be implemented in the coded program as global variables, as parameters passed from module to module, or by whatever mechanism the selected programming language has to offer. If a PDL is to be useful at higher levels of design, it must be able to represent data dependencies merely as information shared by modules independently of the actual implementation mechanism.

Sometimes this characteristic is only partially fulfilled in that the complete data connections among modules are not described. Rather, each module is treated in isolation, naming the information it requires (input) and the information it provides (output), without identifying the origin of inputs or the destination of outputs. Thus, only the data requirement of each individual module is described but not the data relationship among modules.

### Ownership of Resources

In contrast to data that are shared, there are data that need to be solely owned by one module. The idea is that information is hidden from other modules that have no need to know and therefore have no need to access a particular data item. Interfaces between modules exist only when explicitly stated as data dependencies. The idea of information hiding is embodied in Parnas' 'black-box' modularity concept [Parn72a], Dijkstra's level of abstraction [Dijk68], the strength of modules in Structured Design [Stev74], and other design approaches. Privately owned resources are commonly manifested as internal variables.

## Dynamic Structure

The dynamic structure of a system models the behavior of a system as it executes in a real environment. It deals with the conditions under which modules are activated, the events that lead to such activation, the order in which events occur, and how these events are related to the data passing through the system. The design of real-time systems exert a greater demand on PDL's in this respect than business information systems because the order and timing of events are critical in real-time systems. There may be a need to indicate that certain activities take place concurrently. Or, a task in the target system may have to be interrupted by tasks of higher priority.

Most currently available PDL's are deficient on this point. Most of them do not model the dynamic behavior of a system at all, and those that do, do so partially. One PDL, for instance, has the concepts of 'Events', 'Processes' and a relation between them, namely, 'Triggers'. 'Processes' are also related to the data passing through them. An 'Event' can 'Trigger' a 'Process', but there is no way to link this to the data involved in the happening of an 'Event'. Data, therefore, are related only to the static picture, not to the dynamic aspects of a system. Another PDL has the ability to specify that a series of events can take place in random order, and it is being extended to admit expression of concurrent processes. Yet another PDL describes real-time activities in terms of the reception and transmittal of messages.

Each of these three PDL's provides partial means of describing the dynamic behavior of a system, and approaches it in a different way. The fact that no PDL provides a satisfactory representation of dynamic structure (though its importance is not disputed) probably indicates that no one yet knows what constitutes a good description of dynamic behavior. It would be interesting to monitor future work in this area to see if significant improvements are made.

## Data

Data constitute the information passing through the system. As in the representation of the composition of static structure, the representation of data should be capable of reflecting different levels of detail, if recent design techniques such as levels of abstraction are to be applied. A data item is represented at early levels of design as abstract information so that decisions of actual means of implementation can be deferred till later stages of design when a more intelligent choice can perhaps be made to accommodate the combined needs of the higher levels. In other words, data can be represented in both abstract and concrete form. Abstract data representation has emerged as quite a prominent issue since the advent of structured programming.

Abstract representation of data is generally accomplished in one of three ways. The first is by allowing the incomplete definition of a data item at higher levels, with the details to be filled in later. Typically the logical structure of a data item is defined first and the physical representation of the data in the computer is specified later as the design decisions concerning it are made. The second way is by incorporating abstract concepts such as classes and sets into the language itself. The third way is by allowing user-defined data structures. A particular application can conceivably call for concepts of information not readily representable by a fixed set of data types afforded by a PDL. A commonly used example of such a data concept not representable by data types usually found in programming languages is that of a 'stack' - a sequence of elements that can be added to in one direction and deleted from in the other direction. By permitting users of a PDL to define their own data structures, the operations allowed on them, and accessibility rights by modules, a PDL in essence gives a user the freedom of abstracting information in a form natural to it. This differentiation between user-defined data and a fixed set of pre-defined data applies to the aspects of structures, scope or access, and operations discussed below.

## Organization or Structure

This refers to the composition of data and how the constituent parts relate to form the whole. Structure is divided into logical or physical structure. PDL's capable of logical representation are more advanced in the abstraction of data than those that are incapable. The logical structure of a data item is its conceptual organization. A typical example is that of a set of related information. Physical structure is the actual physical form in which a data item is recorded. The set cited above can be a file consisting of similar records, or a data structure as in COBOL and PL/I, or an array (or any other data type in a programming language) that usually implies a fixed means of implementation by the compiler.

A PDL may allow a user the flexibility of defining his own data type, as was discussed above under Data.

## Scope or Access

In accordance with the ownership of resources as discussed above under static structure, the limitation of accessibility of data by modules restricts the interface among modules. The right of access is sometimes implied in the scope of modules that contain them as in the block structure of PL/I. It can also be explicit, which is preferable in design as it renders interfaces among modules more conspicuous.

## Operations

'Operations' refers to manipulations that can be performed on data items such as addition, subtraction, multiplication, division performed on integers and real numbers. 'Read' is an operation defined for records and files. An example of a user-defined operation performable on a 'stack' as introduced above is that of 'push' which adds an element to the top of the stack and deletes the element at the bottom.

## Size

Size refers to the cardinality of the data.

### Flow

Data flow depicts the sequence in which data pass through the modules. Note that data flow also indirectly shows the data dependencies among modules. The difference is that data dependency of modules is from the module point of view and does not depict the sequence, whereas data flow is from the data point of view.

### Derivation

This refers to the algorithm by which data is derived, or to a lesser degree of detail, the data items on which the value of a particular data item depends.

### Control Flow

Control flow, sometimes known as logic, of a module specifies the transfers of control within the module. In cases where a selection of different actions is involved, control flow may also specify the state of variables that dictate the choice of a particular action.

Control flow may be represented at various levels of system design. The flow within a higher level module in overall design illustrates the sequence in which lower level modules are executed. Modern software engineering techniques call for the use of only several control flow constructs - that of sequence, alternation or selection, and iteration.

### Levels of Abstraction

The idea of viewing a system in varying levels of detail in the design process is well accepted as a good means of controlling the complexity of a system. Parts of a system are first considered as abstract entities; decisions on details are postponed until they become necessary. As decisions are made, the system description is refined further and further until all important implementation issues are decided and coding can take place. Most of the modern software engineering techniques listed at the beginning of this section have adopted this type of concept. For a PDL to support any of these software engineering techniques, it must be able to represent a software system description at different levels of abstraction.

Abstraction in this case applies to both functional modules and data items. Part of the abstraction issue has already been discussed separately under static structure and data. This characteristic is not independent of some of the others because the ability of a PDL to represent abstraction is partially reflected in the way it represents functional modules and data items. Modularity in the Parnas sense, the 'black-box' approach, is a means of abstraction where the mechanism internal to the module is not made known to the rest of the system. Another mechanism of abstraction is that of user-defined data types and their corresponding operations. Though dependent on other characteristics, this issue of abstraction is listed separately so that all information on abstraction capabilities can be gathered in one place.

It has also been said that representation of a design affects thinking [Free76]. The more naturally a PDL can express concepts pertaining to a system, the more effective that PDL is, especially in its abstraction capabilities. A concept that is bent and twisted to fit artificially into a language ultimately finds expression in an unnatural representation and might affect adversely subsequent design decisions.

## AUXILIARY CONTENT

### Decision Details

Design is a process whereby decisions are made to gradually refine a system description from requirements toward coded programs. There are aspects of these decisions that are of interest to someone inspecting a design representation.

- Algorithms - How the processing is done; e.g., smoothing algorithms for tracking functions. Algorithms are sometimes represented by derivation dependencies, but for those that involve many variables, the general algorithm can be stated in a clearer and more obvious fashion.

- Solution - As in decision tables. This represents the designer's solution to problems posed by requirements.

- Trade-off between alternatives - If for some reason a decision made has to be discarded, it would be helpful to know what alternatives exist and their pros and cons without retracing all the reasoning in arriving at them.
- Rationale for decisions - Why a decision is made. This states explicitly the goals of the present design and will be useful in further trade-off decisions.

Of the four types of information listed above, the first two may actually exist as part of requirements or design, although theoretically a statement of requirements should avoid specifying more details than necessary. Furthermore, note that the first two are representation issues, whereas the last two are documentation issues in that they describe things about a system, not what is in them.

Most of the PDL's surveyed relegate this function to supplementary free-form documentation.

## Error Handling

One aspect of systems design is how the system will behave in case of any form of malfunction. Malfunction causes error conditions and may take place in the environment, operating system, hardware components, input data or software component parts. The design of a system is not complete until it has specified what error conditions it has taken into account and what actions the system will take at the occurrence of such conditions. Sometimes error handling is specified as part of the requirements.

## Performance Estimates

This is information concerning the resource usage, execution time, response time, etc., of the system being developed. Current PDL's are not too sophisticated in this capability. Many do not address this issue at all. The few that do, do so by only allowing estimates of the designer to be entered into the language representation so that they can be processed by an automated tool. Thus, the estimates are no better than subjective conjectures by the designer and the supporting tool performs only a bookkeeping function.

### Verification Information

Verification information is any information that helps to verify the internal correctness of a program or to verify that the design satisfies the requirements. Current PDL's are not too sophisticated in this area, although awareness of the importance of verification is rising. Verification information usually takes the form of assertion statements placed at strategic places to ensure certain conditions exist at those points. There may be, however, much that is built into a PDL or a design technique that indirectly helps verification. Making interfaces among modules explicit, for instance, facilitates the verification process. Restriction of control flow to a few basic structures with no uncontrolled transfers also prepares the foundations for eventual verification. The very nature of a PDL in formalizing design representation renders verification against requirements possible in the presence of a formal requirements language.

Verification information may also include information useful for the generation of test plans such as the order of implementation of modules, modules critical to testing, and the need for driver modules not part of the system but required for testing.

## CONSTRAINTS

A PDL, because of its underlying philosophy, imposes certain constraints on the design of software. The set of keywords of a PDL is a form of constraint in that it represents the objects and concepts allowable in the description of a system. The format, too, may pose a restriction, e.g., by forcing the modules of a system to be organized as a tree-structure. The structure of the language may imply the automatic application of certain design rules. In addition, explicit rules may be stipulated for use with the language.

### Hierarchy

Some PDL's require that the design of a system be described in hierarchical levels. Others further demand that the resulting structure of the system be expressed as a tree structure. Usage of such PDL's will naturally influence the designer to proceed in a particular fashion and his perspective of the system will also be affected as he complies with the philosophy of the PDL to stay within its bounds.

As one speaks of the hierarchical nature of certain PDL's, extreme care should be taken to isolate to what the restraint is applied. The hierarchical restriction may apply to data or the dynamic sequence of execution of modules. It is insufficient merely to say that a PDL imposes a hierarchical constraint on the design.

### Construction Rules

Construction rules relate a PDL to its accompanying methodology, if any. Those PDL's that have been developed as part of a methodology are expected to be used with construction rules that reflect the philosophy of the methodology. Not only does the set of constructs available in a PDL represent the capabilities provided, but the purposeful omission of some constructs probably indicates objection to those capabilities.

Construction rules can be either implicit - embedded in the available constructs or the format - or explicit - stated in the form of rules or axious.

## FORMAT

The format of a PDL can be textual or graphical. Some PDL's are supported by automatic tools that are capable of parsing design representations in those languages, maybe even store the information in a data base, manipulate the information, and reproduce the design and its related information in formats more eloquent than the format of the original. A prime example is when a designer expresses a design in a textual language, but can avail himself of reports in pictorial as well as textual form.

For this reason, format should be differentiated into input format as provided by the designer, internal format if stored in a data base, and output format as made available to the designer. In many cases, the internal format does not exist and the input format is the same as the output format. Note that this characteristic is discussed in conjunction with characteristics of the support tool.

## APPLICATION CONSIDERATIONS

An intuitive feeling of the learning required to apply a PDL and how applicable it is to the situation on hand may be misleading for it is composed of several factors that have to be considered together.

### Size of the Language

The size of current PDL's ranges from only several constructs to over two hundred keywords.

### Syntax

The more complex the syntax of a PDL the more difficult it is to learn. However, if the syntax is well defined, once learned it will be easy to apply. This factor has to be considered in parallel with the size of the language because a PDL can have a complex syntax and yet contain very few constructs. The difficulty of learning the syntax is then offset by the small size.

### Similarity to Familiar Languages

Similarity to English is obviously an advantage to English-speaking software designers as far as learning is concerned. So is similarity to familiar programming languages such as COBOL, PL/I, and PASCAL. Of course, how much an advantage such similarities are varies with the background of the designer and other parties using the PDL. Similarity to PASCAL is obviously not much use to a designer not familiar with PASCAL.

### Orientation toward Implementation Language

A few PDL's are designed for eventual implementation in a particular programming language. In such cases, the PDL usually bears a similarity to the programming language of choice so that a design can be smoothly transformed into code without any major effort.

### Ease of Transformation into Code

PDL's not designed for implementation in specific programming languages may also be easily transformed into code. This characteristic may help to determine whether a PDL is suitable for use in global design or detailed design.

### Types of Applications Suitable

Some PDL's are specially designed for use in a particular type of application, such as real-time systems. A real-time system has the following attributes: (1) requires the ability to handle interrupts, (2) interfaces with non-standard equipment such as sensors, (3) the sequence of events is critical, and (4) has more stringent performance requirements. Another example is that of systems with concurrent processes. A PDL will be suitable for such applications only if it can express naturally and efficiently the concepts pertaining to these applications.

### Status

The status of a PDL indicates how ready it is for use - whether it is well defined or only proposed; if there is adequate documentation; if support tools have been implemented; if there has been any experience in its application; if it is in a stable state or is still being revised.

## AUTOMATIC TOOL SUPPORT

PDL's supported by automated tools such as parsers and analyzers will be pointed out. The types of analysis will be highlighted.

# SECTION III

## SURVEY OF PROGRAM DESIGN LANGUAGES

A brief survey of program design languages was undertaken, and the results are presented in this section within the framework of the characteristics defined in the previous section. The PDL's surveyed fall within a rather wide scope in that some of them may not have been originally intended to be a PDL but have enough of the characteristics of a PDL to be useful as one. The PDL's surveyed also span a wide spectrum of functions: some address global design issues and others address detailed design considerations in preparation for coding. They also range from PDL's with a very strict syntax and semantics to those that resemble free-form English.

No actual application of these PDL's has been performed for the purposes of this survey. The information reported here is based mainly on the literature, personal prior experience of the author, contact with the developers of the PDL, and the indirect experience of other people who have used the PDL's. In certain cases, knowledge is limited by the documentation available.

As the survey was first conducted, the points of interest were special terminology, typical properties, design activities addressed, information conveyed, applicable problems and environment. Much of this information has since been distilled and incorporated into the set of PDL characteristics in Section II. For each PDL surveyed, a short account of its background will be given first, and where information is available, a table of its characteristics will be provided. Characteristics that cannot be determined at this point for lack of information or documentation are left blank.

## CAINE, FARBER & GORDON PDL (CFG PDL)

The CFG PDL was developed by Caine, Farber & Gordon, Inc., and has been in extensive use there since 1973 [Cain75]. It is designed for use with structured programming and top-down design methods.

It is referred to as a form of "structured English" since it uses the vocabulary of English and a syntax similar to that of a programming language. Free-form English statements are also allowed.

The main design body consists of "segments", some of which are congregated to form "groups". There are Text Segments, Flow Segments, and Data Segments. A Flow Segment expresses the logic of a design and consists of a parameter list, labels, comments, free-form statements, and special statements (IF, ELSEIF, ELSE, ENDIF, DO, UNDO, CYCLE, ENDDO, and RETURN). The CFG PDL also allows references to External Segments, i.e., segments that are not part of the system being designed, such as operating system services or utilities.

A processor has been built to support the CFG PDL by reading in design segments and producing a working design document consisting of a table of contents, a formatted listing of segments, and a cross-reference of procedure calls. The processor is available on several computers but available documentation does not specify which computers. Further information can be found in the reference guide [PDL75].

## Table II

## Characteristics of CFG PDL

1. BASIC CONTENT

a. External Interfaces

Not explicitly specified.

b. Static Structure

(1) Composition and Organization

Text Segments, Flow Segments (like procedures), Data Segments. A Group contains a number of Flow Segments, but a segment cannot contain another segment. The organization is therefore limited to two levels - Groups and Segments.

(2) Calling Dependencies and Sequence

Implicit in free-form statements that reference other segments. Referencing dependencies are listed in a report from the processor.

(3) Data Dependencies

Parameter list of Flow Segments.

(4) Ownership of Resources

Internal variables can be implicitly defined anywhere within a Flow Segment by preceding the data name with a special character.

c. Dynamic Structure

Not expressed.

Table II (Continued)

d. Data

(1) Organization or Structure

Only the name of a data item is defined. No information on the type of data is specified. Each item is a simple item, not a structure consisting of constituent parts. Data therefore cannot be represented in different levels of abstraction.

(2) Scope or Access

Data items defined implicitly in a Flow Segment by flagging with a data character are internal variables known only within the scope of the Flow Segment. Data items defined in Data Segments are all global and can be accessed by any Flow Segment.

(3) Operations

Not defined except in English commentary.

(4) Size

Not defined except in commentary.

(5) Flow

Not represented.

(6) Derivation

Not represented.

e. Control Flow

Specified by Flow Segments with the constructs of IF, ELSEIF, ELSE, ENDIF, DO, UNDO, CYCLE, ENDDO, and RETURN.

Table II (Continued)

f. Levels of Abstraction

Abstraction of modules is limited to two levels: Groups and Segments. Data cannot be refined into constituent parts except in commentaries rendering data abstraction capabilities almost non-existent.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

Only in commentary.

(2) Solution

Only in commentary.

(3) Trade-offs between Alternatives

No provision.

(4) Rationale of Decisions

No provision.

b. Error Handling

None.

c. Performance Estimates

None.

d. Verification Information

None.

Table II (Continued)

3. CONSTRAINTS

a. Hierarchy

Two levels of hierarchy of modules, and none for data.

b. Construction Rules

Structured programming control structures, and top-down design are meant to be applied.

4. FORMAT

a. Input Format

Textual.
Combination of (1) statements with defined syntax and (2) free-form English statements and commentary.

b. Internal Format

Information not available from documentation.

c. Output Format

Textual.
Same as input format as well as additional listings produced by reports (see Automatic Tool Support).

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Small. Several types of segments. About 10 constructs, and one data definition statement.

b. Syntax

Well defined but single syntax for segments and statements within them. Free-form comments.

Table II (Concluded)

c. Similarity to Familiar Languages

Flow Segments are similar to procedures in programming languages. Though the keywords are somewhat different, the constructs in Flow Segments are basically familiar structured programming constructs.

d. Eventual Implementation Language

None.

e. Transformation into Code

Seems easy. Recursive references may require special attention if implementation language does not permit the use of recursive calls.

f. Type of Applications Suitable

Lack of dynamic behavior representation makes it less suitable for real-time systems.

g. Status

A language reference manual exists and the PDL has been applied by Caine, Farber & Gordon, Inc.

## 6. AUTOMATIC TOOL SUPPORT

The CFG PDL is supported by a processor available on several computers. Processor output includes: (1) messages on erroneous input statements; (2) a tree structure showing nesting of design segment references (recursive references are denoted once but not traced further); (3) segment index listing all Flow Segments and External items showing the locations of their definitions and the references to them; and (4) index of data items - where defined and where referenced.

## FLOWCHARTS

Flowcharts are included here because of their long history of being the most widely used form of design representation. Though their merits have been severely questioned since the emergence of structured programming, and though some practitioners in the field have not been as enthralled with them in the first place, one has to admit that they have satisfied an important need for a long time. By discussing their characteristics here, a comparison can be drawn between them and newer PDL's. It is assumed that high-level flowcharts are used to show overall program structure and detailed flowcharts to show control flow within a module.

## Table III

## Characteristics of Flowcharts

1. BASIC CONTENT

   a. External Interfaces

      Shown in high-level flowcharts. There are different symbols for different physical forms of input and output media, e.g., punched cards, tapes, printed reports, etc.

   b. Static Structure

      (1) Composition and Organization

         High-level flowcharts show modules (denoted by rectangular boxes) but not necessarily how they are organized to form the whole system.

      (2) Calling Dependencies and Sequence

         High-level flowcharts show calling dependencies and sequences.

      (3) Data Dependencies

         Not directly represented.

      (4) Ownership of Resources

         No.

   c. Dynamic Structure

      No.

   d. Data

      (1) Organization or Structure

         Flowcharts are not designed to represent data. Data are referred to by name in the control flow; they are not declared ahead of time.

Table III (Continued)

(2) Scope or Access

No.

(3) Operations

No.

(4) Size

No.

(5) Flow

No.

(6) Derivation

Buried in the detailed logic.

e. Control Flow

By means of arrows, diamond-shaped boxes indicate a decision (choice), etc. No restrictions on allowable connections.

f. Levels of Abstraction

Data abstraction is non-existent. For modules, it may be possible in high level flowcharts.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

(2) Solution

No.

Table III (Continued)

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

No.

c. Performance Estimates

No.

d. Verification Information

No.

3. CONSTRAINTS

a. Hierarchy

None.

b. Construction Rules

None.

4. FORMAT

a. Input Format

Graphical.

b. Internal Format

c. Output Format

Graphical.

Table III (Concluded)

5. APPLICATION CONSIDERATIONS

a. Size of the Language

A small number of symbols.

b. Syntax

The symbols are meant to indicate definite things. Explanation within boxes, diamonds, etc., is free-form.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

None.

e. Ease of Transformation into Code

The flow is easily transformed into code if the programming language allows uncontrolled branches (GOTO's). Data definitions have to be worked out.

f. Type of Applications Suitable

Not for real-time systems.

g. Status

Almost exclusively used as design or documentation aid until recently.

6. AUTOMATIC TOOL SUPPORT

Automatic aids have been built to transform code into flowcharts. No analysis of design is provided.

## HIERARCHY PLUS INPUT-PROCESS-OUTPUT (HIPO)

HIPO was developed originally by IBM as a documentation aid, though since then it has also been used as a representation of system design during development. The two basic components are [Kral75, Stay76]:

(1) A hierarchy chart or visual table of contents which shows how each module is further divided into modules, the order in which they are first called, and whether the call is conditional. The modules are also numbered according to the level and their position among modules on the same level.

(2) Input-process-output (IPO) charts, which further define each module in the hierarchy in terms of its inputs and outputs, the processing steps needed to perform the function of the module, and some of the control sequence governing the steps. Each of these charts is numbered according to the corresponding module in the hierarchy chart whose expansion it represents. In the lower right corner of each chart there is usually an extended description where additional free-form notes can be placed on any aspect of the design.

The main difference between HIPO diagrams and flowcharts is that the input-output data requirements for each module are much more apparent in HIPO diagrams. However, control flow is documented in a more-or-less free-form manner and usually not as completely as in a flowchart. For instance, in the hierarchy chart, submodules are shown in order of execution from left to right, and a diamond-shaped box denotes that the execution of that submodule is conditional. However, if iteration of any of these submodules takes place, it is not clear from HIPO charts alone which steps are iterated. The more detailed input-process-output charts may not necessarily show the complete control flow either. For instance, a HIPO chart may not show if a module recycles. Processing steps are represented by English words and the criteria for the breakdown are left entirely up to the discretion of the designer.

The hierarchy chart imposes a tree-structure on the organization of its modules. Service modules that perform a common utility function for more than one module on the tree are not easily identified and to identify the commonality of the function the designer has to bend the rules of numbering of the modules to index the more detailed IPO charts.

## Table IV

## Characteristics of HIPO

1. BASIC CONTENT

a. External Interfaces

Explicitly found in IPO chart of the top level module of a system.

b. Static Structure

Usually explicit for high levels only.

(1) Composition and Organization

Modules are denoted by rectangular boxes in the hierarchy chart. The modules form a tree structure. Functions performed by modules are stated in a few words.

(2) Calling Dependencies and Sequence

For higher level modules defined in the hierarchy chart, calling dependency is represented by arrows; calling sequence is implied from left to right and a conditional call is denoted by a diamond-shaped box.

(3) Data Dependencies

Partially represented in the inputs and outputs of a module as shown in the IPO charts. However, neither the origin of the input nor the destination of the output is shown so that each module is not related to another module via shared data.

(4) Ownership of Resources

No internal data is represented.

c. Dynamic Structure

No.

Table IV (Continued)

d. Data

(1) Organization or Structure

Only data used as input and output to a module is specified. Definition is by name only.

(2) Scope or Access

Access right is implied when used as input. No scope applies.

(3) Operations

None.

(4) Size

None.

(5) Flow

The flow of data can be traced through input/output arrows and the calling sequence, but this can be tedious and involves many pages of charts. Also, as said before, the charts do not show where the inputs came from. To trace the flow one has to identify the data by name.

(6) Derivation

Partially represented by arrows in IPO charts connecting the inputs and outputs to the processing steps. For instance, an output may be generated from some input and is used by a subsequent processing step to help generate another output.

e. Control Flow

At higher levels (in hierarchy chart), conditional calls are denoted and sequence is implied from left to right, but complete control flow is not represented. At the more detailed level of IPO charts, control flow is represented

Table IV (Continued)

in free-form English statements that usually resemble programming code. However, completeness of the control flow representation is not guaranteed.

f. Levels of Abstraction

Functional abstraction is represented by hierarchical levels of modules. There is no data abstraction capability.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

In free-form processing steps.

(2) Solution

Same as above.

(3) Trade-offs between Alternatives

In extended description.

(4) Rationale of Decisions

Same as above.

b. Error Handling

No.

c. Performance Estimates

None.

d. Verification Information

None.

Table IV (Continued)

3. CONSTRAINTS

a. Hierarchy

Modules are required to be organized in a tree-structure. The hierarchy chart cannot be drawn as a network. Common routines from different branches of the tree have to be repeated and may be denoted to be the same in the extended description in a corner of the chart.

4. FORMAT

a. Input Format

Graphical format with free-form statements in detailed charts.

b. Internal Format

Does not apply.

c. Output Format

Same as input format.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

A few conventions and graphical symbols.

b. Syntax

Syntax defined for modules, conditional calls, and input/output.

c. Similarity to Familiar Languages

Free-form statements can be in English or pseudo-code.

d. Orientation toward Implementation Language

None implied.

Table IV (Concluded)

e. Ease of Transformation into Code

Seems fairly easy to transform into code, except control flow is not complete.

f. Type of Application Suitable

Not for real-time systems.

g. Status

Documentation available and has been used.

6. AUTOMATIC TOOL SUPPORT

None.

## HIGHER ORDER SOFTWARE SPECIFICATION LANGUAGE AXES

Higher Order Software (HOS) is a design methodology proposed by Hamilton and Zeldin that includes its own notation for defining software [Hami76]. Proponents of HOS claim that it can be used to define software at the requirement, specification, and design levels.

HOS to date consists of a formal set of axioms and laws to govern the definition of software. "Entities" considered and their relations are listed below:

1. An input set of Elements;
2. An output set of Elements;
3. Functions that are mappings from an input set to an output set; they are organized in a tree-structure;
4. A function is sometimes referred to as a module when it controls a set of functions at a lower level on the tree structure.

A set of six "axioms" and the theorems derivable from it constitute the formal laws governing the definition of software in HOS. The axioms address the issues of control of modules over: (1) functions on its immediate lower level; (2) the ordering of the tree structure at its immediate lower level; and (3) access rights to input and output sets of variables.

The AXES specification language is the language proposed to support HOS. It consists of graphical 'control maps' and an 'invocation tree', as well as textual statements. For systems described in AXES, objects represented are variables, values, functions, and trees; the relationship described is control. "Abstract control structures" can be used to define operations, functions, and structures, all of which contain statements. Structures can be used to define relations such as partitioning of data (set partition and class partition). Statements in AXES are statements of fact, not commands to be performed. For instance, in a programming language, "z = x + y" means to add the values of x and y and store the result in z. In AXES, x, y, and z are variables, each of which represents one of a possible range of values; "=" means identical as in mathematics. "z = x + y" in AXES therefore means z always represents the same value as "x + y".

A design analyzer and a structuring executive analyzer to perform automatic analysis of design interfaces are also proposed but have not been implemented to date.

## Table V

## Characteristics of HOS AXES

1. BASIC CONTENT

   a. External Interfaces

   Shown at the top level on the control map of the system, in very simple form. If A is the system, y the output of A, and x the input of A, y = A(x) appears at the top of the control map of A.

   b. Static Structure

   (1) Composition and Organization

   Functions or modules which control functions beneath it. Organization is in the form of a tree-structure (a control map).

   (2) Calling Dependencies and Sequence

   The invocation tree indicates calling dependencies but not the sequence.

   (3) Data Dependencies

   Shown on the control map, but always in the form of y = A(x).

   (4) Ownership of Resources

   c. Dynamic Structure

   No.

   d. Data

   In AXES, a variable represents one among a possible set of values, and this value remains the same because AXES statements are not commands that can cause the values of variables to change.

Table V (Continued)

(1) Organization or Structure

Fixed types of data (called intrinsic types) as well as user-defined abstract data types expressed in terms of existing or previously defined types.

(2) Scope or Access

The normal scope of a variable is within the structure in which it is declared. Its access rights are governed by the position of the module on the tree and selected rights granted it by its parent.

(3) Operations

Can be defined in terms of primitive or already defined operations.

(4) Size

(5) Flow

From the control map.

(6) Derivation

e. Control Flow

No.

f. Levels of Abstraction

For both data and modules.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

To decompose functions into further functions, the input data is decomposed by primitive composition, set

Table V (Continued)

partition, and class partition. Thus the algorithm is partially indicated.

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

A FAIL structure can be defined to specify output values and actions taken in case of the failure of a function.

c. Performance Estimates

No.

d. Verification Information

Assertions can be made about a data type, but its meaning is not clear from the documentation of the AXES language [Hami76].

3. CONSTRAINTS

a. Hierarchy

Both module invocation and decomposition into functions are restrained to a tree structure.

b. Construction Rules

6 HOS axioms governing invocation of modules, access rights, etc.

Table V (Concluded)

4. FORMAT

a. Input Format

Graphical and textual.

b. Internal Format

Not known at this point.

c. Output Format

Graphical and textual.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Large. Difficult to learn from present documentation.

b. Syntax

The syntax of AXES is not trivial.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

None specified.

e. Ease of Transformation into Code

f. Type of Applications Suitable

Not for real-time systems.

g. Status

AXES has only recently been defined.

6. AUTOMATIC TOOL SUPPORT

Proposed but not implemented.

### IBM PDL

The IBM PDL is documented in [VanL76]; a slightly different version of it is also given as an example of a PDL called Pidgin in Volume VIII of the RADC Structured Programming Series [Kral75]. The primary purpose of this PDL is "to facilitate the translation of functional specifications into computer instructions using top-down structured programming" [Kral75]. In other words, it is designed for use in detailed design.

The language is an English-like representation of program logic utilizing structured programming control structures and indentation to show nested scopes of logic. In this PDL, statements are written similarly to those of programming languages such as PL/I. Control constructs allowed by the language are IF THEN ELSE, DO WHILE, DO UNTIL, CASE, and EXIT. Free-form English statements or textual descriptions are also permitted in addition to the basic constructs. Segmentation is accomplished by INCLUDE and CALL statements and the use of paragraph names.

The IBM PDL is intended to be used in the design of programs that are to be implemented in PL/I to which it bears much similarity. It is in fact a non-compilable dialect of PL/I.

## Table VI

## Characteristics of IBM PDL

1. BASIC CONTENT

a. External Interfaces

Not explicitly specified.

b. Static Structure

(1) Composition and Organization

Modules, identified by paragraph names, are nested to any number of levels. The INCLUDE statement, similar to that in a PL/I preprocessor, identifies a module that is eventually to be expanded as in-line code.

(2) Calling Dependencies and Sequence

Calling dependencies are implicit in free-form call statements and calling sequence is implicit in the order of appearance of such call statements in the PDL description of a system.

(3) Data Dependencies

Shown in the form of parameters between the calling and called modules in call statements.

(4) Ownership of Resources

Files and data can be defined by free-form statements but it is unclear if data so defined are global or internal.

c. Dynamic Structure

Not represented.

Table VI (Continued)

d. Data

(1) Organization or Structure

The emphasis of the IBM PDL is on control flow, so much so that data is sorely neglected. It is not clear if the rules of PL/I on data declaration apply or if data is to be declared in free-form English. In the former case, organization of data is limited by what is available in PL/I and the scope of data is governed by what PL/I implies. In the latter case, any data can be defined representing different levels of abstraction, although access rights will not be enforced. Conceivably, the PDL can be used with either approach - it is just not specified.

(2) Scope or Access
See (1) above.

(3) Operations

Can conceivably be defined by free-form English but the language has no special provisions for enforcing only the legal operations.

(4) Size

Free-form.

(5) Flow

Not represented except for parameters passed among modules.

(6) Derivation

No provision for this. Maybe in free-form statements.

e. Control Flow

Sequencing, IF THEN ELSE, DO WHILE, DO UNTIL, CASE, and EXIT.

Table VI (Continued)

f. Levels of Abstraction

Abstraction as applied to modules can be represented in varying degrees of detail. This PDL has no facility for handling data abstraction or the ownership of resources.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No provision except in note form.

(2) Solution

(3) Trade-offs between Alternatives

(4) Rationale of Decisions

b. Error Handling

No.

c. Performance Estimates

None.

d. Verification Information

None.

3. CONSTRAINTS

a. Hierarchy

b. Construction Rules

Control flow is restricted to IF THEN ELSE, DO WHILE, DO UNTIL, and CASE. Meant to be used with top-down structured programming methods.

Table VI (Continued)

4. FORMAT

(a) Input format

Text and PL/I-like statements.

(b) Internal Format

Does not apply.

(c) Output Format

Same as input format.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Small. 5 keywords.

b. Syntax

Rules of syntax are simple and apply only to the 5 constructs. Other statements are free-form.

c. Similarity to Familiar Languages

Similar to English and PL/I.

d. Orientation toward Implementation Language

PL/I and COBOL.

e. Ease of Transformation into Code

Easily transformable into code by hand.

f. Type of Applications Suitable

Seems unsuitable for real-time problems because of lack of dynamic description.

Table VI (Concluded)

g. Status

Usable immediately, although documentations do not give instructions as to how to handle data. The two versions also seem to differ slightly.

6. AUTOMATIC TOOL SUPPORT

None.

## MODULE INTERCONNECTION LANGUAGE (MIL75)

A class of languages called Module Interconnection Languages (MIL's) was proposed by DeRemer and Kron [DeRe76] for use in the global design of large systems to address issues of overall program structure. They advocate that information concerning the overall design usually buried within the modules, linkage instructions, and informal documentation be brought to light in a more concise, precise, and checkable form by means of expression in a MIL. They also advocate that a different language be used in detailed design languages for programming-in-the-small. MIL75 is their specific proposal of a MIL to satisfy these objectives. As of the writing of their paper, MIL75 existed only in concept and had received little use. Its design was also expected to change. Knowledge of MIL75 in this paper reflects only what is documented in [DeRe76].

MIL75 describes, by design, only the static interconnections of modules, not the dynamic relations exhibited during execution. It allows the description of:

1. System hierarchy - in terms of systems, subsystems, and functions organized as a tree-structure.

2. Provided and derived resources - each function specifies the resources provided by it, and the resources it demands from its children (derived resources).

3. Accessibility - links are drawn among siblings on the system tree; these links can form any directed graph. The links specify all the other functions to which each function has access. "A has access to B" means A can access all the resources provided by B. "Inherited access" refers to access rights of a function that are inherited from its parent on the tree. By default, a child inherits all access rights of its parent, unless a parent "wills" a child a specified subset or nothing. A parent has access to the "derived resources" it demands from any of its children. However, all descendents of its children are invisible to the parent, making it possible to build layers of virtual machines.

4. Module Placement - the concept of 'modules' here is similar to that of procedures in a programming language and is somewhat different from the way it has been used throughout this paper. Modules are to be programmed using programming-in-the-small techniques (detailed design);

their size is determined by intellectual manageability and readiness for detailed design. On each leaf on the system tree one or more modules must be attached. A non-leaf node may or may not have a module. There are rules governing the definition of resources in a module. At the least, each module must state all resources defined in it and all resources used by but not defined within it.

## Table VII

## Characteristics of MIL75

1. BASIC CONTENT

a. External Interfaces

None.

b. Static Structure

(1) Composition and Organization

Has the concept of systems, subsystems, functions organized in a tree structure. One or more modules can be attached to each node of the tree.

(2) Calling Dependencies and Sequence

Calling dependency, but not sequence, is represented by the system hierarchy tree.

(3) Data Dependencies

The resources provided by a function are explicitly stated by the relationship "provides". Which other functions can access these resources are indicated separately by the accessibility relation. Parents can use "derived" resources from children nodes on the tree. Data dependencies then are shown as a combination of "provides", "derives", and "accesses" relations. There is no provision for global variables.

(4) Ownership of Resources

"Provides" is a mechanism to insure ownership of resources. Internal variables can be defined.

c. Dynamic Structure

MIL75 has been designed to express only the static structure of a system, not the dynamic structure.

Table VII (Continued)

d. Data

(1) Organization or Structure

Data can be defined by name only. They can also be formed into "groups" to identify common data accessible by certain functions.

(2) Scope or Access

The "access" relationship explicitly allows one sibling on the system tree to access resources provided by another sibling (at the same level). A parent may demand "derived" resources from its children but not subsequent descendents. A child by default inherits the access rights of its parent unless they are explicitly limited by the parent, in which case only the specified subset is inherited. No resource is automatically global - accessed by all.

(3) Operations

Not defined.

(4) Size

Not defined.

(5) Flow

By means of usage links (the "uses" relation) all functions that have access to a data item can be identified, but not the sequence in which the data item is processed by the functions.

(6) Derivation

No.

e. Control Flow

No.

Table VII (Continued)

f. Levels of Abstraction

Abstraction of functions is achieved by subsystems and the hiding of resources with limited accessibility rights. Data abstractions, however, are not provided for.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

No.

c. Performance Estimates

No.

d. Verification Information

No.

Table VII (Continued)

3. CONSTRAINTS

a. Hierarchy

Functional decomposition of a system is limited to a tree structure. The ultimate modules of the system, therefore, also form a tree structure.

b. Construction Rules

Rules governing access to resources.

4. FORMAT

a. Input Format

Textual.

b. Internal Format

Unknown.

c. Output Format

Textual and graphical.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Not clear at this point. Only a sample is given in the documentation. Judging from the fact that only a few relations are cited, the size of the language should not be too large.

b. Syntax

A rigid syntax is defined for the graphical representation as well as textual statements.

c. Similarity to Familiar Languages

Table VII (Concluded)

d. Orientation toward Implementation Language

No specific implementation language implied.

e. Ease of Transformation into Code

f. Type of Applications Suitable

Lack of dynamic description, hence not suitable for real-time systems.

g. Status

Not defined enough for use in the writing of the definitive paper.

6. AUTOMATIC TOOL SUPPORT

A compiler has been suggested to support MIL75 by providing cross-references and graphs of system structure, accessibility links, and usage links. It can also check for consistency and support modification of system structure. No known detail specification or implementation of the compiler exists.

## P-NOTATION AND V-NOTATION

The Honeywell WELLMADE design methodology [Boyd76] has adopted with slight modification two existing notations for use as the means of design representation in WELLMADE. The P-notation, proposed by Dijkstra [Dijk75, Dijk76] as the guarded command set, represents program structure; the V-notation, based on Hoare's work [Hoar72], represents the data.

The P-notation has seven program constructs: concatenation, selection (If), iteration (Do), assignment, null (skip), abort, and functions (procedures). No order of execution is implied in the actions listed in If's and Do's - if the Boolean expression is true, all actions listed in connection with it are executed, but in an arbitrary order. This mechanism is useful in representing what Dijkstra termed 'non-determinacy'. One good example of where this can be useful is in the representation of design for a transaction processor receiving commands from a user at a terminal who may specify a number of commands, but in any order. Developers of the WELLMADE methodology are considering the addition of the CLASS concept as in SIMULA and a similar concept called MONITOR to manipulate concurrent processes.

The WELLMADE method of design is a constructive approach. An input set of state variables and an output set of state variables are first constructed for the top-level function, and then the output state is used to suggest the 'program' by an informal method. The 'program' is the series of steps to achieve the function. The same strategy is applied to any lower level functions that may appear in the 'program'. This informal approach is recommended for informal use and is a simplification of the more formal method proposed by Dijkstra in [Dijk75] and [Dijk76]. Dijkstra uses the notation wp(S,R), where S denotes a statement list and R some condition on the state of the system, to denote "the weakest precondition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post-condition R." He shows how the weakest precondition can be derived and the program constructed. If the input state can be proved to be a subset of the weakest precondition, then the program constructed will terminate and accomplish the results represented by the post-condition R. This formal approach, needless to say, can be long and tedious, but does provide a proof of correctness at the same time the program is constructed.

Table VIII

Characteristics of P-NOTATION and V-NOTATION

1. BASIC CONTENT

a. External Interfaces

Not apparent. The sets of input and output state variables specify conditions that must be true on entry to and exit from the system.

b. Static Structure

(1) Composition and Organization

Functions that transform a set of specified input state variables into a set of specified output variables.

(2) Calling Dependencies and Sequence

From function call constructs.

(3) Data Dependencies

Only input and output states of each function.

(4) Ownership of Resources

Internal variables.

c. Dynamic Structure

Can specify a set of actions whose order is not specified, i.e., they can happen in random order. Considerations are taken to include concept of concurrent processes.

d. Data

(1) Organization or Structure

Several fixed types. The Array type (suggested by Dijkstra) acts as a structure.

Table VIII (Continued)

(2) Scope or Access

Scope allows the specification of access rights to variables by specifying whether they are:

1. Global - inherited from ancestor, must not be initialized.
2. Latent - inherited from ancestor, must be initialized.
3. Private - introduced and must be initialized.
4. Constant - value must not change.
5. Variable - value may be modified.

(3) Operations

Fixed set of operations.

(4) Size

(5) Flow

Not apparent.

(6) Derivation

May be partially contained in the 'program'.

e. Control Flow

With the constructs of the P-notation. However, since no order is implied in the If's and Do's, one has to construct the Boolean expression in a special way, or nest the If's and Do's, to force an order if so desired.

f. Levels of Abstraction

The breaking down of functions into lower level functions as a result of the construction of the 'program'. For data, the use of Array, which is a structure that can define collections of more primitive types of data. Or, in free-form descriptions in definitions of data at higher levels.

Table VIII (Continued)

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

May appear as a result in 'program'.

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

Hidden in If's and Do's.

c. Performance Estimates

No.

d. Verification Information

The constructive approach of the methodology as explained briefly in the text allows the verification of programs.

3. CONSTRAINTS

a. Hierarchy

With the given constructs in P-notation, the resulting 'program' is bound to obey structured programming rules of construction and fit into a hierarchy structure.

Table VIII (Continued)

b. Construction Rules

The philosophy of WELLMADE naturally applies. However, the P-notation or V-notation can be used separately and independently as Dijkstra and Hoare first envisioned them.

4. FORMAT

a. Input Format

Textual.

b. Internal Format

c. Output Format

Textual.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

6-7 program constructs and about 6 data constructs.

b. Syntax

Well defined but also admits free-form expressions.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

None.

e. Ease of Transformation into Code

Seems easy as long as the data types offered by V-notation are easily implementable in the programming language of choice.

Table VIII (Concluded)

f. Type of Applications Suitable

Admits specification of random events for partial real-time specification.

g. Status

Documentation still not complete on the methodology.

6. AUTOMATIC TOOL SUPPORT

DOCA - to display documentation.

P-notate - on-line tool to format P-notation, assign statement numbers, and retrieve a portion of program structure for display.

V-notate - on-line tool for formatting V-notation, retrieving associated documentation, and producing cross-references of the data.

## PROBLEM STATEMENT LANGUAGE (PSL)

Problem Statement Language (PSL) was developed by the ISDOS (Information Systems Design and Optimization System) project at the University of Michigan [Teic74]. The Air Force has acquired a special version of PSL/PSA with some added capabilities. This version is known as URL/URA (User Requirements Language and Analyzer).

PSL was originally designed to formalize the requirements for a large computer-based information system, but its capabilities make it useful in high-level design as well. It has the ability to describe the static structure of a target system in terms of objects within the system, the relationships among them, including data connections. For example, PSL delineates the boundaries of the system by identifying physical units of data or information external to the proposed system, such as documents used as input to or reports generated by the system - these are called Inputs and Outputs. It also identifies the people, departments, etc., who produce these Inputs and use the Outputs - they are called Real World Entities. Next, the units of data are identified, called Sets of Entities or Groups of Elements. They, together with Inputs and Outputs, represent the information flow through the system. Finally, there are Processes and their constituent processes which operate on the data.

The objects listed above are tied together by relationships such as Subpart Of, Contained In, Uses, Derives, and Updates.

The ability to represent the dynamic behavior of a system is limited to describing Events that Trigger certain Processes. In addition, sizing information, some project management information, and narrative descriptions can be expressed.

Concomitant with the development of PSL was the development of a software package PSA (Problem Statement Analyzer) that builds a data base from a set of PSL statements, checks it for consistency and completeness, and retrieves as well as manipulates selected information from the data base, generating reports to the user for analysis [Berg74]. Reports generated include varied representations of the information in the data base such as selected listings, retrieval by keyword, matrices, and flow diagrams.

## Table IX

## Characteristics of PSL

1. BASIC CONTENT

   a. External Interfaces

   PSL emphasizes this area. Real World Entities in PSL refer to the people, departments, organization, etc., using the target system. Inputs and Outputs represent data external to the system as distinct from internal variables.

   b. Static Structure

   (1) Composition and Organization

   Processes form the basic units. Each Process can be decomposed into Subparts that are also Processes.

   (2) Calling Dependencies and Sequence

   Calling dependencies are embedded in the Subpart of relationship for it is assumed that a Process will call on its constituent parts to perform its function. The calling sequence is not indicated.

   (3) Data Dependencies

   PSL has no parameter-passing capability. The data dependencies among Processes are hidden in the Uses, Derives, and Updates relationships. A Process specifies the data items it uses and modifies, but where the data come from or go to are not specified. One report in PSA tabulates these relations in matrix form.

   (4) Ownership of Resources

   All data are global; no internal data are allowed.

   c. Dynamic Structure

   Events Trigger Processes. Conditions can also be specified. But none of the dynamic activities are related to the data.

Table IX (Continued)

d. Data

(1) Organization or Structure

Units of data are Sets of Entities (e.g., files of records) or Groups of Elements (structure of primitive data types). Entities can consist of Groups. Data is declared by name, and type is specified by ATTRIBUTE. Sets, Entities, and Groups are meant to denote logical collections of data; each can be further decomposed into constituent parts.

(2) Scope or Access

All global, accessible by all.

(3) Operations

Not defined except in free-form description, or contrived by using the ATTRIBUTE keyword.

(4) Size

Cardinality can be specified.

(5) Flow

The processes that interact with the data are indicated by the Uses, Updates and Modifies relations. However, the order in which the data flow through the Processes is not known, i.e., a thread cannot be traced.

(6) Derivation

Partially from the Derives, Uses or Using relations. A Process can Derive a variable using other variables, but the full algorithm is not specified.

e. Control Flow

None.

Table IX (Continued)

f. Levels of Abstraction

Modules are abstracted by specifying Processes in a top-down manner from abstract to concrete. Data abstraction is accomplished by using the four levels of Sets, Entities, Groups and Elements.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

Only in textual description.

(2) Solution

Only in textual description.

(3) Trade-offs between Alternatives

Only in textual description.

(4) Rationale of Decisions

Only in textual description.

b. Error Handling

No.

c. Performance Estimates

Via ATTRIBUTE and textual descriptions.

d. Verification Information

None.

Table IX (Continued)

3. CONSTRAINTS

   a. Hierarchy

   The SUBPART OF relation governing both the Processes and data is in the form of a tree structure. The CONSISTS OF relation forces a hierarchical structure.

   b. Construction Rules

4. FORMAT

   a. Input Format

   Textual.

   b. Internal Format

   CODASYL data base.

   c. Output Format

   Textual, graphical and matrices.

5. APPLICATION CONSIDERATIONS

   a. Size of the Language

   Over 200 keywords.

   b. Syntax

   Defined for Sections and Statements.

   c. Similarity to Familiar Languages

   d. Ease of Transformation into Code

   Since control flow is missing, it is not suitable for use in detailed design.

Table IX (Concluded)

e. Type of Applications Suitable

PSL admits partial description of dynamic behavior for real-time systems.

f. Status

The support tool PSA has been installed at various places, on various computers. Documentation for some versions is available.

## 6. AUTOMATIC TOOL SUPPORT

The Analyzer PSA parses PSL statements and performs checks on the consistent use of names, the completeness of definition of data items; produces various listings; and tabulates relations between Processes and data items. It also produces pictorial output for certain items.

## PROCESS DESIGN LANGUAGE (PDL2)

The Process Design Language is a part of the Process Design Methodology developed at Texas Instruments Inc., Alabama, in support of the Ballistic Missile Defense Advanced Technology Center. The primary goal of TI-PDL is to provide a means of defining and manipulating large real-time processes where timing and order of events are critical. Koppang published a paper [Kopp76] in which Process Design Language is discussed. In his presentation of the paper at the 2nd International Conference of Software Engineering, Koppang stated the following as requirements for the language:

- Extensive error detection capability,
- Supports structured programming,
- Well-defined scope rules,
- Advanced data structures,
- Based on an established programming language,
- Allows efficient code generation (because of real-time application),
- Supports synchronization of concurrent processes, and
- Access to bare hardware.

PASCAL, deemed most suitable, was chosen to be extended to meet the above requirements. The current version of the language is called PDL2. The extensions to PASCAL incorporated the following capabilities:

- Absolute memory accessing (user restricted),
- Memory boundary alignment for execution time efficiency,
- Double precision arithmetic,
- Communication with FORTRAN,
- Tasking and synchronization,
- Variable length arrays,
- Vector operations,
- Assertion statements providing run-time verification,
- Data collection statements for performance data,
- Parameterized string substitutions (MACROS), and
- An Escape statement.

An integrated set of tools, called Process Design System (PDS2), is being developed to support PDL2 and contains a compiler for PDL2. PDS2 is scheduled for completion in 1977. Plans have been made for its use within Texas Instruments.

## Table X

## Characteristics of PDL2

1. BASIC CONTENT

   a. External Interfaces

   b. Static Structure

      (1) Composition and Organization

      Procedures form a tree structure. Macros permit the definition of in-line procedures.

      (2) Calling Dependencies and Sequence

      In procedure call statements.

      (3) Data Dependencies

      Explicitly by parameter list, and implicitly by the fact that a procedure has knowledge of all definitions and variables declared in the higher level modules in which it is nested..

      (4) Ownership of Resources

      Internal variables.

   c. Dynamic Structure

   Processes may be interrupted by higher priority tasks. Tasking and synchronization of concurrent processes.

   d. Data

      (1) Organization or Structure

      PASCAL has good data structuring capabilities including sets of data whose members are denumerable. Variable length arrays and vector operations are added. Data type defines the set of values that may be assumed by that variable. More detailed

Table X (Continued)

information on PASCAL data definition can be found in [Wirt71].

(2) Scope or Access

A data item declared in one procedure can be used by all other procedures defined within that procedure.

(3) Operations

Fixed set of operations.

(4) Size

As in PASCAL.

(5) Flow

Only in parameter lists.

(6) Derivation

No.

e. Control Flow

IF, CASE, REPEAT, WHILE, FOR, GOTO, and ESCAPE statements.

f. Levels of Abstraction

Functional abstraction is achieved by the use of nested procedures. Limited data abstraction is achieved by the use of the set and class structures of data.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

Table X (Continued)

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

No.

c. Performance Estimates

Data collection statements collect program performance data in a standardized, compiler verified format to promote the implementation of general purpose design analysis aids.

d. Verification Information

Assertion statements are verified at run time.

3. CONSTRAINTS

a. Hierarchy

Static system structure must be a tree structure.

b. Construction Rules

Structured programming rules apply.

4. FORMAT

a. Input Format

Textual.

Table X (Continued)

b. Internal Format

c. Output Format

Textual.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

PASCAL + extensions.

b. Syntax

PASCAL syntax.

c. Similarity to Familiar Languages

PASCAL.

d. Orientation toward Implementation Language

Although it has not been stated that eventual implementation is limited to PASCAL, presumably it would be easier to implement the design ultimately in PASCAL.

e. Ease of Transformation into Code

Easy to transform PDL2 into PASCAL code.

f. Type of Applications Suitable

Designed to handle real-time processes.

g. Status

Language reference material has not been received or reviewed for this task.

Table X (Concluded)

6. AUTOMATIC TOOL SUPPORT

An extensive set of tools called Process Design System (PDS2) was planned and scheduled for completion in 1977. Components of the tools include the PDL2 Compiler, Object Module Processor, Process Constructor, Simulation Experiment and Process Design Analysis System. Capabilities include the maintenance of each module as a separate entity, the maintenance of system configuration, control of access to modules, automatic look-up of data available to a module from high-level modules, detection of interface errors, and the simulation of processes to approximate their behavior in a multi-tasking environment.

## PROGRAM DESIGN AND DOCUMENTATION LANGUAGE (PDDL)

PDDL is a product of Jet Propulsion Laboratory, California [Heim 77], consisting of a set of notational conventions for recording the design of a system, and a computer program written in SIMSCRIPT to maintain the system design and generate reports. Three publications are planned: a user's guide, program documentation, and the engineering methodology - though none is available at this point.

Modules are represented as procedures. Each procedure is separate and lists at the beginning variables it receives (GIVEN), uses (USING), and returns (YIELD). Statements within procedures include file manipulation statements, procedure call statements, and typical control structures such as DO, SELECT, IF, ELSE, and ENDIF. Files and data structures are also defined separately from procedures and from one another. Each 'program data-structure' defines a data structure associated with a name and type. Its contents are listed by name and type within it. Data can be entities, data structures or substructures, definitions of which are not clear from available documentation at this point. The support tool generates module reference trees and data cross-references.

## Table XI

## Characteristics of PDDL

1. BASIC CONTENT

a. External Interfaces

Portions of input provided by the user may be found in file definitions, but other than that, external interfaces are not represented.

b. Static Structure

(1) Composition and Organization

Procedures, each of which is separate.

(2) Calling Dependencies and Sequence

In procedure call statements. The support tool, however, generates a module reference tree as well as a procedure cross-reference showing where the code of each function resides and where it is referenced.

(3) Data Dependencies

Stated explicitly at the beginning of each procedure under GIVEN, USING, and YIELD. However, it is not known where these variables come from. Only in conjunction with calling dependencies will data dependency among modules be complete.

(4) Ownership of Resources

No internal variables.

c. Dynamic Structure

No.

Table XI (Continued)

d. Data

(1) Organization or Structure

Data can be a structure, entity or substructure. Each can be defined in terms of the others.

(2) Scope or Access

All data are defined separately in a 'program data-structure' or 'program' and are global. Variables used as parameters concern only the modules involved.

(3) Operations

Not defined.

(4) Size

By the structure or listing of component parts.

(5) Flow

No. The automatic tool does provide a report showing places of reference of each data item in terms of the procedure that performs the reference, and the page and line of the referencing statements. It does not specify the order of the references.

(6) Derivation

No, except in free-form text.

e. Control Flow

In procedures. DO, SELECT, IF, ELSE, and ENDIF statements.

f. Levels of Abstraction

None for modules. Data can be organized into structures but existing documentation does not specify if they can first be declared by name only and have the details filled in afterwards.

Table XI (Continued)

2. AUXILIARY CONTENT

   a. Decision Details

      (1) Algorithms

          No.

      (2) Solution

          No.

      (3) Trade-offs Between Alternatives

          No.

      (4) Rationale of Decisions

          No.

   b. Error Handling

      No.

   c. Performance Estimates

      No.

   d. Verification Information

      No.

3. CONSTRAINTS

   a. Hierarchy

      Each module and data definition is separate.

   b. Construction Rules

      (1) Explicit Rules

      (2) Implicit Rules

Table XI (Continued)

4. FORMAT

a. Input Format

Textual.

b. Internal Format

Unknown.

c. Output Format

Textual. Lists and cross-reference tables.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Medium.

b. Syntax

Many notational conventions that have to be learned.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

None specified.

e. Ease of Transformation into Code

Since PDDL representations are divided into procedures and data definitions, transformation into code should not be difficult.

f. Type of Applications Suitable

g. Status

Table XI (Concluded)

6. AUTOMATIC TOOL SUPPORT

The support tool generates several reports:

1. module reference tree;
2. data versus places (procedure, page, and line) of reference;
3. procedure cross-references in alphabetical order showing, for each procedure, all references to it and the location of its text; and
4. miscellaneous listings showing cross-references of files and changes from the previous version.

## SOFTWARE DESIGN LANGUAGE (SDL-1)

A family of software design languages has been suggested and its first member SDL-1 introduced by Chu [Chu76]. Chu sees a SDL mainly as a design aid to describe the design of software for communication and documentation. Should a simulator become available, the design can be checked out. He also thinks that there is a methodology implied in a software design language.

SDL-1 represents design in several parts. The first three paragraphs in the DESIGN SPECIFICATION (i. e., the PROCEDURES, DATA, and SWITCHES paragraphs) deal with overall design. The PROCEDURES paragraph consists of two parts, one called PROCEDURE DECLARATION where all procedures are declared in nested levels by name and function, and the other called PROCEDURE STRUCTURE where all procedures called by each procedure are listed. Likewise, the DATA paragraph consists of DATA DECLARATION and REFERENCE STRUCTURE. Similar dichotic schemes also exist for SWITCHES, which are similar to status variables. All data and switches declared at this level are global.

The last paragraph of design in SDL-1, the DEFINITIONS paragraph, consists of procedure definitions. Each procedure defined under PROCEDURES is further expanded into detailed statements with appropriate control structures.

## Table XII

## Characteristics of SDL-1

1. BASIC CONTENT

a. External Interfaces

Not explicitly defined.

b. Static Structure

(1) Composition and Organization

Declaration of single-in-single-out (i.e., one-entry-one-exit) procedures within the PROCEDURES paragraph in nested levels. Procedures can also be implicitly declared in the PROCEDURE STRUCTURE as part of a "call list".

(2) Calling Dependencies and Sequence

Another part of the PROCEDURES paragraph is the PROCEDURE STRUCTURE where each procedure declared lists the procedures it in turn calls, in a "call list". No sequence is implied. Sequence has to be deduced from the detailed procedure definition showing control flow and call statements. Procedures in the "call list" may be already declared, or implicitly defined by their occurrences in the list.

(3) Data Dependencies

Shown in the REFERENCE STRUCTURE under DATA, sorted by data name. The inverse relationship (the data items that a particular procedure references) has to be deduced.

(4) Ownership of Resources

Internal variables can be defined within a procedure definition.

c. Dynamic Structure

No.

Table XII (Continued)

d. Data

(1) Organization or Structure

Data items are defined by name, type, and descriptive explanation. Each item may contain several further levels of data items in a tree-structure. SWITCHES act as status variables.

(2) Scope or Access

All data declared in the DESIGN SPECIFICATION are global. All data declared in a PROCEDURE DEFINITION belong internally to that procedure.

(3) Operations

Fixed set, not user-defined.

(4) Size

No.

(5) Flow

The list of modules that reference a data item is given, but the order in which referencing takes place is unknown.

(6) Derivation

No.

e. Control Flow

In PROCEDURE DEFINITION of each procedure. Statements allowed are BLOCK, SET, IF, CASE, LOOP, EXIT, CALL, RETURN, and UNWIND.

f. Levels of Abstraction

As represented in the various levels of procedures and in data structures.

Table XII (Continued)

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

Error exits are handled by UNWIND, but the conditions handled are buried in the PROCEDURE DEFINITIONS.

c. Performance Estimates

d. Verification Information

SWITCHES can be used to represent conditions to be tested later in the design.

3. CONSTRAINTS

a. Hierarchy

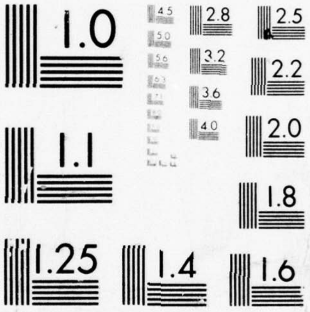
Procedures are declared in a hierarchy, but the calling relationship is not limited - a procedure can call procedures above and below it except the main procedure. Data is restricted to a tree-structure.

b. Construction Rules

AD-A051 672 MITRE CORP BEDFORD MASS F/G 9/2
PROGRAM DESIGN LANGUAGES-AN INTRODUCTION. (U)
JAN 78 L L CHENG F19628-77-C-0001
UNCLASSIFIED MTR-3446 ESD-TR-77-324 NL

2 OF 2
AD
A051672

END
DATE
FILMED
4-78
DDC


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Table XII (Continued)

4. FORMAT

a. Input Format

Textual.

b. Internal Format

Not applicable.

c. Output Format

Textual.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

3 levels of description, each level containing 2 or 3 blocks of information.

b. Syntax

Fixed syntax except in comment.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

e. Ease of Transformation into Code

Seems easy except for SWITCHES.

f. Type of Applications Suitable

Not for real-time applications because of the lack of dynamic representation.

g. Status

Language is well defined, awaits experiences in application to study its sufficiency.

Table XII (Concluded)

6. AUTOMATIC TOOL SUPPORT

Suggested but neither defined nor implemented.

## SOFTWARE SPECIFICATION LANGUAGE (SSL)

SSL, a product of Science Applications Inc., is a formalism for the definition of software specifications. It is designed to emphasize module interconnections, flexible data abstractions, and global data flow; it is not designed to depict the control flow within modules. The formal grammar of SSL is documented in Backus-Naur-Form (BNF) in [Aust 76], and historically its design has been influenced by MIL, levels of abstraction, structured charts, PASCAL, ALGOL 60 and NEUCLEUS.

A subsystem in SSL consists of a collection of modules. SSL statements are used to convey additional information about each module. The Assumes statement specifies conditions that must be true upon entry to the module. The Satisfies statement states data conditions that must be true upon module exit. The Fulfills statement specifies the requirements accomplished by the module. The Accesses statement indicates which environmental objects such as peripheral equipment are utilized by the module. The Receives and Transmits statements indicate real-time data activity. The Creates, Modifies, and Uses statements distinguish between input and output data variables.

Besides a basic set of data types, SSL allows additional user-defined data types. Each data declaration is related to the requirements. Conditions that must be true for the creation of a data item can also be specified.

SSL is the only PDL among the ones surveyed to relate design to requirements.

## Table XIII

## Characteristics of SSL

### 1. BASIC CONTENT

a. External Interfaces

Requirement declaration is used to identify the data flow between the software and its external environment. System level input (stimulus) received by a software package from an external source and system level output (response) are represented as Input and Output parts. The Accesses statement also indicates which environmental objects are utilized by a module.

b. Static Structure

(1) Composition and Organization

Modules are the basic system objects in SSL. A subsystem is a group of modules. The requirement satisfied by each module is stated in the Fulfills statement.

(2) Calling Dependencies and Sequence

Only in Execute statement which is the equivalent of what is generally known as a Call statement.

(3) Data Dependencies

The Creates and Modifies statements within a module show the input and output variables of a module but not where they came from or are going to.

(4) Ownership of Resources

Internal variables can be defined within a module.

c. Dynamic Structure

Receives and transmits statements represent real-time activity.

Table XIII (Continued)

d. Data

(1) Organization or Structure

SSL offers a fixed set of data types and also the capability for user-defined data types. Data is also related to requirements attributes.

(2) Scope or Access

The normal scope of all data items is the subsystem in which it is declared. Global variables can be declared only in the main subsystem.

(3) Operations

Fixed set, not user-defined.

(4) Size

(5) Flow

Can be traced from the Creates, Uses, and Modifies statements.

(6) Derivation

Not the algorithm, but the variables used in the derivation of a variable can be obtained from the Uses and Using statements.

e. Control Flow

Not designed for this.

f. Levels of Abstraction

Concept of a system, subsystems and modules; also data abstraction by means of user-defined variable types.

Table XIII (Continued)

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

The projection of specialized forms of requirements onto the data and module definitions establishes the rationale for the creation of such.

b. Error Handling

No.

c. Performance Estimates

No.

d. Verification Information

Conditions necessary for the execution of a module and its return, and conditions necessary for the creation of a variable are stated so that they can be tested. Assertions represent conditions attached to the creation of variables and entry and exit of modules.

3. CONSTRAINTS

a. Hierarchy

Subsystems are separate from each other but contain modules. No module is contained within another module.

Table XIII (Continued)

b. Construction Rules

Levels of abstraction are meant to be applied. SSL also encourages a designer to relate design to requirements.

4. FORMAT

a. Input Format

Textual.

b. Internal Format

Unknown.

c. Output Format

Textual. No more information at this point on detailed capabilities of support tools planned.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

70 reserved words.

b. Syntax

Formal grammar.

c. Similarity to Familiar Languages

d. Orientation toward Implementation Languages

None.

e. Ease of Transformation into Code

No control flow specified. Detailed design necessary before transformation into code.

Table XIII (Concluded)

f. Type of Applications Suitable

Attempts to handle real-time applications.

g. Status

Language is well defined, but not the tools.

## 6. AUTOMATIC TOOL SUPPORT

Several tools, collectively known as Software Specification and Evaluation System (SSES), are planned, including a static code analyzer, a dynamic code analyzer, and a test case analyzer.

## STRUCTURED CHARTS

Structured charts are graphical notations used to support a design technique called structured design. The underlying consideration of Constantine's structured design technique [Stev74] is to isolate more simple independent modules with minimal connections among them. The result is that modules so defined are easier to comprehend, easier to implement, less error-prone, and also less repetitive because they are more likely to be reusable without recoding, thus avoiding duplicate code.

Design takes place at three hierarchical levels: system, program, and module. In [Stev74] the authors discuss coupling - the strength of the association among modules; cohesiveness; and binding. They also describe a charting method to record the decisions made, the modules identified, and their connections.

This structured design technique also places much emphasis on identifying abstractions of external conceptual streams of data and the transformations necessary to change them into the output stream. "External" data lies outside the system; a "conceptual" stream of data is a stream of related data that is independent of any physical input-output device.

Structured charts consist of a system structure tree with rectangular boxes denoting modules and lines connecting them denoting invocation (or calling relationship). All calling relationships are further defined by specifying the input variables (variables passed to the called module) and output variables (variables returned). Both modules and variables are identified by name or a short phrase of description such as "a list of unsafe factors and names".

## Table XIV

## Characteristics of Structured Charts

1. BASIC CONTENT

   a. External Interfaces

   Although external data are emphasized in the structured design technique, they are not represented in the charts. Structured charts show data connections between modules, but not between the whole system and the outside world.

   b. Static Structure

   (1) Composition and Organization

   Modules, as denoted by rectangular boxes and a free-form description, form a tree-structure.

   (2) Calling Dependencies and Sequence

   Calling dependencies are apparent in the system tree. The actual sequence is not depicted although it is roughly from top to bottom.

   (3) Data Dependencies

   Shown for every called module. The name or a short description of data passed from the calling module to the called module and vice versa is recorded.

   (4) Ownership of Resources

   No internal resources are depicted.

   c. Dynamic Structure

   No.

   d. Data

   (1) Organization or Structure

   Free-form description of data type or structure or enumeration of values.

Table XIV (Continued)

(2) Scope or Access

Only data passed among modules are specified. All data named concern only the calling and called modules.

(3) Operations

None, except maybe in free-form description.

(4) Size

Only in description, or implied by enumeration.

(5) Flow

Can be traced indirectly from system tree, but the tree itself is based on hierarchical levels of modules, not on following one thread of data.

(6) Derivation

No.

e. Control Flow

No. Only calling relationship.

f. Levels of Abstraction

Hierarchical levels of modules. For data, abstraction can be represented only in free-form description. It is interesting to note that although abstraction issues are emphasized in the design method, the charts are kept simple enough so that no special provisions are made to represent them in the notation.

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

Table XIV (Continued)

(2) Solution

No.

(3) Trade-offs between Alternatives

No.

(4) Rationale of Decisions

No.

b. Error Handling

No.

c. Performance Estimates

No.

d. Verification Information

No.

3. CONSTRAINTS

a. Hierarchy

Calling structure of module is in the form of a tree.

b. Construction Rules

The philosophy of the structured design method is implied. In constructing the modules, the rules of coupling, cohesiveness and binding apply. In constructing data, it is assumed that the design will start from a conceptual stream of data and proceed toward actual physical representation in the machine.

Table XIV (Continued)

4. FORMAT

a. Input Format

Graphical.

b. Internal Format

Not applicable.

c. Output Format

Graphical.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Small. Boxes, lines, input and output variables.

b. Syntax

There is a definite syntax.

c. Similarity to Familiar Languages

Simple charts.

d. Orientation toward Implementation Languages

None in particular.

e. Ease of Transformation into Code

Since no control flow is specified, much design needs to be done in transforming structured charts to code.

f. Type of Applications Suitable

g. Status

Documented and has been applied.

Table XIV (Concluded)

6. AUTOMATIC TOOL SUPPORT

None.

## UMTA SPECIFICATION LANGUAGE (USL)

USL was developed by the Urban Mass Transportation Administration for the detailed technical specification of computer programs. It is intended to replace flowcharts and most of the prose documentation of program design.

Modules are specified in USL as procedures. Lower level modules can first be identified by name and expanded later. The logic within procedures is limited to IF THEN, IF THEN ELSE, WHILE, and FOR EACH statement. Indentation is used to denote the scope of a clause.

Data are declared as RESOURCES and are considered to be sets. Primitive data types are considered a set with one member. Attributes of resources can be used to define the maximum number of members, methods of access, scope of the definition, and members.

USL is similar to ALGOL, but can be translated into code in any algorithmic language. One support tool automatically translates a design written in USL into FORTRAN [Zieg73]. It is claimed that "the technical specifications written in USL can also provide the design for a simulator of the system to be developed." [Zieg74]

For a language designed to specify detailed technical information of design, USL, though inadequate in specifying inter-module connections, is surprisingly capable of managing abstractions.

## Table XV

## Characteristics of USL

1. BASIC CONTENT

   a. External Interfaces

   No.

   b. Static Structure

   (1) Composition and Organization

   Basic units are Procedures.

   (2) Calling Dependencies and Sequence

   Embedded in procedure call statements.

   (3) Data Dependencies

   Only in parameter list.

   (4) Ownership of Resources

   Internal resources can be defined by defining them LOCAL. For recursive procedures these local variables can be stacked.

   c. Dynamic Structure

   No.

   d. Data

   (1) Organization or Structure

   All data are considered sets of objects, primitive data types being sets with one member. Data are declared by name as RESOURCES.

Table XV (Continued)

(2) Scope or Access

Scope or access is defined in 3 ways: by declaring resource LOCAL; by explicitly listing all procedures that can reference the resource; in the absence of a specified scope, a resource is assumed global - any procedure can reference it.

(3) Operations

Random and consecutive methods of access are provided. Access methods are specified in a general way so that the intent of the access method is specified, not one way of implementing it.

(4) Size

By maximum number of members.

(5) Flow

Hidden in statements in procedures.

(6) Derivation

No.

e. Control Flow

By IF THEN, IF THEN ELSE, WHILE, and FOR EACH statements.

f. Levels of Abstraction

Procedures are defined in the abstract and expanded afterward. Data are considered sets and can also be conceived of in the abstract with details filled in afterward.

Table XV (Continued)

2. AUXILIARY CONTENT

a. Decision Details

(1) Algorithms

No.

(2) Solution

No.

(3) Trade-offs between Alternatives

(4) Rationale of Decisions

Detailed design decisions can be recorded and the intent stated in prose under DICTIONARY.

b. Error Handling

No.

c. Performance Estimates

No.

d. Verification Information

No.

3. CONSTRAINTS

a. Hierarchy

Because of control constructs available, resulting structure of modules form a hierarchy.

Table XV (Continued)

b. Construction Rules

Indentation is to be used to denote the scope of a clause in statements within procedures. Control flow is restricted to structured programming constructs.

4. FORMAT

a. Input Format

Textual.

b. Internal Format

Not applicable.

c. Output Format

Textual.

5. APPLICATION CONSIDERATIONS

a. Size of the Language

Fair.

b. Syntax

Combination of free-form statements and well defined statements.

c. Similarity to Familiar Languages

Similar to ALGOL.

d. Orientation toward Implementation Languages

Not limited to any particular one.

Table XV (Concluded)

e. Ease of Transformation into Code

Has been automatically translated into FORTRAN. Ziegler claims USL is readily translated into FORTRAN, PL/I, COBOL, JOVIAL or ALGOL.

f. Type of Applications Suitable

g. Status

Well defined.

6. AUTOMATIC TOOL SUPPORT

USL has been translated into FORTRAN via an automatic tool.

# SECTION IV

# CONCLUSIONS

Up to this point, the background, purpose, and a definition of PDL's have been discussed; characteristics of PDL's in general have been defined; and specific properties of 14 PDL's have been examined. It is appropriate to see if the totality of this information offers any indications toward trends in the development and criteria in the selection of PDL's.

Related work that might warrant investigation are Earley's work on high level languages [Earl74], Schwartz' set-theoretic language SETL [Schw74], Parnas' specification language [Parn72b], Alphard programs [Wulf76], the Semantic Description Language for the Information Automat [Wils77], and System Analysis Design Technique (SADT) [Soft76]. These works have not been included in this survey because of the lack of timely documentation, the proprietary nature of the work, or time constraints.

## SELECTION

Some of the more directly observable characteristics of the PDL's surveyed are summarized in Table XVI. This tabulation of survey results is intended only to give a quick comparison of the PDL's reviewed. The information is very cryptically coded and nuances of the language may be lost in the transcription. Judgment of the applicability of a PDL should not be solely based on information conveyed by such a tabulation, although it may point out a few more promising candidates. A PDL is a communication tool and nothing can supplant first-hand knowledge obtained by learning from the original documentation and trial use.

In selecting a PDL for use, all of its characteristics have to be considered. Domination of one particular aspect with no regard for others may be deceptive. A poor format, or ineffective representation, for instance, will detract from the merits of a PDL despite its adequacy in information content. Some characteristics will be more pertinent for certain applications and should be given more weight than others in the selection process. This brings up the question of 'desirability' of the characteristics. Some characteristics merely offer a choice of options such as textual format or graphical format, and 'desirability' does not apply. For

Table XVI

Summary of Some Characteristics of PDL's

| Characteristics / PDL's | BASIC CONTENT | | | | | | | | | | | | | | | AU |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Static Structure | | | | | | Data | | | | | | | | |
| | External Interfaces | Composition and Organization | Calling Dependencies | Calling Sequence | Data Dependencies | Ownership of Resources | Dynamic Structure | Organization | Access | Operations | Size | Flow | Derivation | Control Flow | Levels of Abstraction | Decision |
| CFG PDL | N | Groups, Segments, (2 levels) | HT | H | P | Y | N | Simple Item | All Local or Global | N | N | N | N | Y | P– | N |
| Flowcharts | P | Modules | Y | Y | N | N | N | N | N | N | N | N | N | Y | P– | N |
| HIPO | Y | Modules | Y | Y | P | N | N | P | N | N | N | H | P | Y | P | P– |
| HOS AXES | P | Modules, Functions, (tree-struc.) | Y | N | P | | N | User-Defined | Y | Y | | P | | N | Y | P– |
| IBM PDL | N | Modules | H | H | P | | N | | H | N | N | P | N | Y | P | N |
| MIL75 | N | Systems, Subsystems, Functions | Y | N | Y | Y | N | P | Y | N | N | P | N | N | P | N |
| P-Notation & V-Notation | PH | Functions | H | H | P | Y | P | Y | Y | Y | | N | N | Y | Y | N |
| PDL 2 | | Procedures, Macros | H | H | Y | Y | P | Y | Implied by Nested Procedures | Fixed | | P | N | Y | P | N |
| PDDL | P | Procedures | HT | HT | Y | N | N | Struc. Entity Substruc. | Global | N | P | NT | N | Y | P | N |
| PSL | Y | Processes | Y | N | P | N | P | Sets, Entities Groups, Elements | All Global | N | Y | P | P | N | Y | N |
| SDL-1 | N | Procedures | Y | N | Y | Y | N | Y | Global or Internal | Fixed | N | P | N | Y | Y | N |
| SSL | Y | Modules & Subsystems | H | H | P | Y | P | Fixed | Global or Internal | Fixed | | P | P | N | P | P |
| Structured Charts | N | Modules | Y | N | Y | N | N | Free-Form | N | N | N | H | N | P | P | N |
| USL | N | Procedures | H | H | P | Y | N | Sets | Y | Y | Y | H | N | Y | Y | P |

2

Table XVI

Summary of Some Characteristics of PDL's

| C CONTENT | | | | | | | | AUXILIARY CONTENT | | | | FORMAT | | TOOLS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Data | | | | | | | | | | | | | | |
| Organization | Access | Operations | Size | Flow | Derivation | Control Flow | Levels of Abstraction | Decision Details | Error Handling | Performance Estimates | Verification Information | Input Format | Output Format | Automated Tool Support |
| Simple Item | All Local or Global | N | N | N | N | Y | P– | N | N | N | N | Text | Text | Y |
| N | N | N | N | N | N | Y | P– | N | N | N | N | Graph | Graph | N |
| P | N | N | N | H | P | Y | P | P– | N | N | N | Graph | Graph | N |
| User-Defined | Y | Y | | P | | N | Y | P– | P | N | P | Text & Graph | Text & Graph | Planned |
| | H | N | N | P | N | Y | P | N | N | N | N | Text | Text | N |
| P | Y | N | N | P | N | N | P | N | N | N | N | Text | Text & Graph | Planned |
| Y | Y | Y | | N | N | Y | Y | N | N | N | P | Text | Text | Planned |
| Y | Implied by Nested Procedures | Fixed | | P | N | Y | P | N | N | P | P | Text | Text | Y |
| Struc. Entity Substruc. | Global | N | P | NT | N | Y | P | N | N | N | N | Text | Text Tables | Y |
| Sets, Entities Groups, Elements | All Global | N | Y | P | P | N | Y | N | N | P | N | Text | Text, Graph, Matrices | Y |
| Y | Global or Internal | Fixed | N | P | N | Y | Y | N | P | | P | Text | Text | Planned |
| Fixed | Global or Internal | Fixed | | P | P | N | P | P | N | N | P | Text | Text | Planned |
| Free-Form | N | N | N | H | N | P | P | N | N | N | N | Graph | Graph | N |
| Sets | Y | Y | Y | H | N | Y | Y | P | N | N | N | Text | Text | Y |

**LEGEND:**

Y – Yes
N – No
P – Partial
H – Hidden
T – Provided by Automated Tool

Preceding Page Blank - NOT FILMED

others, except perhaps basic content, desirability varies according to the job they are intended to do.

OBSERVATIONS

Judging from the frequency of occurrence in the PDL's surveyed, some form of graphical representation seems popular for the high level description of a system. If this is not available in the language itself, it can be part of the output of an automatic support tool, although the design of good graphical output is by no means easy. The only form of graphical representation for low level design of logic within a module is the much-used but perhaps outdated flowchart. It is not unreasonable to conjecture that the success of structured programming and the consequent familiarity with basic structured programming constructs have rendered textual representation of control flow adequate.

In the realm of information content, i.e., information representable in a PDL, one thing is apparent: no one PDL satisfies all the characteristics set forth. Some emphasize abstractions and interconnections among modules and therefore seem particularly more suitable for global design: MIL75, PSL, HOS, SSL, and structured charts. Others emphasize control flow and are typically more suited to detailed design: IBM PDL, CFG PDL, PDL2, PDDL, and of course, flowcharts. HIPO diagrams, SDL-1, and USL seem applicable to both global and detailed design, but are less satisfactory in the handling of data. Areas not well covered by current PDL's seem to be dynamic structure; data flow and data derivation; abstract data representation; and all auxiliary content, namely, decision details, error handling, performance estimates, and verification information.

Another interesting observation is the number of disparate approaches taken in the development of design languages. On the one hand, there are approaches that start from statement of requirements. Proponents of HOS claim that it can be used to represent requirements, specification, and design of software. PSL, originally designed for problem statement of business information systems, was also meant for logical systems design. Both stop short of detailed design. On the other hand, programming languages are being extended to form design languages: PDL2 is extended from PASCAL; IBM PDL and CFG PDL are both pseudo-code similar to PL/1. Then there is MIL75, whose proponents advocate the use of a language for detailed design (languages for programming-in-the-small) entirely different from languages for global design (module interconnection languages). Much overlap seems to take place among the areas of requirements, design, coding, and documentation, although there seems to be some indication that global and detailed design issues be considered separately. Peters and Tripp, in their report on graphical and non-real-time design representations [Pete76], actually dealt with the two groups separately.

Practitioners spoke of 'requirements languages', 'specification languages', and 'design languages' as distinguished from programming languages, and yet their work resulted in products that do not distinctly separate the associated activities. One basic question is left unanswered: should different languages be used for requirements, design, and coding, or will one language ultimately suffice? If one language suffices, no one has yet come up with such an all encompassing language. If different languages are required because of unresolvable differences in the various functions, then the questions of smooth transition from one stage to the other and verification issues have to be addressed.

Meanwhile, the study of PDL characteristics attests to the fact that most PDL's, with all their imperfections, are definite improvements over flowcharts. They may slowly but surely help point the way toward building better software.

# REFERENCES

Alfo76 Alford, Mack
"A Requirements Engineering Methodology for Real-Time Processing Requirements," 2nd International Conference on Software Engineering, 13-15 October 1976, San Francisco, Calif.

Aust76 Austin, S. L.; B. P. Buckles, J. P. Regan
SSL - A Software Specification Language, SAI-77-537-HU, Science Applications, Inc., Huntsville, Ala., Jan. 1976.

Boyd76 Boyd, D. L.; G. J. Gustafson
The Design Methodology WELLMADE and its Relationship to the Software Generation Process: An Overview, Corporate Research Center, Honeywell, Minneapolis, MN, Oct. 1976.

Cain75 Caine, S. H.; E. Kent Gordon
"PDL - A Tool for Software Design," National Computer Conference, 1975.

Chu76 Chu, Y.
"Introducing a Software Design Language," Proceedings of the 2nd International Conference on Software Engineering, Oct. 1976.

DeRe76 DeRemer, F.; H. H. Kron
"Programming-in-the-large vs. Programming-in-the-small," IEEE Transaction on Software Engineering, June 1976.

Dijk68 E. W. Dijkstra
"The Structure of the 'THE' Multiprogramming System," Communications of the Association of Computing Machinery, May 1968.

Dijk75 Dijkstra, E. W.
"Guarded Commands Non-determining, and Formal Derivation of Programs," CACM 18, August 1975.

Dijk76 Dijkstra, E. W.
A Discipline of Programming, Prentice-Hall, England Cliffs, N. J., 1976.

Earl74 Earley, J
"High Level Operations in Automatic Programming," Proceedings of A Symposium on Very High Level Languages, CM SIGPLAN Notices, March 1974.

Free76 Freeman, P.; A. I. Wasserman
"Tutorial on Software Design Techniques," IEEE Catalog No. 76CH1145-2C, Oct. 1976.

Gaul76 Gaulding, S. N; J. D. Lawson
"Process Design Engineering: A Methodology for Real-Time Software Development," Proceedings of 2nd International Conference on Software Engineering, Oct. 1976.

Gerh76 Gerhart, S. L.; L. Yelowitz
"Control Structure Abstractions of the Backtracking Programming Technique," presented at the 2nd International Conference on Software Engineering, IEEE Trans. on Software Engineering 1976.

Grah73 Graham, R. M.; G. J. Clancey, Jr.; D. B. DeVaney
"A Software Design and Evaluation System," CACM, Vol. 16, Number 2, Feb. 1973.

Hami76 Hamilton, M; S. Zeldin
Integrated Software Development System/Higher Order Software Conceptual Description (Version 1), U. S. Army Electronics Command, ECOM-76-0329-F, NJ, Nov. 1976.

Heim77 Heimburger, D. A.
Private Communication, Jet Propulsion Lab., California Institute of Technology, Pasadena, CA, Jan. 1977.

Hend72 Henderson, P.; R. Snowdon
"An Experiment in Structured Programming," BIT 12, 1972.

Hoar72 Hoare, C. A. R.
"Notes on Data Structuring" in Structured Programming, (Dahl, D. J., Dijkstra, E. W., and Hoare, C. A. R.), Academic Press, New York, N. Y., 1972.

Horo75 Horowitz, E.
"FORTRAN - Can It Be Structured - Should It Be?," Computer, IEEE Computer Society, June 1975.

Kopp76 Koppang, R. G.
"Process Design System - An Integrated Set of Software Development Tools," Proceedings of 2nd International Conference on Software Engineering, Oct. 1976.

Kral75 Kraley, T. M.; J. J. Naughton; R. L. Smith; N. Tinanoff
"Program Design Study," Structured Programming Series, Vol. VIII, IBM, Gaithersburg, MD, RADC-TR-74-300, Vol. VIII, May 1975.

Lisk74 Liskov, B.; S. Zilles
"Programming with Abstract Data Types," ACM SIGPLAN Notices, Volume 9, Number 4, April 1974.

McCr73 McCracken, D. D.
"Revolution in Programming: An Overview," Datamation, Dec. 1973.

McGo75 McGowan, C. L.; J. R. Kelly
Top-Down Structured Programming, Petrocelli, 1975.

Mill71 Mills, H.
"Top-Down Programming in Large Systems," from Debugging Techniques in Large Systems, Prentice Hall Inc., NJ, 1971.

Parn72a Parnas, D. L.
"On the Criteria to be Used in Decomposing Systems into Modules," Communications of the Association of Computing Machinery, Dec. 1972.

Parn72b Parnas, D. L.
"A Technique for the Specification of Software Modules with Examples, Communications of the ACM, May 1972.

PDL75 PDL - Program Design Language Reference Guide, Caine Farber, & Gordon, Inc., March 1975.

| | |
|---|---|
| Pete76 | Peters, L. J.; L. L. Tripp<br>"Design Representation Schemas," Boeing Computer Services, Seattle, Washington, Nov. 1975, Revised April 1976. |
| Reif75 | Reifer, D. J.; L. P. Meissner<br>"Structured FORTRAN Preprocessor Survey," Lawrence Berkeley Laboratories, Univ. of Calif., UCID-3793, November 1975. |
| Reif76 | Reifer, D. J.<br>"Software Specification Techniques - A Tutorial," Digest of Papers, Compcon Fall'76 Conference, Sept. 1976. |
| Rich76 | Richter, M. D.; J. D. Mason, et al.<br>Software Requirements Engineering Methodology, TRW Defense and Space Systems Group, Huntsville, Ala., 1 Sept. 1976. |
| Ross76 | Ross, D. T.; K. C. Shoman<br>"Structured Analysis for Requirements Definition," presented at the 2nd International Conference on Software Engineering, IEEE Trans. on Software Engineering, 1976. |
| Schw74 | Schwartz, J.<br>"Automatic and Semiautomatic Optimization of SETL," Proceedings of a Symposium on Very High Level Languages, ACM SIGPLAN Notices, Mar. 1974. |
| Soft76 | SofTech, Inc.<br>An Introduction to SADT, SofTech document #9022-78, Feb. 1976. |
| Stay76 | Stay, J. F<br>"HIPO and Integrated Program Design," IBM System Journal, Vol. 15, No. 2, 1976. |
| Stev74 | Stevens, W. P.; G. J. Myers; L. L. Constantine<br>"Structured Design," IBM Systems Journal, May 1974. |
| Teic74 | Teichroew, D.; M. J. Bastarache; E. A Hershey III<br>An Introduction to PSL/PSA, ISDOS Working Paper No. 86, Univ. of Michigan, Ann Arbor, Michigan, March 1974. |

VanL76 Van Leer, P.
"Top-down Development Using a Program Design Language," IBM Systems Journal, Vol. 15, No. 2, 1976.

Wils75 Wilson, M. L.
The Information Automat Approach to Design and Implementation of Computer-based Systems, draft, IBM, Gaithersburg, MD, June 1975.

Wils77 Wilson, M. L.
Semantic Analysis and Description: Language Reference Summary, draft, IBM, Gaithersburg, MD, March 1977.

Wirt71a Wirth, N.
"Program Development by Stepwise Refinement," Communications of the Association of Computing Machinery, April 1971.

Wulf76 Wulf, W. A.
"An Introduction to the Construction and Verification of Alphard Programs," Proceedings of the 2nd International Conference on Software Engineering, Oct. 1976.

Zieg73 Ziegler, E. W.
"The UMTA Specification Language and Translator -- An Interim Report," The MITRE Corporation, MTR-6459, McClean, VA, June 1973.

Zieg74 Ziegler, E. W.
"An Introduction to the UMTA Specification Language,"ACM SIGPLAN Notices, April 1974.