A Survey of Program Design Languages (PDLs)

Brian A. Nejmeh

Herbert E. Dunsmore

Purdue University, dunsmore@cs.purdue.edu

Report Number: 85-510
A SURVEY OF PROGRAM DESIGN LANGUAGES (PDLs)

Brian A. Nejmeh

H. E. Dunsmore

CSD-TR-510
April, 1985
A Survey of Program Design Languages (PDLs)

CSD-TR-510

Brian A. Nejmeh and H. E. Dunsmore *

Department of Computer Sciences
Purdue University
West Lafayette, IN 47906

Abstract

The need to improve software productivity and reliability has become an area of increasing importance to the software engineering community. During the early phases of the software life cycle, there is a strong need to emphasize the use of tools which provide a representation of the program structure that can be easily understood and modified. Thus, a significant usage of Program Design Languages (PDLs) as a design tool in practical environments has surfaced in recent years. This paper discusses the desired characteristics of such a design tool and surveys a representative sample of existing program design languages. Finally, a new PDL Environment is proposed for further consideration.

Index Terms - design tools, program design languages, formally defined design constructs, software reusability, software metrics, PDL Environment.

1 Introduction

This paper surveys the state of the art in the area of program design languages (PDLs). Section 2 defines the classical software life cycle, while Sections 3 and 4 of the paper focus more on the software design phase and motivate the need for software design tools. Section 5 of the paper discusses existing software design tools and establishes PDLs as the design

* This research was supported by the National Science Foundation under contract number 1298-50-13985 and International Business Machines under contract number 0266-57-13985.
tool of choice for further consideration. Section 6 of the paper defines PDLs more formally and describes eight desired characteristics of them. Section 7 surveys seven existing PDLs in terms of these desired characteristics. Section 8 of the paper offers a perspective on the future of program design languages in software engineering and proposes a new PDL Environment.

2 The Software Life-Cycle

There are basically six phases of the traditional software life cycle, each phase having a fairly well-defined starting and ending point. However, it should be noted that overlap is typical when making the transition from one phase into the next phase of the life cycle. A brief description of the essential phases of the software life cycle, assuming the feasibility of the project has been established, now follows. For a more complete definition of the software life cycle phases see [Boehm81]:

Requirements Analysis During this initial phase of the life cycle a complete, validated specification of the required functions, interfaces, and performance of the software system is specified. It is during this phase of the software life cycle that the issues of resource needs and preliminary budget estimates should be addressed.

Software Design In this phase of the software development process the determination, specification, and review of software architecture, program design, and data base design necessary to meet the requirements of the software system are completed.

Coding/Implementation In this phase of the development process a complete, verified set of program components are constructed.

Integration During this part of the software life cycle, a properly functioning software system composed of the software components is assembled.

Testing At this point in the software life cycle the verification and validation of the software system is performed.

Maintenance Upon the delivery of the software system to the user, the maintenance phase of the software life cycle begins. During this final phase of the software life cycle the correction, modification, and enhancement of the software system takes place.
3 The Design Phase

Since this paper focuses primarily on the design phase, a closer look at this phase of the software life cycle now follows.

3.1 Transition from Requirements Analysis to Software Design

It has become common practice during the requirements phase of the software life cycle to retain some ambiguity. It is important to note, however, that any items which are vague, missing, undefined, or contradictory should be resolved prior to entering program design. That is, the prevailing attitude of "It is always too early to specify desires, and never too late to make a change" should be prevented; otherwise schedule slippages, low reliability, and cost overruns will be commonplace. Indeed this phenomenon of changing or inadequate requirements is a major obstacle in the software development process [Zolno82].

Once the requirements phase has been thoroughly completed, design should begin. Each of the requirements should be mapped onto segments of the design that satisfy them. In short, a set of requirements form a specification for design.

3.2 The Essence of Software Design

As suggested by DeMarco [DeMarco82], a specific software design can be considered a model of one particular way of meeting the software requirements. The best designs are formal representations of the software to be implemented. DeMarco further suggests that design is the thinking process that has to precede the action of implementation. A software design can be thought of as a "rigorous blueprint for implementation."

There are many different design methodologies. Generally, one uses a structured (i.e., top-down) design which involves the analysis of the functions of a process and their refinement to modules small enough to be effectively developed and managed. Typically, modules should not exceed one page of code; some empirical data suggests that modules in excess of one page of code are very error prone [Potier].

One of the most widely used design methodologies, the Jackson Method [Jack75], involves the following four basic steps.
1. Design the structure of the input data.
2. Design the structure of the output data.
3. Indicate the required mapping from input to output data (i.e., which input yields which output).
4. Design the program to produce the desired transformation from inputs to outputs.

Another popular design methodology is Parnas' [Parnas72, Parnas79] "information hiding" concept. This methodology contends that software systems should be designed with
change in mind, so that they can remain basically the same throughout the life of the system. This will help to decrease the overall effort required to make the changes which eventually come about in a system. Parnas also suggests that the details of how a particular module makes use of information and does its processing should be "hidden" from the person using the module.

Regardless of the design methodology one employs, the design process itself is typically divided into two general phases - the general program design and the detailed program design phase. General program design involves deciding what functions are needed to fulfill the software requirements. Detailed program design involves the consideration of how to implement these functions. In order to make the transition from general to detailed program design, design by iteration is often used. Design by iteration takes place in the following manner:

1) Initial design (General Design) partitions the over-all problem into a set of related functions.
2) Design is then repeatedly refined as more and more detail is added to the functions.
3) The first few iterations constitute general design, while the later iterations constitute the detailed design phase.

3.3 Transition from Design to Implementation

When the software design has been completed and approved, subsequent iterations convert the design into code which constitutes the delivered software product. Each segment of the design can be mapped onto code. The implementation (coding) process yields a more complete understanding of the design, and hence may bring about changes in the design. If this is indeed the case, one should expect to perform several iterations from design to coding.

In short, design forms a specification for implementation. Hence, there should be a strong relationship between the design, and the code implemented from the design. The following five relationships should be maintained in this transition:

1. 1-Many relationship between design constructs and the lines of code necessary to implement the constructs.
2. 1-1 relationship between modules.
3. 1-1 relationship between module interfaces.
4. 1-1 relationship between control flow (While, For, If-Then, etc).
5. 1-1 relationship between intermodule connections.
4 Motivation for Software Design Tools

The need to improve software productivity and reliability has been increasingly important to the software engineering community. A potential step toward improvement in these areas is the emphasis on tools during the early phases of the software life cycle which provide a representation of the program structure and which can be easily understood and modified. The motivating factors for useful software design tools follow.

4.1 Software Development Costs Continue to Soar

Software development costs continue to soar [Boehm81, Vick84, Williams84], hence there is a need for programming environments that support the entire software life cycle. Such programming environments could help increase programmer productivity, thereby reducing costs.

4.2 Schedule Slippages Are Commonplace

In addition to the cost overruns associated with the development of today’s software systems, we are having increasing difficulty in delivering software in a timely manner. Boehm [Boehm81] and others cite cases of 100 to 200% delay in the delivery of software. Effective design tools might help in the delivery of software in a more timely manner.

4.3 Increasing Importance of Software Reliability

Software reliability has become a problem of increased importance as real-time applications of computers emerge in (for example) financial arenas, the military, and the medical profession. In these domains there is an increasing need for software to do precisely what it was designed to do. Thus, a high degree of reliability is essential, so as not to cause huge economic loss or endanger human life. The ability to develop reliable software is of extreme importance. Mendis [Mend82] reports that up to 60% of software errors can be traced back to design errors. Beregi [Beregi84] also found that over 50% of the “post-release” errors in software developed at IBM were found to be traceable to errors in product design. Moreover, it is a well-known fact that the later in the software life cycle a design error is found, the more costly it is to fix the error. Lack of good, reliable design tools can lead to very high software development costs and low levels of software reliability. Thus, better software design tools are essential if we are to produce more reliable software systems.
### 4.4 Monitor Changing Requirements

As was noted in section 3.1, there has been increasing concern over the problem of changing or inadequate requirements. At least one researcher [Zoll82] has pointed out that the use of automated design tools could be of assistance in monitoring the problems of changing user requirements. The "fingerprinting" technique proposed by Wang [Wang84] could be used to monitor changing requirements (and the like) during software development. In general, a closer monitoring of the mapping from requirements to design would give one some indication of the stability of system requirements.

### 4.5 Test Plan Specification

Testing and debugging costs range from 50% to 80% of the cost to produce a software system [Beizer84]. This implies that more effective ways of managing the testing of software systems is certainly needed. A good design tool should enhance our ability to draft an a priori test plan to check such conditions as:

1. interface verification
2. boundary value data exceptions
3. proper terminating sequences
4. constraint violations
5. singular points

When drafting such a test plan one should refer back to the software requirements as well; doing so will lead to a more thorough and complete test plan for the software system.

### 5 Existing Design Tools

For the past several years, researchers have been working on design tools to improve the software development process. However, although many design tools have been introduced, no single design tool has gained wide acceptance. Typically, software developers find themselves in a situation where no single design tool suits all their needs [Bracon83]. This is due to the fact that design is very dependent on software quality factors such as efficiency, reliability, testability, reusability, maintainability, and portability. Thus, the software engineering community is in quest of a design tool which promotes a high level of software quality across the board.

Given the number of software design tools, design is often expressed in a variety of notations, including graphic displays, flow charts, mathematical representations, and programming language-oriented representations. Often multiple notations are used for a single project [Beregi84]. This forces designers to transfer from one design notation to another as they redefine the design. This can lead to problems of increased software defects, decreased
understandability, and increased levels of effort required to produce a software system. Hence, the ideal situation would be the emergence of one superior design tool. A closer look at the different classes of design tools follows.

5.1 Graphic Approaches

Structured flow charts [Nassi73] and HIPO diagrams [Stay76] typify this class of software design tools. These provide a high-level overview of a program. They describe in general terms what data and procedural components make up the program. The advantages of such a design tool are that it yields a design specification which is understandable to nonprogrammers, and it provides a good pictorial display of module dependencies and the like. The disadvantages of using a graphic-oriented design tool are as follows:

1) it is difficult to modify such graphic design representations
2) there are few standards to enforce design and coding that proceed from these tools
3) there is a lack of formal verification tools available for use with graphic design tools
4) it may be difficult for software developers to go from graphic oriented representation of software design to the actual implementation of the software system specified by the graphic representation.

5.2 Requirements Oriented Tools

Requirements/Design tools such as SSL [Buckles77], PSL [Teich77], and SADT [Dick77] fit into this category. Such tools concentrate on identifying input and output data but not on the algorithmic steps necessary to transform the input into the output. At best there are comments describing the steps required for the transformation. In short, such tools cover the requirements phase of software development rather than the software design phase. The focus of this paper is on the design phase of software development.

5.3 HOS - High Order Specification Language

HOS [Hami77] is a design tool often related to "proof of correctness" efforts. When using this design tool, the designer prepares boolean expressions which must remain true throughout some subset of the design. For instance, a designer may specify that some variable must remain positive. Since such a design tool does not address the central problem of describing a program's procedural steps, it yields little insight into the algorithmic structure of the software system.
5.4 Program Design Languages (PDLs)

In order to facilitate the early stages of software development, a large number of design languages have surfaced. Such languages are typically an outgrowth of the original design language PDL, proposed by Caine and Gordon [Caine75]. Such design languages can provide the software developer with the capabilities of algorithmic specification, automation, and verification. Moreover, while flowcharts emphasize explicit flow of control, PDLs have a greater emphasis on program structure. In a recent study [Ramsey83], it was suggested that PDLs have the advantages of lower cost and easier maintainability, relative to flow charts and other graphics-based design tools. The study also suggested that the designer's use of a PDL encourages more detailed specification of the design than does the use of flow charts. In summary, this study appears to provide a fairly strong case for the use of PDLs over flowcharts for the expression of detailed software designs by the designer. PDLs are the design tool of choice for further investigation. A closer look at PDLs now follows.

6 Program Design Languages (PDLs)

6.1 PDLs Defined

A program design language (PDL) is a tool which uses the vocabulary of a natural language and the overall syntax of a programming language (e.g., Pascal). Thus, a PDL can be thought of as "structured English." We view a PDL as a tool to be used during the detailed design phase of software development independent of the design methodology in use. In short, PDLs should allow for the description of algorithms which are to be implemented in software.

6.2 Putting PDLs in Perspective

Figure 1 depicts where PDLs fit in among requirements tools (i.e., SSL, PSL, SADT), very high, high-level languages (VHHLs) such as SETL [Dewar82] and APL [Pakin72], and traditional high-level programming languages such as Pascal, C, or Ada.
There are essentially two noteworthy points to make at this time. First, requirements tools do not specify the algorithmic steps necessary to realize a solution, whereas PDLs encourage the specification of such steps. In this sense, PDLs are a level of abstraction below that of requirements tools such as SSL, PSL, and SADT. Secondly, while PDLs allow for the specification of design, they do not go so far as to allow for the high-level or very high-level formal specification of requirements which would result in a full-scale software system. In the figure above, VHLLs appear to the right of PDLs because while they allow for a lower-level of abstraction than do PDLs, they are not at a level of abstraction entirely below that of PDLs, as is the case with high-level languages.

### 6.3 Desired Characteristics of PDLs

There are many PDLs in use today, but it is not clear that their use will ensure the overall quality of the resulting system. In this section we outline the desired characteristics of a PDL if it is to enhance the overall quality of the design process.

#### 6.3.1 Functional Characteristics

From the functional point of view, there are basically two different types of PDLs, semi-formal and formal PDLs. Semi-formal PDLs have a minimal amount of syntactic and semantic constraint; the terms and expressions used are basically determined by the user. In general, such a language is able to stay close to natural language. On the other hand, formal PDLs have highly constrained syntax and semantics; the user is restricted to using keywords like `READ` and `WRITE`. Some critics of formal PDLs argue that it is difficult for nonprogrammers to understand such a PDL and that they rest on very precisely defined concepts. These may prove sometimes to be too tightly defined to describe unusual or new

---

**Figure 1: Putting PDLs in perspective.**

<table>
<thead>
<tr>
<th>Requirements Tools (SSL, PSL, SADT)</th>
<th>PDLs</th>
<th>Very-High, High-Level Languages (SEITL, APL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-Level Languages (Pascal, C, Ada)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
situations. Many argue that, while a formal PDL is more precise than an informal PDL, a formal PDL is also more difficult to learn and use.

Nonetheless, we believe that a certain amount of formalism should be supported by the PDL. This could include a fixed syntax of keywords that provides the necessary structured constructs, data declarations, and modularity characteristics, as well as a fixed syntax of design constructs used to convey design ideas. The advantages of using formally defined design constructs in the design phase, as opposed to actual code, are really three-fold:

1. Design constructs are more easily understood by all involved parties.
2. Criticisms, suggestions, and modifications can be obtained early in the software life cycle.
3. Software size and effort estimation models based on a PDL version of a program may be obtainable earlier in the software life cycle than is typically the case.

The desired functional characteristics of a program design language are now detailed.

**Promotion of Structured Coding During Implementation** The notation used within a PDL should allow one to state program logic and function in a structured and top-down fashion. This will promote the use of structured programming during the implementation of the eventual system. Thus a PDL should exhibit a syntax of pseudo-natural language that describes processing features such as:

1. data type specifications (i.e., int, real, ...)
2. block structures (i.e., Begin ... End)
3. conditionals (i.e., if-then-else)
4. control structures (i.e., Do While, For Next, ...)
5. formally-defined design constructs (see next section)
6. procedure specifications (i.e., Proc name paramlist)
7. procedure calls (i.e., Call name paramlist)
8. input and output (i.e., Read and Write)
9. error/exception clauses (i.e., On(Cond) Do)

**Use of Formalized Design Constructs** The fundamental difference between a PDL and a high-level programming language lies in the use of high-level primitives to describe a software system. That is, the number of detailed specification points in a PDL-defined software system should be an order of magnitude less than the ultimate software system implemented in some high-level programming language. The advantages of using formally defined constructs during the design phase, as opposed to actual code, were discussed earlier. In addition, the notion of executable designs, as provided through the use of formally-defined design constructs, allows a prototype of the system to exist at an early stage in development which reflects the software design decisions that have been made. This in turn can result
in feedback to the designer on the consequences of a proposed design. Likewise, the use of such formally-defined design constructs could allow users to change requirements early in the software life cycle, before full-scale development proceeds further.

In order to reduce the number of specification points by the previously-mentioned “order of magnitude” from that of the eventual implementation, a PDL should have a set of formally-defined design constructs. For example, Create_Socket and SendTo_Socket might be examples of design primitives for a network application. Moreover, it should be possible to extend the set of formally-defined design constructs when it is deemed that an additional design construct would prove to be beneficial.

**Programming Language Independence** While several researchers have promoted the use of programming language-dependent PDLs [Boehm-Davis82, Sheffield83, Sammet81], only few have realized the harmful effects of using low-level code during design. The danger associated with the use of a programming language-dependent PDL is that there is a tendency to be too detailed in the design, with the net result that the PDL description is (at least in some instances) nothing less than detailed code. In short, a PDL should encourage design rather than coding. Hence, we believe that programming language independent PDLs are essential.

**Good “Code-to” Ability** The PDL should possess constructs which have a good “code-to” ability. That is, while the PDL constructs should allow for program design to be expressed independent of a programming language, they should also lend themselves to be easily coded into many high-level programming languages. It is the ease with which a PDL description is converted into code that makes it particularly suited to representing software design.

**Promote Reusability** A PDL should promote the use of reusable designs and code. Raising the level of abstraction of design constructs is likely to bring about the use of similar design constructs, and hence similar code. One could think about incorporating the use of a macro expansion facility to substitute the code from previous implementations of a given design construct, thereby facilitating reuse of code.

### 6.3.2 Support Characteristics

There are basically three different areas in which PDLs should support or complement the other phases of the software life cycle. The use of a PDL (1) should ensure that the software documentation for a system comes about in a timely manner, (2) should provide automated support tools that assist the developer during the development of the system,
and (3) should lend itself to early software metric and effort modeling. Each of these three different areas are next discussed in more detail.

**Timely Documentation** The use of a PDL should give rise to an excellent design document to guide the implementation. Such a document can also serve the purpose of software documentation during the debugging and maintenance stages of the life cycle. The PDL should help in the goal that design documents be of high quality and be produced at the appropriate time as opposed to after the system has been fully developed.

**Automated Tool Support** The PDL description of a software system should serve several useful purposes, including the following:
1. The description should be machine readable so that various tests, including interface consistency, can be automated at the design phase.
2. It should be possible to produce a design graph of the software system indicating the intermodule dependencies and such.
3. It should be possible to produce a table specifying which constructs used in the PDL software specification are undefined.
4. It should provide a trace facility which displays the mapping from system requirements to design specifications.
5. An Emulation Tool similar to the one described by [Freedman80] should be available. This tool should enable a software designer to "walkthrough his design so that the control logic of the design can be verified." However, it should be pointed out that since the "walkthrough" of design "...does not entail the execution of instructions, the emulator has no way of evaluating whether a test condition is true or not. Thus, whenever a Do While or If statement is encountered, the operator must indicate which course of action to take." While it is difficult to fully test a design during a walkthrough, the emulator does provide a means of ensuring that sequencing errors are minimized. This is likely to result in a higher level of confidence in the correctness of the design. In addition, the emulator can also provide a "data flow diagram for each emulation."
6. A Language Conversion Tool similar to the one described by [Freedman80] should be available to convert automatically as much of the design as possible to code. That is, where possible, the programming language-specific constructs corresponding to the generic design language construct should be generated from the PDL version.

**Early Metric and Model Availability** The earlier in the software life cycle that accurate software metrics are available, the more likely it is that costly redesign and reimplementation can be avoided. Hence a PDL should promote our ability to compute these metrics at the design phase.
The use of software metrics to develop measures of software quality and reliability have become commonplace in the software engineering community [Gilb77, Perlis81]. Commonly used software metrics include Halstead's Software Science measures [Halstead77], McCabe's Control Flow Complexity metric $v(G)$ [McCabe76], and source lines of code. Moreover, DeMarco [DeMarco82] defines several software design metrics that should be computable from a program design language specification. Obtaining accurate estimates of eventual software metric values during the design phase of the software life cycle would be of great economic and managerial benefit.

Through the use of a PDL and the metrics available from a PDL software description, one could use existing software estimation models (e.g., [Boehm81, Thib83]) to predict the effort required to develop the given software system. Likewise, one may be able to deduce accurate expansion factors from a PDL specification to its equivalent coded implementation, thereby yielding accurate software size estimates early in the software life cycle. This too would be of great help to the software community as well, since most of the existing software effort estimation models rely upon software size as the major parameter.

7 PDL Survey

A representative sample consisting of seven different program design languages has been surveyed in this study. The seven different program design languages considered are BYRON (Intermetrics), PDL-Caine, SLAN-4 (IBM), PDL/Ada (IBM), Ada-PDL (TRW), PDL-Arcturus (UC-Irvine), and ADL (Ford Aerospace). In Appendix B of this paper the interested reader can find an example design specification written in each of the seven PDLs. Below we discuss briefly each of the PDLs and then evaluate them according to the desired characteristics outlined in the previous section.

7.1 The PDLs Introduced

The seven program design languages surveyed in this paper are discussed briefly below.

7.1.1 BYRON

BYRON [Gordon83] is the program design language used by Intermetrics as part of their Ada Integrated Environment (AIE). In addition to supporting the Ada programming language, BYRON also adds constructs to Ada programs as legal Ada comments which have some meaning. In turn, these comments are used by a BYRON processor to generate design documentation and to perform some design analysis.

There are several different BYRON constructs, each of which is prefixed by a -- . The first type of BYRON construct, directives, takes on the form -- KEYWORD TEXT.
There are several different BYRON KEYWORDS which address issues of data abstraction, program description, timing requirements, exception handling, and performance analysis. The TEXT part of each BYRON directive is used to detail or describe the given directive more thoroughly.

The second type of BYRON construct, flags, take on the form ---, and are used to denote the scope of BYRON statements. That is, the lines processed by the BYRON processor are those which appear between the two flags. It should be noted that there are also several other characters which can follow the BYRON prefix construct that serve a similar purpose.

7.1.2 PDL-Caine

PDL is the program design language outlined by Caine and Gordon [Caine75]. This program design language is especially significant in that it was the first to appear in the literature. For brevity, in the remainder of this section we shall refer to PDL-Caine as simply "PDL".

PDL was developed for the "production of structured designs in a top-down manner." It allows the designer to specify a complete design - including interface definitions, procedure definitions and calls, data declarations, control blocks, varying level algorithm processing specifications, and error definitions. PDL is supported by a processor which takes as input "control information plus designs for procedures" and produces as output a working design document.

7.1.3 SLAN-4

The language SLAN-4 [Beich83, Beich84] was developed at IBM as an aid in the specification and design of software systems. "SLAN-4 is a language spanning the complete range from an almost natural language to an almost compilable language. It can be used as a software specification, design, communication, and documentation tool." [Beich83, p.558]

SLAN-4 allows for algebraic and axiomatic specification, generic data types, and modules. The programming constructs supported by SLAN-4 are very similar to those used in high-level programming languages such as Pascal and Ada. That is, all of the basic data types and operations associated with them, operations on sets and lists, and control structures from the traditional programming languages are part of the SLAN-4 language. Also, SLAN-4 introduces a semaphore construct as a synchronization tool.

In addition to offering constructs for detailed specification, a subset of the SLAN-4 language allows the designer to express software design as well. This sublanguage of SLAN-4 is called "pseudocode", and contains control structures for sequential and concurrent processing, as well as assignment and procedure call constructs. In addition, pseudocode supports
the assign, assert, and goto constructs. Furthermore, comments may appear anywhere within a design specification. Such comments are started by a "<<" and ended by a ">>". Thus pseudocode "allows a user of SLAN-4 to start with a specification written in natural language, but in a structured way." Pseudocode is designed to offer "a way of presenting algorithms independently of the language in which the final program is to be written." Then, as the development of the system progresses, these informal pseudocode specifications can be made more precise by formalizing the informal constructs (i.e., comments) by making full use of the SLAN-4 language.

7.1.4 PDL/Ada

PDL/Ada [Sammet81] is a program design language (based on Ada) which was developed at IBM. The design language includes formal specifications for procedures, control structures, assignment statements, functions, procedures, limited generics, data declarations and data typing, modules (packaging), and comments. A comment starts with a double hyphen (--) and is terminated by the end of the line.

A significant motivating factor for PDL/Ada is the use of an Ada translator as a design tool. Although the translator cannot generate code from PDL/Ada design specifications, it can perform syntax checking and type checking in parameter lists of invoked procedures. Thus, PDL/Ada design specifications are acceptable by an Ada translator. However, abstract functions and predicates written simply as comments would not be accepted by a translator. Thus, a special means of specifying such abstractions was devised. In particular, to combat such difficulties, PDL/Ada has a specially-defined predicate CONDITION and a null procedure called THENPART. The figure below illustrates the point.

Unacceptable by Ada Translator

if -- new satellite detected
 then
 -- compute its orbit
endif;

Acceptable by Ada Translator

if CONDITION -- new satellite detected
 then
 THENPART; -- compute its orbit
endif;

Thus, by using the two previously mentioned constructs, CONDITION and THENPART, the original design notation is acceptable to an Ada translator.

7.1.5 Ada-PDL

The program design language Ada-PDL [Spoon84] was developed at TRW for recording designs at all levels of the software design/development process. The developers of Ada-PDL
state that the two primary objectives of Ada-PDL are (a) to provide a more complete and structured PDL than PDL-Caine so that more detailed automated analysis is possible, and (b) to address and exploit the introduction of Ada in the development of software systems. Ada-PDL maintains the flavor of PDL-Caine and adds new capabilities to support software design by providing additional items such as structured data, hierarchical design structures, detailed interface specifications, and inter-module dependencies.

While Ada-PDL is based on the programming language Ada, it is not itself Ada. Ada-PDL is a combination of formal and informal constructs. The formal constructs of Ada-PDL must be written using specific syntax much like Ada, while the informal constructs are “almost free of syntactic constraints, allowing nearly all English constructions.” Ada-PDL offers the designer a rich set of constructs including conditional and iterative constructs, data definition/naming constructs, modularity constructs (e.g. subprograms, packaging, tasking, and modules), and context specification constructs (e.g., import, with, use, and separate) all of which are Ada-based.

In addition to these Ada-specific constructs discussed in the previous paragraph, Ada-PDL has two other constructs: design narratives and comment constructs. These two constructs place virtually no restrictions or constraints on Ada-PDL. The difference between these two constructs lies in the fact that “if an Ada-PDL keyword begins a line, then that line is identified as part of a more formal Ada-PDL construct (name-declaring or conditional) that may, itself, allow design narrative following the keyword.” In contrast, the second kind of text that may appear in both name-declaring and algorithm constructs is a comment. A comment begins with two hyphens (---) and includes all of the text until the end of the line is reached.

Ada-PDL is supported by the Ada-PDL processor that analyzes input design text to do such things as producing a “variety of reports useful during the design phase, determine those aspects of design that are deferred/incomplete, check inter-module interfaces for consistent use, and maintain and control a database of design library units.”

7.1.6 PDL-Arcturus

Arcturus [Tadman84, Standish83a] is an Ada-based programming environment under development at the University of California - Irvine. One component of the Arcturus environment is a program design language. The Arcturus program design language uses normal Ada syntax forms in which text in braces can be substituted for declarations, expressions, names, statements, or types. Some examples of PDL comments follow.

```ada
while (Interval is non empty) loop

raise {an exception to be defined later};
```
Procedures, functions, packages, types, etc. with PDL comments are defined in a manner similar to Ada. Subprograms and packages containing PDL comments can be executed in a manner such that any PDL comment except a statement will cause the “break package” to be invoked; thereby halting program execution. In situations where a PDL statement is encountered, the break package is invoked, and execution may continue just as if the procedure “break” had been called.

In addition to allowing PDL constructs to be included in a program, the Arcturus environment also provides for a “Rapid Prototyping Language.” This language allows PDL comments to be refined by defining macros for each PDL comment. In this case, the PDL comment is referred to as a “calling form.” When a calling form is encountered during the execution of a program and no macro is defined for such a PDL comment, then the break package, as previously described, is entered. If, however, a calling form is defined, it is executed and “the results of the macro is substituted into the program before execution continues. The result actually replaces the calling form so that each macro is executed and replaced only once during execution.” Arcturus also provides a mechanism whereby the calling forms in a program can be replaced without executing the program. An example of a calling form follows.

Given the definition of the following macro,

```
macro {print list (seqlist)} return stmtnode is
  begin
    return genstmt('for i in 1.. $seqlist'last loop '' &
    ''put(list(i)); put ('' ');'' &
    ''end loop;'' &
    ''newline;''));
  end;
```

The use of the calling form in a program such as

```
{print list (list)};
```
results in the following macro expansion of this PDL comment in the program:

```pdl
for i in 1..list'last loop
    put (list(i)); put (' '); end loop;
```

The Arcturus environment also provides many support tools such as pretty-printing, directory listings, and editing tools.

### 7.1.7 ADL

ADL is an Ada-based program design language developed at Ford Aerospace and Communications [Thompson83a, Thompson83b] to provide detailed software design specifications. ADL supports a subset of the Ada programming language in which correct Ada syntax is checked. In particular, ADL supports a large subset of the data types and control structures found in Ada. A rich set of arithmetic and string operators are also supported by ADL. Furthermore, the Ada language constructs - procedures, functions, and packaging are also part of ADL. In order to permit the verification of detailed design interfaces, ADL requires all procedures and function calls, along with passed parameters, to be written in compilable Ada code.

ADL provides a TBD ("to be done") construct which allows for the fact that during early design some interfaces may not be specified. In order to alleviate the problems inherent with interface consistency checking in this case, ADL has a special package PACKAGE_TBD defined, in which types, records, and arrays will be defined as TBD as well. Another facility offered by PACKAGE_TBD is the procedure CALL_TBD, which has no parameters and performs no action. CALL_TBD allows the designer to call a procedure without concern for whether or not the interface with that procedure has changed since it was last called. The benefit of using the CALL_TBD procedure call is that it eliminates the necessary recompilation which result when the interfaces change. In addition, variables within procedures can later be declared to be one of the above types once the necessary information becomes known.

Another Ada construct which is used in ADL is the comment statement which contains text. The developers of ADL suggest that "ADL comments [should] be included in structured English format for all detailed processing required of a procedure or function. These comments [should] include all processing requirements in sufficient detail to allow independent coding of modules in [various programming languages]."

The NYU interpreter is the only support tool available, and is used to perform syntax and semantic checks of ADL source code.
7.2 Evaluation of Functional Characteristics

The evaluation of functional characteristics for each of the Program Design Languages included in this survey now follows. A summary of the evaluation of functional characteristics appears as Figure 2 in Appendix A.

7.2.1 Promotion of Structured Coding During Implementation

**Above Average: BYRON, SLAN-4, PDL/Ada, PDL-Arcturus, ADL**  All of the PDLs in this category, with the exception of SLAN-4, are closely tied to the Ada programming language. Hence such program design languages strongly promote structured coding in the same way that Ada supports structured coding. However, it could be that some of the Ada-specific detailed constructs which are a part of these PDLs may be too difficult, if not impossible, to simulate using structured coding when the implementation language is something other than Ada. For instance, the Ada packaging or tasking construct would be very difficult, if not impossible, to implement if Fortran were the implementation language.

The software designs resulting from the designer’s use of SLAN-4 may promote the use of structured coding during implementation. The reason for this is that SLAN-4 provides many of the control structures, procedure specifications, and data declarations provided by the traditional high-level languages. However, it should be pointed out that SLAN-4 does allow the use of the “goto” statement which is not a construct which promotes structured coding [Dijkstra68]. Likewise, SLAN-4 supports concurrency and semaphore constructs which are not generally available in high-level programming languages.

**Average: Ada-PDL**  While Ada-PDL is bound to the data declarations and module specifications of Ada, it does not support the detailed programming constructs of Ada. As such, it does not strongly promote structured coding during implementation.

**Below Average: PDL-Caine**  PDL-Caine may do a reasonable job in promoting the use of structured coding during implementation since it provides control flow constructs, procedure specifications, and data declarations, as part of its own language. Moreover, the constructs provided by this program design language are available in any high-level programming language. However, it is never stated that the designer is required to use any part of these structured constructs. Hence, we conclude that the designer is at liberty to design without the use of structured control flow constructs and the like. Thus, the use of this PDL may not promote structured coding during implementation.
7.2.2 Use of Formalized Design Constructs

Above Average: PDL-Arcturus  More so than with any other program design language included in this survey, we found PDL-Arcturus to exhibit the capability of allowing formally-defined design constructs to exist. This is not to say that the calling form annotation of formally-defined design constructs meets our criterion established in the “Desired Characteristics” section of this paper; it simply allows for, but does not require, such formally defined design constructs to exist. Nonetheless the fact that they have included such a capability in their PDL implies that it would require only simple modification of the PDL-Arcturus language to require an exact and formal syntax for all design constructs. This modification will be outlined in the “Future Directions” section of the paper.

Average: BYRON, Ada-PDL  While neither BYRON nor Ada-POL provides formally-defined design constructs, both provide a means of inserting narrative design text in well-defined locations in a program.

Byron supports an algorithm directive construct which can be placed in certain places throughout a program. However, since any text may appear within an instance of an algorithm directive, this implies that it is quite possible to do software implementation during the software design phase of the development. Several examples in the literature describing BYRON were such that the PDL description had a number of specification points far less than the corresponding software implementation. But, this certainly may not be the case in general.

Similarly, Ada-POL provides two such constructs, namely design narratives and comments. While the locations in which such constructs can appear is limited, the fact that the text which may appear within either of these constructs is virtually unlimited, leads us to the conclusion that the Ada-PDL environment, as it is currently defined, does not have the capability of recognizing or defining formal design constructs.

Below Average: PDL-Caine, SLAN-4, PDL/Ada, ADL  When designing software systems with any of these program design languages, there is no notion of formally defined design constructs. The only trace of a design construct is in the use of comments describing the processing which a given segment of code is to achieve. However, the fact that such comments can appear anywhere within the program and are written simply using English text, certainly does not demonstrate any formal notion of a design construct.
7.2.3 Programming Language Independence

Independent: PDL-Caine, SLAN-4, Ada-PDL. Both PDL-Caine and SLAN-4 are programming language-independent program design languages. However, in the case of SLAN-4 we conclude that since the syntax and structures it supports are so unconventional, it does not have all of the advantages of programming language-dependent PDLs as previously described. On the other hand, clearly PDL-Caine possesses the advantages previously described.

While Ada-PDL is dependent upon the data declaration and module specification constructs of the Ada programming language, it does not support the use of detailed Ada coding constructs. It is this independence from Ada-specific coding constructs which allows it to be classified as a programming language independent PDL.

Dependent: BYRON, PDL/Ada, PDL-Arcturus, ADL. These program design languages are very much tied to the Ada programming language. The fact that these program design languages possess this strong programming language dependence surely implies that they have the shortcomings associated with programming language-dependent PDLs that were described earlier.

7.2.4 "Code To" Ability

Above Average: PDL-Caine, Ada-PDL. Since PDL-Caine and Ada-PDL are not tied to a particular programming language, there is a good possibility that these program design languages will yield a good "code to" ability at the implementation level. However, the lack of formally defined design constructs implies that it is quite possible for ambiguous design descriptions to exist. Such a design specification would clearly not possess a good code-to ability. Likewise, the fact that Ada-PDL is not closely tied to any particular programming language and that it supports a rich set of structured constructs, implies that it will possess a good "code to" ability.

Average: PDL-Arcturus. PDL-Arcturus has the possibility of possessing a very good "code-to" ability, provided liberal use of "calling forms" exists. This is because such constructs provide a clear, concise description to be mapped into an implementation. However, care must be taken when making such a statement. For, if minimal use is made of
calling forms and excessive use made of detailed Ada constructs, PDL-Arcturus may not possess a very good code-to ability.

Below Average: BYRON, SLAN-4, PDL/Ada, ADL Since each of these program design languages is supported by a rich and healthy set of operators, it is unlikely that a good code-to ability would exist if the program design specification is to be implemented in a language other than the one supported by the given program design language. The point here is that it may be very difficult to take constructs such as Ada Packages and Tasking from the design specification, and implement such constructs in some other language.

7.2.5 Promote Reusability

Above Average: PDL-Arcturus PDL-Arcturus has the most promise in terms of its ability to promote software reusability. This is due to the fact that the calling form macro facility allows one to associate coded implementation with a design construct. Moreover, it provides a means by which it can determine whether or not such an exact match exists between a PDL comment currently being used, and a PDL comment defined and used in the past. This decreases the likelihood of having multiple formally-defined design constructs which perform the same function. Also, providing an on-line document describing the exact syntax of existing formally-defined design constructs, and their associated semantics, would also promote reusability. In short, PDL-Arcturus has a strong potential to promote reusability. In fact, the developers of PDL-Arcturus cite a case in which a 62 % software reuse factor was realized in the construction of a prototype system [Standish83b].

Average: ADL The only notion of reusability addressed in ADL is in the “data dictionary packages” which allow a designer to make reference to data in the dictionary without redefining the data types, range of values, etc. within each package. However, due to the lack of formally-defined design constructs in ADL, it is not possible to automate the reuse of design constructs and their respective implementations.

Below Average: BYRON, PDL-Calne, SLAN-4, PDL/Ada, Ada-PDL None of these program design languages provide a mechanism by which a designer can formally define design constructs. This suggests that they do not promote the reuse of code. That is, given a high-level design construct and an implementation for this construct, one should be able to associate the original implementation of the design construct with later uses of the same design construct. However, the lack of formally-defined design constructs could
result in a situation whereby several different design descriptions exist for the same design construct, thereby making it very difficult to detect that reuse is even possible.

7.3 Evaluation of Support Characteristics

The evaluation of support characteristics for each of the program design languages included in this survey follows. A summary of the evaluation of support characteristics appears as Figure 3 in Appendix A.

7.3.1 Timely Documentation

Above Average: PDL-Arcturus PDL-Arcturus supports the use of formally-defined design constructs (calling forms) at a high-level of abstraction. This implies that this design language is likely to yield timely and meaningful documentation. However, care must be taken not to overemphasize this aspect of the Arcturus environment. It is quite possible that a designer may choose to use a minimal number of calling forms, in which case the design specification produced is very close to actual Ada code. Clearly such a design specification would not yield very timely or meaningful documentation.

Average: BYRON, Ada-PDL Both of these design languages provide a means of formally defining locations in a program where narrative design constructs can exist. Thus, it is possible for these design languages to yield reasonable documentation. This is because the use of design narratives is encouraged at certain locations in a program. However, the fact that the degree of detail specified within each of the design constructs is unlimited, implies that the design specifications written in either of these design languages may be nothing less than complete programs written in Ada. Surely if this is the case, the use of BYRON or Ada-PDL will not result in reasonable documentation.

Below Average: PDL-Caine, SLAN-4, PDL/Ada, ADL It is unlikely that the use of any one of these design languages would yield timely or meaningful documentation. None of these design languages provide a means of formally defining design constructs; program comments are the only means by which design can be conveyed. Thus, it is likely for design specifications written in these design languages to be nothing less than complete programs written in the low level primitives provided by the given design language. Thus, we conclude that the degree to which these program design languages yield timely and usable documentation is a function of the level at which the specification is written. Hence, since each of these design languages supports low level primitive operations, complemented
by the fact that they do not support formally-defined design constructs, implies that such
design languages will yield inadequate documentation.

7.3.2 Automated Tool Support

Above Average: BYRON, Ada-PDL

Both BYRON and Ada-PDL are supported
by a very healthy set of tools to aid in the design of software systems. A brief discussion of
the tools available for each of the design languages now follows.

BYRON is supported by five different tools -

1. The "Analyzer" is a tool which checks BYRON source code for correct Ada syntax
and semantics (optional). If it does not detect any errors it stores an internal representation
of the source code in a program library. The analyzer also checks for the proper use of
BYRON constructs.

2. The "Call Tree" is a tool which displays the functions and procedures that a specific
program unit calls. In addition, this tool also displays the functions and procedures which
call the specific program unit.

3. The "DataDict" is a tool which displays information such as declaration of types,
subprograms, packages, tasks, entries, or any combination, in a selected set of program
units.

4. The "DepTab" is a tool which displays the dependencies among program units. It
generates a report which shows the units which would have to be recompiled if a given unit
is recompiled.

5. The "UserMan" is a tool which creates a report describing the external interface
to a subprogram or package in the program library. This report also displays all of the
information necessary for someone who wishes to use a predefined subprogram or package.

Ada-PDL has a processor capable of producing several different reports. There are
basically two different types of reports which can be generated by the processor: "Pretty
Printed Listings" and reports derived from the structure of the design. A summary of the
reports available by Ada-PDL now follows.

1. The "Cross Reference Listing" report contains entries for all names that are refer­
enced in the specified design unit, along with their declared type, the line on which each
name was declared, and the number of the line(s) which reference the name.

2. The "Name Directory Listing" report contains the names declared in a specified
design unit, along with the location of their declarations. In addition a set of attributes for
each name, including its type, is provided.

When using Ada-PDL, the designer also has the option of producing a simple source
listing in which an exact copy of the input stream is produced; or a "Pretty Printer Listing,"

24
In which keywords are highlighted and structure indentation takes place.

In addition to the above, the following reports are currently under development: Module Dependency Reports, Subprogram Call and Tasking Entry Call Hierarchy Reports, and a Parameter Checking Report to validate the correctness of module interfaces.

**Average: PDL-Arcturus**

PDL-Arcturus is supported by a reasonable set of tools to aid in the designing of software systems. In general, the same tools which process normal ADA programs are available for use when expressing design in PDL-Arcturus. Such tools include syntax analysis, pretty program listings, execution-time performance monitoring via a color graphics monitor, and a template-driven (syntax-directed) ADA text editor to aid in the stepwise refinement of programs.

**Below Average: PDL-Caine, SLAN-4, PDL/ADA, ADL**

The program design languages in this category provide a minimal amount of tool support. A closer look at the tool support available with each of these design languages follows.

There is a limited amount of tool support when working in the PDL-Caine environment. In particular, the PDL processor yields a document consisting of the following five things:

- A title page
- A table of contents
- The body of the design in pretty printed form
- A "reference tree" showing how design segment references are nested
- A cross-reference listing indicating the page and line number where each design segment is referenced.

There is really no mention of any tool support in either one of the two references on SLAN-4 available to us for this survey [Beich83, Beich84]. The only form of tool support in existence seems to be an interface verification facility and a syntax/semantic checker. The same is true for PDL/ADA.

There is a limited amount of tool support available when designing with ADL. The first tool allows items that have been defined to be stored in a data dictionary. In turn, modules which later reference previously-defined items can simply reference the data in the dictionary without redefining the data types or range of values. Secondly, the NYU ADA Interpreter is used to check ADA syntax and semantics. While no mention is made of the fact that it is feasible to produce a deferred development report of the TBD constructs in an ADL specification, certainly such a report would be noteworthy.

### 7.3.3 Early Metric and Model Availability
Above Average: PDL-Arcturus  
Provided that extensive reuse of design calling forms are used, timely and accurate software metric estimates could be made based on a PDL-Arcturus design specification. However, if extensive reuse of calling forms does not take place, then PDL-Arcturus would have to be classified along with the other Ada specific program design languages below.

Average: BYRON, SLAN-4, PDL/Ada, ADL  
All of these program design languages allow one to accurately estimate the software metrics of systems to be developed in Ada. However, the fact that these design languages allow for detailed Ada constructs to be used, implies that accurate software metrics will come about too late in the development process to be of much value. Furthermore, the usefulness of metric estimates based on an Ada-specific design language could prove to be of minimal value in cases where a language other than Ada is used for implementation.

It should be pointed out that the developers of PDL/Ada [Sammet81] state that the software design metrics "may be closely related to those for the [implementation] language. Thus if one is counting source lines of code or software science metrics, this can be done automatically on the design statements. This then measures the design, not the implementation resulting from it." The developers fail to note the fact that the correlation between software metrics computed at the design phase and those computed on the resulting implementation is very important. A poor correlation between software metrics computed during the design phase, as compared to those metric values obtained from the resulting implementation, is not very helpful. If such a scenario is realized, one can be lead to erroneous conclusions concerning the reliability of the system or testing effort required of the actual system.

Below Average: PDL-Caine, Ada-PDL  
Due to the fact that both of these PDLs are supported by control structures, procedure calls, and data declarations, it may be possible to derive reasonable estimates for some software metrics based on the design specifications they yield. However, the level at which the design specification is written will determine how early in the development process these metric estimates are available, as well as the accuracy of such estimates. Thus, since there is no formally-defined syntax for detailed constructs of these PDLs, we conclude that it is unreasonable to consider software metric modeling when using such program design languages.

8 Perspectives for the Future

There is a widespread demand for safe, verifiable, and reliable software to be delivered in a timely manner. It has been a goal of this paper to demonstrate that program design
languages can make a valuable contribution toward this goal. This survey has discussed the positive and negative aspects of existing program design language. While no single program design language emerged as the PDL of choice, many positive PDL attributes have been revealed through this survey. We believe it is the union of all the positive attributes from the various PDLs, complemented by some additional features, which will give rise to a class of PDLs that will emerge as the superior software design tools. A discussion of such a PDL Environment follows.

8.1 The PDL Environment

The ideal program design language environment would be one which supported software design assuming a number of implementation languages. That is, for purposes of module, data, and control flow specification, we believe such constructs should be programming language-dependent (as is the case with Ada-PDL, assuming the implementation language is Ada). If this were not the case, module interface verification, accurate metric estimates, and the promotion of structured coding during implementation would be difficult for any universal PDL to achieve. However, we do not believe a PDL Environment should support the low level, detailed constructs of any particular programming language.

Suppose, for purposes of illustration, that the implementation languages supported in a given development environment include Pascal, C, and Ada. In such a case the PDL Environment is depicted in Figure 4.
In this case the PDL Environment should support the module, data, and control flow specifications available in each of these languages. Then, on command, the designer could invoke a particular instance of the PDL Environment based on the implementation language to be used. In such a case only the module, data, and control flow constructs supported by the specified design language would be permitted during this instance of the PDL Environment. In situations where the implementation language is unknown or unsupported in the PDL Environment at the time of software design, a generic PDL instance should be supported. Such a PDL instance should enable the specification of modules, data, and control flow constructs in a language independent, but formal manner.

8.2 Design Constructs

Most PDLs do not support formally-defined design constructs. Thus, it is impossible to enforce a “design rather than code” philosophy during the design phase when using such PDLs. In terms of supporting formally defined design constructs, clearly PDL-Arcturus stands out as the PDL of choice. Likewise, the formally-defined constructs of PDL-Arcturus promote the reuse of code resulting from the implementation of such constructs. However, the fact that it allows only Ada-specific low level constructs to be used during implementation, implies that PDL-Arcturus only supports reuse of Ada code. Thus, a more general notion of formally-defined design constructs would be of great value.

We promote the use of formally defined design constructs in which the language is supported in the environment. However, using the method of PDL-Arcturus calling forms is thought to be quite adequate and generalizable beyond the Ada domain.

There are basically three key issues to be researched with respect to the development of design constructs. These three issues are the abstraction, the interface, and the transformation of design constructs.

Abstraction The key to success in a PDL Environment is to develop a way of abstractly specifying the basic operations and objects while still promoting the desirable characteristics of the PDL. In situations where formally-defined design constructs exist and are
accompanied by an implementation, the abstract specification of the design construct is not a problem, namely it is simply the formally-defined calling form. However, in situations where no formally-defined design construct exists for the given design construct, a mechanism of describing design, while avoiding environmental specifics such as programming languages, operating systems, and machine is needed. The description of such constructs, at an appropriate level, warrants further consideration.

Interface Another aspect of design constructs is the means by which the designer interfaces with formally-defined design constructs. The interface must provide an on-line mechanism to enable, on demand, a specification of the functionality of the construct, as well as the environmental conditions necessary for the construct to function properly. Likewise, such an interface must be accompanied by a mechanism to catalog, retrieve, and update formally-defined design construct libraries.

We believe that the UNIX (tm of Bell Laboratories) “include” facility can be exploited to build a reasonable interface. For example, suppose a group of network oriented formally-defined design constructs have been developed in the C programming language and stored in the file “netfddc.C”. Then, in order to access these formally defined design construct definitions during the design of a network application using PDL-C, the designer would simply put the statement “include <netfddc.C>” in the PDL-C design specification in order to legally make use of the formally defined design constructs which appear in this file. The location of a programming language-dependent library of formally defined design constructs in the PDL environment is depicted in Figure 5.

It may also be possible to use formally-defined design constructs written in language X when using PDL-Y in the PDL Environment. That is, it may be possible to exploit existing software technologies to use a formally-defined design construct written in one language, while designing with a PDL based in a different language. Several researchers [Jones85, Tichy80, DeRemer76] have argued that structuring a large collection of modules to form a system is a distinctly different activity from that of constructing the individual modules. Correspondingly, they argue that distinctly different languages should be used for the two activities. An advantage of using a separate language for describing module interconnectivity is that the modules themselves may more easily be coded in different languages. Systems such as MIL [DeRemer76], INTERCOL [Tichy79, Tichy80], and Matchmaker [Jones85] could each provide the capabilities for composing a large system from modules written in several different programming languages through the use of a module interconnection language. Such module interconnection techniques should be given further consideration in order to allow formally-defined design constructs written in one language to be used by a person designing in another language.

The on-line mechanism could be to print the header and comment lines from implemented formally-defined design constructs based on an input generic command. For ex-
ample, the generic command "Socket" might result in the headers and comments from the functions Create_Socket, Bind_Socket, SendTo_Socket, and RecFr_Socket (assuming such functions are defined in the library of formally-defined design constructs) to appear in response to this generic inquiry.
Transformation: The transforming of high-level, abstract design specifications into efficient and correct implementations of formally-defined design constructs in various high-level languages is of paramount importance for the development of an effective PDL Environment. Furthermore, the higher the level of abstraction, the more difficult it is to provide an effective transformation to code.

8.3 Tool Support

Both BYRON and Ada-PDL provide a healthy set of design support tools. In addition to the tools available in these environments, several other tools should be supported by the PDL Environment. Surely an emulation tool, as described in section 6.3 would be beneficial. Second, a mechanism to produce a design graph detailing module interconnectedness should be part of the tool environment. Another tool worth exploring would be one that allowed for the expansion of formally defined design constructs to be turned "on" or "off". This is because managers may not be interested in detailed implementation, but merely in the abstract design specification of a software system. In such cases the formally defined design construct expansion mechanism can be turned off, thereby allowing the designer to "hide" the low level design details. On the other hand, when doing design reviews, one would want to turn the expansion facility on to enable the fully developed design to be revealed for review. Third, the whole notion of TBD constructs available in ADL should be made available in the PDL Environment. Such a construct could be exploited to generate
a "Deferred Development" report of TBD constructs in a PDL specification. Finally, a mechanism to show the mapping from requirements specifications to design specifications is needed to ensure the traceability of requirements to design.

8.4 Software Metric Availability and Modeling

The difficulty of obtaining early and accurate software metric estimates through PDL specifications was emphasized earlier. For the PDLs included in this survey, Figure 6 depicts this difficulty.
Low-Level PDL (e.g., ADL)

- accurate metric estimates, but late in the software life cycle
- desired level of PDL (e.g., none exist)
- early in the software life cycle, but inaccurate metric estimates

High-Level PDL (e.g., PDL-Caine)

Figure 6: Difficulty in obtaining usable software metrics based on PDLs.

The difficulty in obtaining usable metric estimates with Low-Level PDLs (e.g., ADL) is that while they yield very accurate metric estimates, such estimates are available only upon completion of the coding phase. Thus, such late metric estimates do not point out problems in design until the end of the coding phase. Clearly this is too late in the software life cycle to really make a significant difference in terms of whether or not to redesign.

On the other hand, high level PDLs (e.g., PDL-Caine) have the problem of yielding inaccurate software metric estimates based upon the design specifications which they yield. That is, it is difficult to accurately predict the characteristics of the corresponding implementation, and hence software metrics, based upon informal narrative text. Perhaps liberal use of formally-defined design construct calling forms in the proposed PDL Environment will enable early and accurate software metric estimates to take place.

Another area of the PDL Environment to be exploited with respect to software metrics, is monitoring the completeness of software design by observing the DC (design completeness) ratio defined as

\[
DC = \frac{\text{Metric Estimate Based on PDL Envir. Design Specification}}{\text{Actual Metric Value Based on Implementation}}
\]

Thus, if DC equals 1, the design specification was very complete. As DC decreases from 1, the design specification does not accurately represent the resulting implementation. In the latter case, clearly this is a trigger to management that incomplete software design specifications are being developed. In addition, if the ratio DC( V(g) ), that is the DC ratio
for McCabe's V(G) metric [McCabe76], is not close to 1, this could result in very grave consequences if management has based testing effort distribution on the V(G) estimate based on the design specification.

8.5 Shifted Distribution of Life Cycle Effort

The use of a PDL Environment, and its associated tools, will most likely result in more effort devoted to the requirements and design phases of the software life cycle. This is due to the more thorough assessment of requirements and formalisms associated with the use of a PDL Environment. However, we suspect that the correct usage of the PDL Environment will decrease the coding, testing, and maintenance effort, and hence the overall effort required to develop a software system, while increasing the overall quality of delivered software. Surely this would be a well received improvement in the software development process.

8.6 Summary

Among the current software design tools, program design languages are certainly the most advanced and most useful. Such a design tool has been demonstrated to be supportive during several phases of the software life cycle in addition to the design phase. As such, program design languages are likely to become a standard tool in the integrated software engineering environments of the future.
Appendix A

Summary of PDL Evaluation

<table>
<thead>
<tr>
<th>PDL Name</th>
<th>Structured Coding</th>
<th>Design Constructs</th>
<th>Prog. Lang. Code To Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYRON</td>
<td>AA</td>
<td>AV</td>
<td>DE</td>
</tr>
<tr>
<td>PDL-Caine</td>
<td>BA</td>
<td>BA</td>
<td>IN</td>
</tr>
<tr>
<td>SLAN-4</td>
<td>AA</td>
<td>BA</td>
<td>IN</td>
</tr>
<tr>
<td>PDL/Ada</td>
<td>AA</td>
<td>BA</td>
<td>DE</td>
</tr>
<tr>
<td>Ada-PDL</td>
<td>AV</td>
<td>AV</td>
<td>IN</td>
</tr>
<tr>
<td>PDL-Arcturus</td>
<td>AA</td>
<td>AA</td>
<td>DE</td>
</tr>
<tr>
<td>ADL</td>
<td>AA</td>
<td>BA</td>
<td>DE</td>
</tr>
</tbody>
</table>

Figure 2: Summary of Functional Characteristic Evaluation.

<table>
<thead>
<tr>
<th>PDL Name</th>
<th>Timely Documentation</th>
<th>Tool Support</th>
<th>Metric Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYRON</td>
<td>AV</td>
<td>AA</td>
<td>AV</td>
</tr>
<tr>
<td>PDL-Caine</td>
<td>BA</td>
<td>BA</td>
<td>BA</td>
</tr>
<tr>
<td>SLAN-4</td>
<td>BA</td>
<td>BA</td>
<td>AV</td>
</tr>
<tr>
<td>PDL/Ada</td>
<td>BA</td>
<td>BA</td>
<td>AV</td>
</tr>
<tr>
<td>Ada-PDL</td>
<td>AV</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>PDL-Arcturus</td>
<td>AA</td>
<td>AV</td>
<td>AA</td>
</tr>
<tr>
<td>ADL</td>
<td>BA</td>
<td>BA</td>
<td>AV</td>
</tr>
</tbody>
</table>

Figure 3: Summary of Support Characteristic Evaluation.

**KEY**

<table>
<thead>
<tr>
<th>AA</th>
<th>AV</th>
<th>BA</th>
<th>DE</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above Average Rating</td>
<td>Average Rating</td>
<td>Below Average Rating</td>
<td>Prog. Lang. DEpendent PDL</td>
<td>Prog. Lang. INdependent PDL</td>
</tr>
</tbody>
</table>

35
procedure FIND(SearchPat: in string;  --| find patterns in text
          --| Regular expression to search for)
--| Modifies: StdOutput
--| N/A: Errors, Raises
is
 --| Overview
 --| Find is a program used to scan standard input, searching for
 --| a pattern. The PatMat pattern matching package is used for
 --| pattern matching functions, Streams is used for I/O, and UIM
 --| is used to provide the standard user interface.
 --| Requires
 --| SearchPat must be a valid regular expression pattern.

  type maxLine is range 1 .. 256;

  Lin: string(maxLine);       --| a line from the text file
  Pat: pattern;              --| a pattern
  M: string(maxLine);        --| text which matched Pat in Lin

 --| Algorithm

  --| If SearchPat is empty then explain usage and exit
  --| Convert search string into a pattern
  --| For each line in stdinput
  --| Display the line if it matches the pattern
  --| Exit

begin
  null;
end Find;

Figure 7A: BYRON High-Level Design Description to search for a pattern in text.
procedure FIND(  -- find patterns in text
   SearchPat: in string  -- Regular expression to search for)
   --| Modifies: StdOutput
   --| N/A: Errors, Raises
is
   --| Overview
   --| Find is a program used to scan standard input, searching for
   --| a pattern. The PatMat pattern matching package is used for
   --| pattern matching functions. Streams is used for I/O, and UIU
   --| is used to provide the standard user interface.
   --| Requires
   --| SearchPat must be a valid regular expression pattern.

   type maxLine is range 1 .. 255;
   Lin: string(maxLine);  -- a line from the text file
   Pat: pattern;  -- a pattern
   N: string(maxLine);  -- text which matched Pat in Lin

begin
   --| If SearchPat is empty then explain usage and exit
   if searchpat = "" then
      PutLine(ErrOutput, "Usage: FIND pattern");
      return;
   endif;
   --| Convert search string into a pattern
   Pat := MakePat(ArgV(1));
   --| For each line in stdin input
   loop
      Lin := GetLine(StdInput);  -- read a line
      if MatchPat(Lin, Pat) then
         PutLine(StdOutput, Lin);  -- Display the line if it matches the pattern
      endif;
   end loop;
   exception
      when StreamEmpty =>  -- Raised when input exhausted
         StreamClose(StdOutput);  --| Close the output file
   end Find;

Figure 7B: BYRON Low-Level Design Description to search for a pattern in text.
PUSH "SOE" (START OF EXPRESSION) ONTO OPERATOR STACK
PROCESS OPERAND
DO WHILE NEXT TOKEN IS AN OPERATOR
   DO WHILE OPERATOR IS NOT SAME AS OPERATOR ON TOP OF OPERATOR /
      STACK AND ITS PRECEDENCE IS LESS THAN OR EQUAL TO /
      PRECEDENCE OF OPERATOR ON THE TOP OF THE OPERATOR STACK
      BUILD TOP NODE
      POP OPERATOR STACK
   ENDDO
   IF NEW OPERATOR IS SAME AS TOP OPERATOR ON OPERATOR STACK
      INCREMENT OPERAND COUNT IN TOP OF OPERATOR STACK BY ONE
   ELSE
      PUSH NEW OPERATOR AND OPERAND COUNT OF 2 ONTO OPERATOR STACK
   ENDDO
PROCESS OPERAND
ENDDO
DO WHILE TOP OF OPERATOR STACK IS NOT "SOE"
   BUILD TOP NODE
   POP OPERATOR STACK
ENDDO
POP OPERATOR STACK
(TOP OF OPERAND STACK CONTAINS TOP NODE IN EXPRESSION)

Figure 8: PDL-Caine Design Description to Process an Expression.

get : module
   interface
      n : parameter(read) Posint,
      pos, buffer : import(write),
      last, store : import(read).
   endinterface

   specification
   pre-get : true
   post-get : pos' = n and
               buffer' = store(n)
   exceptions: if n>last : pos' = last + 1
               | n = 0 : pos' = 1 endif
   endspecification
endmodule get

Figure 9: SLAN-4 Design Specification to get characters into a file.
type index is range 0..1023;
type vector is array (index range <>) of value;

-- [ sort the target array into ascending order ]
procedure singsort (target : in out vector) is
  pivot : value;
  type extent is record bot, top : index; end record;
  active, smaller, larger : extent;
  package ext_stk is new stack_facility (extent);
  use ext_stk;
  unsorted : stack;
begin
  -- initialize the unsorted stack to contain the entire extent of
  -- the target array;
  while
    is_not_empty (unsorted)
  loop
    get_top (unsorted, active);
    -- [ repeatedly whittle the active extent down to an appropriate
    -- size, segregating elements between pairs of subextents and
    -- saving the smaller subextents on the unsorted stack ]
    while
      condition
      -- the active extent contains more than 10 elements
      -- or the active extent starts with the first
      -- element of the target array
      loop
        -- [ sort three elements in the active extent of the target array
        -- (namely, the two boundary elements and the midpoint element)
        -- into ascending order, making the middleman the pivot value ]
        boundary_and_midpoint_sort (target, active, pivot);
        -- [ segregate the elements in the active extent of the target
        -- array according to whether they are greater than or less
        -- than the pivot value, and divide the active extent into a
        -- pair of nonoverlapping contiguous subextents, one smaller
        -- and one larger in size ]
        segregate_and_divide (target, active, pivot, smaller, larger);
        put_top (unsorted, larger);
        active := larger;
        end loop;
    end loop; -- siftsort the active extent of the target array;
    end loop;
end singsort;

Figure 10: PDL/Ada Design Specification to sort a vector.
PACKAGE Bank_Account
IS
PROCEDURE Open_Account(Id : OUT Integer)
PROCEDURE Make_Deposit(Id : IN Integer; Money : IN Money_Type)
PROCEDURE Make_Withdrawal(Id : IN Integer; Money : OUT Money_Type)
PROCEDURE Close_Account(Id : IN Integer; Money : OUT Money_Type)

Account_Closed : EXCEPTION
Overdrawn : EXCEPTION

END Bank_Account

PACKAGE BODY Bank_Account
IS

--the following type and object definitions describe the variables

TYPE Account_Info_Type
IS RECORD
  Name : String(1..30) -- name of account
  Address : String(1..50) -- mailing address
  Account_Number : String(1..8) -- character format of account number
  Current_Balance : Money_Type -- running total of funds in the account
END RECORD

TYPE Customers IS ARRAY (1..1_000_000) OF Account_Info_Type

TRW_Credit_Union_Sheaf_Draft_Info : Customers

PROCEDURE Open_Account (Id : OUT Integer)
IS SEPARATE
PROCEDURE Make_Deposit (Id : IN Integer; Money : IN Money_Type)
IS SEPARATE
PROCEDURE Make_Withdrawal (Id : IN Integer; Money : OUT Money_Type)
IS SEPARATE
PROCEDURE Close_Account (Id : IN Integer; Money : OUT Money_Type)
IS SEPARATE
END Bank_Account

40
PROCEDURE Open_Account (Id : OUT Integer)
IS
BEGIN
get new Id number -- the next empty slot in the array
initialize all fields of the new account (Id) record
END Open_Account

PROCEDURE Make_Deposit(Id : IN Integer; Money : IN Money_Type)
BEGIN
add Money to TRW_Credit_Union_Share_Draft_Info(Id).Current_Balance
END Make_Deposit

PROCEDURE Close_Account(Id IN Integer; Money Out Money_Type)
IS
BEGIN
indicate that the account is closed to future transactions
IF TRW_Credit_Union_Share_Draft_Info (Id).Current_Balance > 0.00
THEN return Money to the customer
ENDIF
END Close_Account

PROCEDURE Daily_Transaction
IS
    Cash_In_Hand : Money_Type
    My_Id : CONSTANT Integer is top secret
BEGIN
Make_Deposit (My_Id, Paycheck_Amount)
Make_Withdrawal (My_Id, 2 * Paycheck_Amount)

EXCEPTION
WHEN Account_Closed => print "account closed" on teller screen
WHEN Overdrawn => print "nice try, but you don't have enough funds to cover the withdrawal"
END Daily_Transaction

Figure 11: Ada-PDL Design Specification for Bank Account Application.
function Binary_Search_Design(K: in Key; T: in Table) return Integer is
{Left Boundary}; -- Integer := T’First - 1;
{Right Boundary}; -- Integer := T’Last + 1;
{Mid Point}; -- Integer;
begin
{Initialize Mid Point and Left and Right Interval Boundaries};
while {Interval is non empty} loop
if K = {The Key at the {mid point} in Table (T)} then
return {mid point};
elseif K > {The Key at the {midpoint} in Table (T)} then
{Search the Right Half Interval};
else
{Search the Left Half Interval};
end if ;
{Compute new mid point};
end loop;
return {A value to indicate Key (K) was not in Table (T)};
end Binary_Search_Design;

Figure 12: PDL-Arcturus Design Specification for Binary Search.

package STACK is
-- specification

procedure PUSH (x: INTEGER)
function POP return INTEGER;
end STACK;

package body STACK is
-- body

MAX: constant := 100
S: array (1..MAX) of INTEGER;
TOP: INTEGER range 0..MAX := 0;

procedure PUSH (X: INTEGER) is
begin
TOP := TOP + 1;
S (TOP) := X;
end PUSH;

function POP return INTEGER is
begin
TOP := TOP - 1;
return S (TOP + 1);
end POP;
end STACK;

Figure 13A: ADL Design Specification for Stack.
procedure EMVPRC -- process Element Msg Values

    (GP_ELSET_COUNT : in out INTEGER) is

    ARG1, ARG2, ARG3, ARG4, ARG5, ARG6 : TYPE_TBD;

procedure EMVADD
    (ARG1 : in TYPE_TBD;
     ARG2 : out TYPE_TBD) is
begin
    null;
end EMVADD;

procedure EMVUPD
    (ARG3 : in TYPE_TBD;
     ARG4 : out TYPE_TBD) is
begin
    null;
end EMVUPD;

procedure EMVDEL
    (ARG5 : in TYPE_TBD;
     ARG6 : out TYPE_TBD) is
begin
    null;
end EMVDEL;

begin
    -- select the case which applies
    -- case 1 : Add GP element set
    EMVADD (ARG1, ARG2);
    -- case 2 : Update GP element set
    EMVUPD (ARG3, ARG4);
    -- case 3 : Delete GP element set
    EMVDEL (ARG5, ARG6);
    -- end case;
    -- GP_ELSET_COUNT = GP_ELSET_COUNT - 1

end EMVPRC;

Figure 13B: ADL Design Specification for Maintaining Message Processing.
REFERENCES


