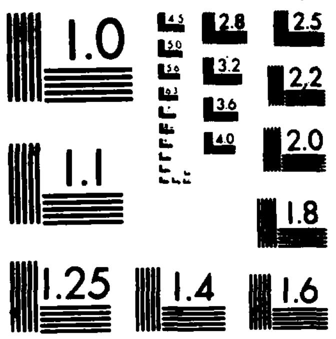
AD-A182 918   RAPID PROTOTYPING SYSTEM(U) INTERNATIONAL SOFTWARE SYSTEMS INC AUSTIN TX T WELCH 15 DEC 86 TR-T101/85/0640-5 N00014-85-C-0640   1/1

UNCLASSIFIED   F/G 12/5   NL


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

Final Scientific Report

Rapid Prototyping System

Office of Naval Research

Contract No. N00014-85-C-0640

December 1986

International Software Systems Inc.
12710 Research Blvd., Suite 301
Austin, Texas 78759

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		**READ INSTRUCTIONS BEFORE COMPLETING FORM**
1. REPORT NUMBER TR-T101/85/0640-5	2. GOVT ACCESSION NO. ADA182918	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Final Scientific Report: Rapid Prototyping System		5. TYPE OF REPORT & PERIOD COVERED Final 6/15/85 - 3/31/86
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Terry Welch, et al.		8. CONTRACT OR GRANT NUMBER(s) N00014-85-C-0640
9. PERFORMING ORGANIZATION NAME AND ADDRESS International Software Systems, Inc. 12710 Research Blvd., Suite 301 Austin, Texas 78759		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Procuring Contracting Officer, Office of Naval Research, Dept. of the Navy 800 N. Quincy St. Arlington, VA 22217-5000		12. REPORT DATE December 15, 1986
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Defense Contract Administration Services Management Area-San Antonio 615 E. Houston St., P. O. Box 1040 San Antonio, TX 78294-1040		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
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) Rapid prototyping, Reusability		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Rapid prototyping is pursued as a means to develop and validate functional specifications prior to extensive code development in a large system. This report details and analyzes a strategy for rapid prototyping emphasizing these capabilities (1) a specification-level language, PSDL, which minimizes design decisions, (2) a library of reusable software modules, and a database system to aid in module retrieval and and analysis, and (3) a set of interactive tools which aid in the construction and execution of prototyping experiments.		

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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Rapid Prototyping System

Table of Contents

Rapid Prototyping System

Executive Summary

Rapid prototyping is pursued as a means to develop and validate functional specifications prior to extensive code development in a large software system. To achieve a functional prototype execution with moderate development effort three capabilities are specified: 1) a specification-level language, PSDL, which minimizes the extent of the design decisions which must be provided by the prototype developer; 2) a library of reusable software modules, and a database system to aid in module retrieval and analysis; and 3) a set of interactive tools, including a system interpreter for debugging purposes, which aid in the construction and execution of prototyping experiments.

This report analyzes the functionality needed within each of these capabilities. It provides a design for the prototyping language and its related support tools. It describes a strategy of component reuse which deals with generic components. We provide an approach to understanding reusable modules within the context of an intended application by means of a visual interpreter which animates module behavior. We describe designs for system tools which help users cope with complex system design information by means of graphical presentations.

The prototyping language uses a dataflow style of presentation to express precedence relationships between operations. It supports a hierarchical and object-oriented structuring which is similar to that which is appropriate for Ada. It supports abstract datatype concepts mixed with conventional Ada data typing, but shields the user from data space management concerns.

The prototyping support environment is centered around an object-oriented data management system which is closely coupled to a workstation display. This structure is effective for storing and viewing hierarchical compositions of program graphs, data type definitions, and other design data.

We conclude that the specifications for a prototyping system developed in this report provide an effective basis for developing such a system.

Final Scientific Report

Rapid Prototyping System

1. Introduction

This report covers work done on a joint project with the Honeywell Computer Sciences Center on RaPIER (Rapid Prototyping to Investigate End-user Requirements). ISSI was responsible for the technology used to construct prototype software, with Honeywell performing the remaining system work, including end-user interface, library of reusable modules, and methodology.

This effort was part of a multi-year project to design and build such a system. The initial contract, reported here, concerned design of the prototype system design language and software base management system.

This report describes the design of the key components of a prototyping system. As a design document, its function is to give a detailed discussion of implementation objectives and to decompose the overall system problem into a system of subproblems which are more tractable.

1.1 Objectives and Approach

In order to reduce the total cost of developing complex software systems, it is often desirable to evaluate and validate proposed system functionality prior to extensive system design and coding. This is to ensure that proposed system specifications are appropriate to end-user needs, and that the user interface is properly designed. In addition, prototyping will often be used to help users better understand their own needs, in the case where the system involves unfamiliar technology and objectives.

The problem today is that prototype construction requires almost as much effort as building a final system. For prototyping to become widely effective it must become truly rapid. This includes not only initial development of a system but also subsequent modifications as the specified functionality is adjusted.

This very simple problem statement triggers many possible solution paths, however, and probably no one technique or approach will provide an effective solution. Rather, an integrated set of tools and techniques will be needed to reduce prototyping effort to an acceptable level.

Our approach is to rely very heavily on reuse of prior designs, as encapsulated in reusable software modules. We then hope to provide significantly improved ease of construction for prototype systems with innovation in three major areas: system specification, support for reuse, and prototyping tools.

1.1.1 System Specification Techniques.

We seek to minimize the effort needed to describe the proposed system functionality in executable semantics. To this end, we propose and develop a Prototype System Description Language, PSDL. PSDL is a non-procedural specification-level language which expresses functionality with a minimum of control and data management decisions needed from the prototyper. It should be viewed as an interconnection language rather than a programming language because its principal function is to interconnect reusable modules.

For an objective of improving readability of system descriptions, PSDL is given a graphical syntax. To enable it to deal with real-time and concurrent systems, it is based on a dataflow style of description. This is a particularly challenging combination of objectives because no prior widely-used language has achieved such a combination of attributes, and in particular, an executable semantics for a function-level dataflow language does not exist (in a form that we find simple enough to be effective).

1.1.2 Support of Reuse

We seek to provide the prototyper with effective access to previously debugged specifications/programs. These may be used directly as subsystems, be modified to adjust for new data types and new functions, or serve as templates for new designs. By their character, modules which are reusable are generic in character, meaning that they are easy to customize via parameters to a variety of applications or, even better, that they appear self-adaptive to input data types.

To help the prototyper achieve reuse, the prototyping system must give support for 1) finding a useful module in a large collection of candidates, 2) understanding the functionality of candidate modules in the context of the target system, and 3) customizing the module and inserting it into the current design. This is viewed principally as a database problem in helping the prototyper deal with the large quantity of information which will be helpful in distinguishing and understanding modules (e.g. specification text, PDL descriptions, test cases, prior usage examples, etc.). We believe that innovative database capabilities are needed to organize this information so that browsing by navigation through multidimensional relationships is convenient.

1.1.3 Prototyping Tools

Prototype development differs from normal programming in that it is inherently a trial-and-error activity. Further, it should be an activity involving system analysts, not programmers.

We seek to develop an integrated set of tools which support this activity directly. We assume the use of current workstation technology which provides high-bandwidth graphical displays and fast-response interactivity. The challenge is to harness this technology to provide the ease-of-use which is critical to prototyping system effectiveness.

1.2 Deliverables

No technical deliverables per se were specified for this contract, beyond periodic reports. The substantial deliverables were scheduled for later months in the effort (subsequently canceled). There were four tasks scheduled for this initial contract, being the specification of 1) the prototyping language, 2) the supporting database system, 3) an Ada implementation, and 4) reusable components. This report constitutes the finished deliverable on all four tasks. The following discussion summarizes the results.

1.2.1 Prototyping Language

A Prototype System Description Language, PSDL, is proposed here. It is described in terms of its 1) control structures, which are very simple,

2) data structures, taken from Ada, and 3) function definitions, which are based on the available set of reusable modules. (Section 3.)

1.2.2 Database

A database system for support of reusability is called the Software Base Management System, SBMS. It is described here in terms of its data model, access operations, control system, and tool support (Section 4).

1.2.3 Mapping to Ada

The result of a prototype specification will be Ada source code which can be compiled to produce an executing prototype. This mapping is straightforward because the PSDL structure has been developed to fit within the packaging concepts of Ada, and because Ada data typing is used directly (Section 3.4). The principal problem here is that Ada data typing is not always natural for functional specifications.

1.2.4 Reusable Module Specification

The modules used in prototype execution must meet certain interface standards, to permit rapid incorporation (Section 3.5). A standard procedure-call interface is used, but the module must adhere to certain parameter standards. Additionally, modules should be reentrant, with external storage of state information.

1.2.5 User Interface

Not specified in the original contract, but subsequently determined to be an important topic, was design of the human interface needed for prototyping tools. These requirements are discussed here at some length (Section 5).

1.2.6 Conclusions

The overall result of this work, and of related follow-on work, is that the approach taken here is a viable one. Continued development of this systemis expected, as funds can be obtained.

2. Components of Prototyping Environment

Any design environment must have three fundamental tools: 1) A language to describe designs, 2) a database to hold design objects, and 3) a human interface through which the designer can create and view design objects. These tools should be fully integrated and be available across all phases of the software life cycle, to minimize designer training time and to minimize reentry of data.

2.1 Language

We can anticipate a specification language which is: 1) executable, in the sense that it can be translated into code which gives correct input/output functionality; 2) simple, in that it can express system functionality much more concisely than (say) Ada does; and 3) hierarchical, in that it explicitly supports the composition of subfunctions into functions, and of data elements into more complex elements.

Such a language will achieve simplicity by providing three types of abstraction: 1) Control abstraction, using data flow to express precedence relationships and otherwise assuming that all activities are concurrent tasks; 2) Data abstraction, using the methods of data management to express logical data relationships without physical encodings; 3) Function abstraction, whereby input-output relationships are expressed through decomposition and basic mathematical operations.

2.2 Database

A database system must not only store design objects, namely fragments of specifications, designs, and code, but must store the relationships between those objects. The database is the principal integration element carrying information across the phases of the software life cycle. All of the facilities provided must equally well support the data objects that are used at different stages in software development, namely specifications, designs, and code.

a) Composite Objects. Software objects will almost always be made up of components or defined by the interaction of components. Each object must be linked to its components and must state, explicitly or implicitly, the relationships between code modules. Further, objects can have links to other objects as attributes, as when a block of code has a delay model associated with it. We presume that a database system will provide the ability to extract views of these composite structures, as when showing the decomposition of a performance specification down into component delay values, while not displaying unrelated component information.

b) Reusable Objects. Existing objects, including those from prior projects, must be easily accessible through a classification system. These classifications would categorize objects by diverse criteria such as subject area, optimization criteria, or by naming conventions. Users will find reusable objects by browsing through sets of objects identified by logical combinations of categories.

c) Configurations. All objects will be subject to continuing revision. This requires a version control system to maintain multiple versions of each object. It also requires configuration management so that overall systems can be configured from selected versions of component objects.

d) Test Cases. All objects in the database may have an associated set of input values which stress the behavior of that object. for specifications these might be called scenarios. Each test case would have a description of acceptable output values. These are used both to test object implementations and to illustrate object behavior (for human understanding).

e) Data Dictionary. Data types described as part of a specification will be used in all later phases of software development. Types will be composite objects, typically defined in terms of other types. Types look like other objects in the database, but are distinguished here because of their very extensive reuse throughout the development process.

2.3 Human Interface

To aid the human in dealing with complex information, tools for viewing and editing system data are of course needed. A strong emphasis in the prototyping systems is on graphical descriptions which display interactions of components, to augment textual and symbolic descriptions.

a) Editing. A syntax-driven editor is assumed to be available for each language in the system. It would support graphical as well as textual syntaxes. This editing ability would also be available for creating and revising data objects in the data base.

b) Browsing. A viewing mechanism for objects in the database will also serve as the general-use data projection mechanism of the system. It will provide the ability to look at complex objects by "panning" and "zooming", as those terms are interpreted for multidimensional images. For browsing and other purposes, the system can assemble a "view" as the user develops an understanding of what information is needed; a view indicates what data to project and what object relationships to be followed in looking for related objects.

c) Interpretation. A visual feedback mechanism in the execution of descriptions will provide a natural debugging support for the design representations. It will also aid in understanding the dynamics of reusable components and their function in a particular context.

3. Language Aspects

The Prototyping System Description Language (PSDL) is the most visible part of the prototyping system, but is also the most difficult because of conceptual developments needed to make it effective.

3.1 Requirements

a) Specification-Level Language. We assume that the prototyping done in RaPIER will be for functional specification validation not for design evaluation. The system description developed in PSDL will correctly reproduce the functionality (outputs as a function of inputs) of the system, but will not necessary map into any later implementation. That is, the internal components used in a PSDL description will not always directly correspond to components of a final design.

While PSDL may be extended later to be a design language, that is not the objective of the present RaPIER effort. This priority manifests itself in the description of system timing. The enforcement of software delay constraints is a design issue relating to the partitioning of code and speed of processors used, and so is not part of a functional level prototyping effort. PSDL will provide the ability to describe timing constraints, but they will not affect prototype execution, except 1) when programmed actions depend on delay values, and 2) when prototype execution speed is adjusted manually to meet design constraints, by means of actions outside PSDL semantics (e.g. faster machine, optimized compiling, etc.).

The goal of doing functional-level prototyping, rather than design prototyping, is to 1) achieve the lowest complexity in system description via abstraction, and 2) to permit prototype results to be applicable to all possible later embodiments.

b) Prototyping. The use of PSDL for functional prototyping imposes a certain focus on it. Prototyping requires an efficient interface to a code generation system, so that the executable code is not dreadfully slow. This rules out the use of the type of specification language which is often used for formal verification, namely one which states invariant properties of computation. For example, the Sort function can be specified easily by giving the ordering relationship on the sorted objects (it's outputs), but generating a program from that relationship can yield a slow implementation.

As a prototyping language, PSDL makes no attempt at this time to express alternate views (e.g. dataflow versus control flow) of a design, as is found in some non-executable specification languages. Rather, it is intended to focus entirely on the data relationships implemented by the prototype.

c) Reuse. The RaPIER prototyping approach stresses reuse of existing software modules as a strategy, rather than generation of new code for each prototype. This implies a focusing of development effort on interfacing software modules with each other, rather than building code generation facilities.

Reuse also requires dealing efficiently with the kinds of code that lend themselves to reuse, e.g. generic procedures and object-oriented structures. The object-oriented style is useful because it packages together all functions dealing with a single data structure, so that the data is hidden from the system specification, and need not be expressed in PSDL. However, this stresses the need in PSDL to deal both with those functions as entities in a process and with the collection of functions as an object; the linkage between them requires special attention.

An implication of the reuse orientation is that a PSDL description may not be understandable to a human who does not understand the operation of the reusable software modules employed. While annotation text for abstracting the meaning of modules may be provided, it is not the intent of PSDL to provide comprehensive functional descriptions.

d) Data. PSDL best fits in Ada environments, because Ada is a suitable language for code reuse. Therefore it is desirable that the data types in PSDL be similar to Ada in style.

e) Incompletely-defined Systems. Prototyping will frequently be carried out on partial specifications. It is desirable for PSDL to work as effectively as possible in these circumstances, by use of suitable defaults. This requires facilities for hierarchical definition of data and function, plus good support for progressive redefinition of these objects.

3.2 Semantics

PSDL uses a dataflow approach to description, as do many other specification languages. This permits minimal expression of operation sequencing, with only data precedence relationships shown. This form provides for expressing the most possible concurrency in execution. Concurrency per se is not an objective of PSDL, but the availability of concurrency is a mark of least binding to any particular implementation, as is appropriate for a functional specification.

The system being described is viewed as a large black box with specified functions relating inputs to outputs, and with definitions of I/O data types. The functional description is decomposed into a composite of interacting black boxes, each defined in the same style as the original, with interconnections defined as direct linkages of outputs to inputs. The following sections describe these basic elements: the data types, the function descriptions, and the control/interconnections.

3.2.1 Data Descriptions

Data will be described in PSDL using Ada type and subtype facilities, e.g. using range, array, record, case, etc. This data model is too closely tied to physical representations to provide high-level abstractions of data, but it is sufficient and convenient for code generation capability. In later versions of PSDL, extensions to the Ada facilities will be provided to achieve somewhat greater abstraction. Virtual variables will be added, which are essentially procedure calls to calculate implied results from other data in the type. Substitution of generalized objects (abstract data types) for explicit data types will be provided.

It will be frequent that two functions to be linked together will not have matching data types. If function A feeds function B, with A producing an array of integers and B accepting simple integers, then PSDL will perform type conversion by executing B multiple times over the values in A's array. This simple type conversion will be provided initially as explicit functions, with built-in abilities provided in later versions.

The semantics of variable name usage will be "single assignment". That is, every variable instance is written only once, and never modified (conceptually). Thus the execution of the PSDL description is interpreted to mean that function outputs are streams of results with each output value being a different instance of the output variable. Of course few implementations will do it that way, but it is the responsibility of the implementer to not let space-saving data assignments change the result of the computations.

Support for the prototyping of partially defined specifications will be provided through default values. Any operation on an undefined input will cause an undefined output. The undefined type state will carry a code value which is propagated unchanged though each function so that undefined system outputs can be traced by to their cause.

3.2.2 Function Descriptions

The contents of each "black box" can de defined in several ways:

1. As Ada code, either a reusable module or coded directly through RaPIER;
2. By PSDL composition, where the function is defined by the interaction of smaller functions;
3. By PSDL primitives which define outputs in terms of inputs by a variety of mathematical notations.

An early version of PSDL will rely heavily on Ada and composition, with the exception that PSDL will provide direct substitution operators (where the output data type is a rearrangement of input data). Later versions of PSDL will be principally distinguished by increased capabilities in PSDL primitives.

Support for partially defined specifications will be done with library software, not PSDL semantics. A default module will be provided which generates a random output value, from a specified range, for each input value. Another software module will substitute human activity for a function by going to a terminal for the function output. Many variations on these default strategies can be envisioned, and made available to the prototype as reusable functions.

3.2.3 Control Descriptions

The activities carried out in a directed graph of functions are interpreted to be completely concurrent except as limited by data availability. The output of each computation is a message which is copied into a separate input queue for each receiving function. Functions are activated by messages in their input queue(s). Of course real implementations will seldom execute that way, but they must produce the same computational results as that conceptual model.

To make the described system executable, a small amount of explicit control information is required from the programmer/prototyper. This occurs in the form of indicating which input data triggers execution of a function. Thus a function receives input data in two forms:

Activation data: a message which causes execution of the function.

Reference data: a message which is available for use by the function

A critical distinction between these two occurs when a sequence of messages passes over the same path to a function. If this sequence occurs on an activation path, it causes multiple executions of the receiving function. If it occurs on a reference path, only the latest message is seen by the receiving function.

All other aspects of control, namely of determining the sequencing of operations, is determined by precedence relationships between functions. The semantics of control are prescribed by a serializing execution model. That is, a serial execution of the functions is determined by the rules of an execution model, and any other implementation must produce equivalent functional results. The three rules are:

1. Precedence Rule: If function A has output which is an input to function B, then A executes before B (this is a trivial rule).

2. Pipeline Rule: If function A produces multiple outputs on the same activation arc, then all down-stream activities caused by the first activation are completed before the second activation commences.

3. Synchronization Rule: If function A has several inputs, then A should not execute until all possible input data is available (i.e., all upstream activities that are going to execute have executed).

In case of conflict, the lower number rule overrides a higher numbered rule. In the case of cycles, all cycles are broken at the point of feedback for the sake of determining function priority. In the case where the relative sequencing of two functions is not determined by the above rules, a random sequencing is chosen.

Timing is not a semantic issue in PSDL, as it is defined in the RaPIER version. For user convenience, a notation can be attached to specify delay values for responses times between inputs and outputs, but these can have no interpretation in a prototype of the specification (where time is compromised). Later extensions to PSDL are expected to provide a better treatment of timing.

3.3 PSDL Syntax

This syntax discussion has three major parts: data representation, operator representation, and system structure. The overall system being defined is viewed as one big object (operator) having inputs and outputs with specific data representations. That object is described internally as a PSDL composition, namely saying its function is defined by the interaction of some set of components. Each component is an object of the same form as the original. Thus, the descriptions of the system inputs and outputs, by data type, and of the operator function in terms of the interconnect of other operators, serve to define a PSDL system. Each of these is described here in both a graphical syntax and a database representation (textual syntax).

3.3.1 Data Descriptions

Ada data types are used exclusively in the initial version of PSDL. If there is a difference between Ada and PSDL usage, it will be that PSDL more heavily uses "array" and "case" type structures, so these need to be easy to express.

3.3.1.1 Textual Data Description

The Ada syntax will be used directly as the basic syntax. Extensions will probably be added later to facilitate the use of abstract data types.

The database representation of data has two parts, type descriptions and instance values. An instance value is just a bit string which is interpreted according to its associated type description. Initially there will be no usage of composition on instance data, where several instances are interpreted as concatenated to form another instance; but that extension will be an early performance enhancement. The type descriptions are composed in a straightforward way from type objects, each of which represents one level of record structure and array. Thus a complex data type will be described by a linked tree of objects even though its instance data is a single contiguous object.

3.3.1.2 Graphical Data Description

The exact graphical format for presenting data will be the subject of experimentation, within the guidelines of the following strategies. The structure will be tree shaped, with each node being a _record_ construct. Each such record construct will be assumed to define an _array_ (repeating group) of such records unless explicitly noted as having a unique value. Case structures are shown as an extension of the record structure, yielding a choice among record types for each instance. Fields in a record are denoted by horizontal separation while options in a case are denoted by vertical separation. Care will be taken to achieve good visual separation between element names and element type declarations, because users frequently are looking at names only or at types only in using or understanding an overall data structure. An alternative layout to be investigated is a vertical sequence of components, with indentation to show groupings.

3.3.2 Operator Descriptions

A PSDL composition operator is defined to be the interconnection of component operators, also called a graph. The basic job of PSDL descriptions is to describe what outputs connect to what inputs. Added to that basic function structure are four topics of special interest: 1) Data relationships for type checking, 2) activation information for each operation, 3) interfaces to object-oriented software modules, and 4) default values for undefined inputs.

3.3.2.1 Textual Operator Description

Our fundamental approach is to represent operators and their interconnections non-procedurally, in very much the same way that they will appear in an object-oriented database. Each operator has two interconnected parts: Interface and Body.

An Interface object includes these attributes:
Input ports, and their data types
Output ports, and their data types
Actual parameter to formal parameter name mappings.
Presentation information (operator icon)

Each port can be either an activation port or a reference port.
There are no bidirectional (in-out) ports.

A PSDL Body includes these attributes:
Component List, with port names,
Interconnection list (arcs), with set of connected port names,
Presentation information (graphical positions)

This component list gives the name of each internal operator, and specifies where the data for each operator input is obtained (actual parameters). Each data source is some output of a component in that same list. For completeness, the first component in the Component List is always the interface component, which mirrors the input-output ports inherited from the interface of the object being described. (Since this "interface component" looks at the surrounding interface from the inside out, its directions are reversed from the ports in the interface object; its inputs are the output ports, and its outputs are the input ports). Note that component names are just pointers to the Interface objects of the component operators.

Of course there are other syntaxes for describing a directed graph, often achieving more clarity of presentation by differing degrees of repeating some of the information. Such alternate forms may be appropriate for PSDL, depending on implementation objects in later versions.

3.3.2.2. Graphical Operator Descriptions

The graphical syntax for operator icons and graphs is the least stable aspect of the PSDL system definition. This is because the graphical syntax is easily changed and because future experiments with human usage are likely to cause continued evolution in the syntax.

The basic syntactic elements are:

a) Function. An operator with no internal state is represented by a circle. The name of the function appears below the circle. The name of the body (implementation) appears inside the circle. Some common functions, such as simple conditionals, may have distinctive icons other than a circle.

b) Procedure. An operator which maintains internal state is represented by an icon which is part circle and part box. Its labels are the same as those for a function. The term operator which includes both functions and procedures is represented by a "bubble", which refers to all of the above icons.

c) Port. An operator has input and output ports by which arcs connect to bubbles. Ports can have names outside the bubble (actual parameters) and inside the bubble (formal parameter, in the body of the operator). Each type of port can have a separate icon. An activation port is represented by an arrowhead with a single shaft, while a reference port is shown as a double-shafted arrow.

d) Arc. The connection from some output port to some input port represents data flow between those ports. A net (arc) connects one or more output ports to one or more input ports, with a single data type throughout. The data type associated with an arc is optionally displayed as commentary on the net.

e) Persistent Data. A procedure can contain state information which is retained between activations of the procedure. This is shown explicitly as a box embedded in a net. If a declared data element is initialized from an external port then it is shared global data; else it is static data.

Object-Oriented Modules. An O-O module typically has multiple activation inputs. It can be represented directly that way in PSDL without semantic loss, but such a representation does not give a clear picture of system operation. Preferably, the various methods are split out as separate operators, sharing common state information on a reference basis. In that case, the state information need not be typed, because its structure is hidden within the object; it can be given a data type name which is not further defined.

The operator names are given by "object_name.method" so that the object name is preserved in each operator, but even that is not essential for correct functioning. The key understanding here is that when such a module is found in SBMS it provides several operators at once rather than just one; those operators can be used (or ignored) in whatever combination makes sense to the program.

Graphical Operator Icons. Operators will be shown as ovals with inputs as arrows to the left for activation inputs and on top for reference inputs. Outputs will leave from the right and bottom. Data types will be shown in square boxes along the arcs connecting ovals. Reference inputs will be shown with short double arrows (two parallel shafts) and a name of the referenced data (often a single letter will be used, like when long wires are broken on an electrical schematic.). Activation inputs will have arrows continuous from their sources.

Built-in data mapping functions will often have special icons to graphically portray their function, particularly for merge and case. Other mapping functions will have very small icons, so they will not interfere with visualizing system operation. This is an area in which continuous refinement can be expected, so exact icon definitions will not be attempted at this time.

3.4 Mapping to Ada

The structure of PSDL was developed to permit easy translation into Ada. The strategy of this mapping is seen in the three dimensions of function, data, and control.

3.4.1 Function Mapping

The library modules from which PSDL prototypes are composed will be provided in Ada. They frequently will be Ada packages, but could be procedures. In certain kernal modules, the reusable component will not be source code but will be a generator which creates source code. Any user-produced Ada procedure will be includable if certain mechanical interface standards are followed (section 3.5).

3.4.2 Data Mapping

PSDL is defined in terms of Ada data types, so mapping of data can occur directly. The stress on this approach is that Ada data typing is very laborious for some useful constructs at the functional level. These involve generic functions which are reusable modules intended for wide use. An example is a module which generates (or interprets) commands taken for a list of commands having a variable number and type of parameters. If the command list is provided at compile time, the data is well defined. However, the generic parameter list is not easy to describe. This type of problem may restrict the extent to which type checking can be enforced at the time the PSDL graph is constructed, so errors could be detected only at the time the PSDL function is translated into Ada.

3.4.3 Control Mapping

Each level of PSDL description, namely a single graph, would be mapped into an Ada package. For simple graphs the package contents would be just a set of procedure calls to contained components. In the case where a function can produce several outputs (activations or messages) for one input, then a dynamic scheduler is needed within the package. A simple queuing mechanism will probably do that. This dynamic scheduling marks a difference between an object-oriented language and a standard procedural language.

3.5 Specification of Reusable Modules

At this point we know of no particular constraints on reusable modules to fit in PSDL prototypes other than that they must have a procedure call interface with certain parameter conventions. For Ada translation, the modules must be Ada source code. For interpretation, compatible object code from any language will do.

A database entry for the modules must be prepared, giving its interface information. The one quirk with PSDL is that a module with multiple entry points is represented by a separate logical function for each entry point, so that the explicit set of outputs for each entry point can be displayed in the PSDL graph.

The interface specifications for code modules are established to provide for interchangeability of modules. These specifications are implementation specific (not dictated by PSDL) and may be changed. First, all parameters are called explicitly (no globals) and are passed by name, in the simplest implementation strategy. Second, each output message is conveyed by an explicit procedure call from inside the module to a message handler, so that multiple output messages can be accommodated. If procedures exist that do not have these properties, they are usually easily fixed by putting the preexisting code inside another procedure which serves only to adjust the interface formats.

In summary, the specifications on reusable modules are not very confining, and a wide range of capabilities can be accommodated.

4. Database Aspects

This section provides a logical view of a database system for storing reusable ADA code and other software objects in RaPIER. The intent of RaPIER is to exploit effective reuse of existing software modules to expedite the construction of prototypes. This emphasis on reuse results in the need for an effective database management system.

The primary function of the database management system, called herein Software Base Management System (SBMS), is to provide support for human interactive selection of software modules. SBMS will also support the design activity itself by storing design objects and composites created through use of the prototyping language PSDL.

4.1 Requirements

The user will participate in three types of activity during a RaPIER prototype design session: Browsing, Probing, and Construction. These activities will be freely inter-mixed, but each one presents particular demands on SBMS functionality. Primarily, the data to be stored in SBMS will be 1) software modules (mainly in ADA and PSDL), 2) data descriptions, 3) classification data, 4) test cases, and 5) current/prior design information.

4.1.1 Browsing

Browsing is the activity of the user looking for something that he could not precisely describe. Much of the effort in browsing involves the human trying to understand the classification system, relative to the needs at hand, and locating a potentially useful component. This is done by retrieving a set of objects in a preliminary query, determining attributes of wanted/unwanted objects in that set, modifying the query, and repeating the process. the user sees the database as a large unstructured set of objects, and only by forming queries on attributes can a workable subset be found.

A classification system is needed by which the general properties of available software modules are described in a standardized vocabulary. Typically classification schemes will be key-word oriented (e.g. math routine, sort routine, device driver, etc.) or will form a hierarchical partitioning (like Dewey decimal system for books). It is expected that multiple such schemes may be tried on SBMS, over time, so the requirement on SBMS is that it effectively support any such classification system.

The query language needed to support browsing will facilitate retrieval of small groups of related documents, either in terms of classification attributes or in terms of relationship to known objects such as data types. It should also be able to utilize an application-specific thesaurus if one is provided with the classification system.

4.1.2 Probing

When a database object has been identified to be of interest, it is examined more closely. This involves seeing its components, its usage constraints, its usage in prior designs, its reaction to test input cases, etc. This probing type of access will likely be the mode used by software tools when they access SBMS data. By tools we mean the browser, editor, PSDL interpreter, and PSDL translator.

In many respects this probing facility is conventional, needing only to retrieve attributes of identified objects. A possible difficulty arises when hierarchical attributes are involved, for the query needs to specify how much information is of interest for each attribute.

4.1.3 Construction

During the process of composing a design, the user will use SBMS as a depository of intermediate results. It is expected that a finished design will feature several levels of composition, with components at each level treated equally in terms of storage and retrieval. It is efficient to store all such components in the SBMS because 1) SBMS provides basic tools for accessing and viewing these objects and their interactions, 2) many components of the design will already be in the SBMS and do not need to be copied, and 3) the new design components should be saved in the SBMS for future reuse or reference.

Interactive use of SBMS during the design process implies the need for part of SBMS to be running on a workstation with sufficient graphical capabilities to present object structures in a meaningful way. Of course, there are various degrees of interactive viewing/editing tools that might be provided with SBMS.

SBMS should also provide a version control mechanism and/or a way to enforce separate private workspaces which share public data.

4.1.4 Data Types

The data to be stored in a prototyping system is rather heterogeneous. Software modules will basically be chunks of text that make up the bodies of Ada packages. They will also have interface descriptions (called "specifications" in Ada), which are separate but interlinked objects that specify inputs, outputs, data types, and other descriptive information common to all bodies which are embodiments of that function.

Other stored data include classification information, which may be either relational or hierarchical, and probably will be a mixture of both. There will also be numeric data stored for testing and demonstration purposes. Each object may also have annotations attached which will be textual. The system metadata, by which other data types are defined, is also stored in the system and manipulated by the standard operators used for other data.

One typical but demanding data type is the description data for program (Ada and PSDL) data values. This type information is frequently referenced and clearly hierarchical, being expressed of layers of record, array, and case composition operators. Retrieval of one complex data type description is a good test of a DBMS, and demonstrates the inherent weakness of the relational model.
With this diversity, a general purpose, object-oriented, data model is needed which will be suitable, in the long run, for the representation of hardware and software system designs.

4.1.5 Implementations

It is anticipated that the functional capabilities of SBMS will increase from phase to phase. Initially, a demonstration system will have enough capability to retrieve objects from a small library. A second phase will have the necessary tools for general-use prototyping. A later version can be anticipated that should provide good support for the process of designing production software. The following discussion of SBMS capabilities indicates specific levels of functionality intended for these phases.

4.2 SBMS Functions

The following strategies of SBMS development give an informal definition of SBMS semantics. A later section discusses the query syntax.

4.2.1 Data System

An object-oriented data model will be used, so SBMS will be seen as managing a pool of objects. An object has an identifier, a type (class), some attributes, some components, and some constraints. The identifier is unique in a flat name space (at least in the early versions). Each object may also have user-provided name attached as an attribute. The type of the object is described by another object (of type "type" or "class"), whose attributes describe the domain information for each attribute of the instance object. Type objects will be arbitrarily constructable, but the initial version will have only a small number of fixed types such as Interface, Ada code body, PSDL body (graph), view, text string, and numeric (test case) data. These are logical types; the implementation will likely have a somewhat different set of physically implemented schemas and objects.

Attributes have values which may be compared, edited, calculated, printed, etc. These values may be complex structures of simple values, as definable by PSDL data definitions of arrays, records, etc. In addition to attributes, an object can have components, which are objects themselves, and whose aggregation makes up the parent object. Some would argue that components are attributes and do not deserve special treatment. The difference between them is mild, but it turns out to be very useful in design databases and so is accentuated: An attribute of X directly describes X, while a component of X is encoded as a pointer which can be followed to obtain indirect descriptive information.

Constraints are of two types, implicit and explicit. Implicit constraints are implied by type information, and their violation prevents the object from being stored in that form in the data base, as being not meaningful. Explicit constraints are user-defined procedures which may be automatically invoked. If X=f(A,B,C) is a constraint, for example, it can be used to supply a value for X or to check the value of X; if an existing value for X does not conform, it is left in place but the object is marked as inconsistent. Implicit constraint enforcement can be supplied in the initial version of SBMS, while explicit constraint management will be supplied to some degree in later versions.

4.2.1.1 Classification

Classification data will be treated as attributes on the object, and each object can have as many classification values attached as needed. Of course an SBMS implementation may physically store classification attributes differently than other object attributes, because they will tend to be used in different access patterns (namely an index structure would be used). In an initial version of SBMS, no particular facilities will be provided in support of classification, other than the ability to attach attributes to objects and retrieve groups of objects accordingly. In later versions tools for building classifications can be provided, namely mechanisms which make it easy to assign objects to categories and improve access efficiency.

4.2.1.2 Queries

The choice of a good query strategy for an object-oriented database is the subject of extensive national research at this time. The SBMS strategy is to provide a basic query facility for the initial version, and be open to further research results before committing later versions. The initial version assumes only two types of usage, namely interactive access during browsing/probing, and single object at a time access by the PSDL translator that generates Ada. In neither of these two cases will complex queries be encountered: rather, complex operations will be accomplished by moving (navigating) incrementally through the possible queries.

Select, project and navigate capabilities, which are discussed below, are to be provided in the initial version.

Select:
The set of objects viewable at one time is determined by a view defined by AND, OR, and NOT combinations of attributes. In addition to these Boolean combinations, the operator "includes" and "is included in" are also provided, to deal with composite data. For example if we wish to find a software module that deals with floating points numbers, we would select modules whose input data types "include" floating point numbers, to find them even when they are mixed in with other data types.

Project:
When an object is retrieved, only part of its attributes are of interest. A "projection" will determine which attributes are needed. The projection description will be managed separately from selection description in SBMS since they will be modified by the user rather independently. It is recognized that the simple binary suppress/expose form of projection is inadequate for later versions of SBMS, but it would suffice for the initial version.

Navigate:
For a given object, the attributes of related objects will be retrieved by implication (or navigation). Related objects are those whose names appear as attributes of the first object. Often an attribute of the initial object will yield a set of related objects, as when a set of components of a composite object are retrieved, so the returned data might be an array of records.

4.2.1.3 Update

The normal update mechanism will be "append", used when adding a new object instance to the database. Changes which involve modifying or deleting existing data must be done within the context of a data control policy (discussed in following sections), which will be application dependent. For initial prototype-level execution of SBMS, the update policies will be manually enforced and subject to change.

Therefore initial update capabilities will be very simple and in two forms. For programs, SBMS provides access to objects in the form of direct reference to the bits of the object, so no constraints are enforced (they are assumed to be built into the application program). For interactive users, an editor allows modification of individual object fields, subject to explicit constraints for that object. Neither form of update requires an update language syntax at this time.

4.2.2 Data Control System

The control of data integrity and access involves three sets of functionality: Write protection, access control, and constraints. Of these, access control is the most complex because it deals both with access concurrency and with privacy. The initial SBMS version assumes only a single user and minimal needs for privacy.

4.2.2.1 Write Protection

Large portions of the data base will be read-only, so enforcement of write protection is important. Every logical object will have separate write protection. A separate privilege, "append-only", will be available for such things as classification information, which may be extended during normal usage. In all versions of SBMS, object protection will be exerted at least by explicit protection bits, or the logical equivalent thereof.

4.2.2.2 Access Control

In the early versions, local workspaces will be provided, each with access to shared database objects. The intent of this is that new objects for a demonstration can be built up on such a workspace, and then deleted as a whole at the end of the session. These workspaces can be retained and used individually for specific demonstrations. Individual objects from a workspace can be moved into the shared area at will. Objects in the shared area are read-only so that existing workspaces are not affected by updates to the shared area.

In a later versions the fact of multiple people working together in development of one prototype causes the need for much better access control. As an end goal, privacy restrictions will be imposed by a View mechanism, whereby a user is provided visibility to a subset of the data objects and is not aware of objects outside that view. A view is expressed like a compound query, and so can distinguish any object from any other object (one viewable, the other not).

Concurrent access is facilitated by versions of objects, whereby older versions are available for public use even as newer versions are being prepared. A "Configuration" is any assemblage of object versions which make up a design. Thus, an over-all version of some prototype design can be in use, even while parts of its are being updated, by forming a configuration of stable versions of its component objects. No provision will be made in early versions for concurrent updates to the same version of an object by concurrent users. Normally, two users would be working on different versions of the object.

4.2.2.3 Consistency

Consistency rules come in two degrees: 1) typing rules, whose violation prevents an object from being stored in the database, and 2) policy rules whose violation causes an object to be marked as inconsistent (needing further processing). The initial version of SBMS will enforce typing rules on data and objects at the level of Ada data types. Later versions will enforce expanded typing rules as appropriate, and will provide a basic mechanism for tracking and maintaining a consistency policy. This will consist of a "level of certification" attribute attached to each object which will help control in appropriate public access to uncertified objects.

4.2.3 Tool Support

The principal utility tools for SBMS will be the browser and editor.

4.2.3.1 Browser

The browser provides a human interface for forming queries and displaying results during a user's search of the database. The initial version of the browser will be adequate but possibly primitive in that it provides only textual output. There is considerable latitude for supplying graphical output, but an appropriate way to do this is undefined at this time.

The browser will be incremental in style. The user will alternate between selection of sets of objects and probing individual objects. Sets of objects are incrementally refined by adding in one attribute condition at a time. Initial implementation will require that the first browsing action will be selection of object class (type) so that all sets of objects will be homogeneous by class. A set of objects selected by an attribute specification will be shown as a list of object names. The user will then probe individual objects to see their particular attribute values. Later implementations may provide the ability to show certain attributes for all members of a set in the initial listing. Probing provides a view of the object's attribute structure as a vertical list of fields, with indentation to highlight internal structures.

4.2.3.2 Editor

The editor provides the means to enter data into specific attributes of SBMS objects. It will be capable of checking type constraints on each field entered. This permits the initialization of SBMS objects and the entry of data for objects which are not directly supported by other tools. The extent to which this editor is merged with other specialized editors, such as for PSDL graphs, is unclear at this time. In late versions of SBMS the editor should become a generic tool, so that it adapts to each object type and edits the object in a manner appropriate to the semantics of that object. This requires self-describing objects, to an extent not immediately anticipated.

4.2.4 Query Syntax

Queries in the initial system can have two forms, interactive and programmed. Browsing and probing are interactive forms which are implemented on top of a programming interface. The browser has no textual query language. Most browser decisions are made by selecting from a dynamic set of options. That is, user indication of object class, attribute field, and object within a constrained list are all done by means of pointing device (e.g. mouse) rather than keying in a name.

The syntax of the program interface is in the normal procedure-call form. The user (program) has a set of procedure calls for these functions (initially): create object, access object, copy object, delete object, find by name, and find by attribute value. Other functions may be added in later implementations, but this basic set is sufficient for initial prototyping efforts of browsing, editing, and translation. Note that objects are referred to in these commands by unique token values which are administered by SBMS. Note also that all metadata on objects is encoded in other objects, so the above commands serve to provide a wide range of DBMS control and query functions.

The syntax forms given here for both the browser and program interface are simple so long as an object has matching logical and physical schemas. An extension of SBMS will allow logical objects to be defined independent of physical objects, which will solve both projection and navigation problems with the existing structure described here.

5. Human Activities in Prototyping

The construction and debugging of a prototype involves substantial human activity, so success of the PSDL system depends in part on how well these prototype developers can carry out their functions.

The expected users of PSDL will be people who are systems analysts, namely people who understand computers but should not be spending their time programming. These people may be occasional users of PSDL, so the learning curve should be short. They will be much more interested in target system functionality than in PSDL representations, so the PSDL expression mechanism must be unobtrusive. The functionality being described will be complex, composed of reasonably complicated reusable modules, so the PSDL/SBMS system must be prepared to support complexity and to help the user manage a complex description. For example, the data objects used in the target prototypes will often involve several levels of type definition, so editing and viewing mechanisms geared to simple fields and records will be inadequate.

The construction of a prototype is inherently an experimental process, where the system is expected to change each time it is run. An initial version is built and executed, showing ultimate users (often different from the analyst who is building the prototype) where the specification is ambiguous or inappropriate. The prototype is then extended or revised for another cycle. If the prototype is well designed, it is possible to try out alternative execution policies in various combinations by straight-forward parameter changes. It is this process of rapid evolution for a software system that we must support.

The human activities in developing a prototype are considered here in four levels: system specification, component reuse, prototype execution, and tool usage.

5.1 System Specification

A principal human interface problem in specification is simply the difficulty of coping with complexity. When the prototyper begins to construct a system, he/she thinks in many dimensions at once. Some aspects of the system are seen in terms of data relationships, some in terms of vague functional specifications, and some in terms of detailed implementation concepts. As the design progresses, and more details get filled in, the prototyper will still feel the need to look at various parts of the system from various points of view. While PSDL is not intended to be a general-purpose specification language which supports multiple descriptions, nevertheless the prototyping system must partially support these needs.

Our approach is to supply strong support for hierarchical structuring of the system description, both for functions and data. There are four aspects of hierarchy which are accentuated.

5.1.1 Abstraction

If properly used, hierarchy is more than just a means to hide detail or control the scope of variables. It provides a means for abstraction, in the sense that every level of decomposition deals with a single set of design issues. To make this effective, we intend each level of the hierarchy to be a free-standing description which can be read and understood with minimal reference to higher and lower levels of the hierarchy. This is accomplished by making all functional relationships explicit at each level and by encouraging the use of names/comments/annotations independently at each level. This is different, for example, from the common use of block-structured languages, e.g. Pascal, where the various levels of composition are only occasionally given clear semantic abstractions.

5.1.2 Mixed Representations

A particular function can be viewed in several ways: its PSDL graph, Ada source code, a PDL description, English textual description, timing model, or input-output data relationships. As a system description matures, all of these may become available for each function. This gives the viewer the opportunity to understand the system functionality better. The important issue here is to be able to freely intermix these descriptions throughout a single design representation, rather than having separate documents for each representation. The problem with this approach is one of controlling clutter, in that too much data will swamp the viewer. Our approach is to utilize the increasing economical technology of interactive graphics to let the viewer switch electronically between representations at various levels of granularity. Initially this will occur by letting the user select repr·sentations for each function individually: the user uses a pointing device to select a function within a graph, and a pop-up menu provides the ability to choose between the available descriptions for that function. In later implementations it will be desirable to experiment with display modes, such as having all the data structure descriptions on a graph be exposed or suppressed as a unit.

5.1.3 Bridging the Hierarchy

All of the design decisions in a software system cannot be adequately encapsulated in a single hierarchical composition. One alternative way is to augment the hierarchy is the use of an object-oriented style whereby certain data operations used throughout the hierarchy are described in a centralized and coherent declaration. These object descriptions permit collective parameterization and replacement of the various methods which make up a class description.

Our approach, initially, is to explicitly support abstract data typing, a key component of object-oriented programming. This support includes the ability to define a data type without specified internal structure and to retrieve appropriate functions for manipulating that type from the SBMS module library. The adverse aspect of using such an abstract data type, which hides its internal state structure, is that generic browser and editor tools must be augmented to deal with each such type.

5.1.4 Configuration Management

A PSDL graph at any one time denotes a configuration of functions which deliver an overall behavior. During the course of prototyping the analyst will alter this graph in multiple ways: add/delete components, modify parameter settings, substitute components, or just revise the documentation (both graph and text). Some of these changes are trial-and-error efforts which should be forgotten, but others have value. In many cases alternative representations of the same function will have value, and need to be alternately invoked. The specification of a particular selection of functions for each experiment is called configuration management.

Our initial approach will be to do the selection of a configuration manually using normal editing tools. This is a statement of belief that the graphical editing tools are convenient enough to be as effective as alternative configuration specification techniques. Making this approach effective, however, requires the ability to name and retrieve prior configurations. At that point it becomes a database issue, and is covered by the SBMS data control discussion.

In later implementations, further tools may be added if our simple initial approach does not prove productive.

5.2 Component Reuse

During the course of developing a prototype, the analyst will frequently call in functions available in SBMS. This reuse of existing function definitions is a critical aspect of making rapid prototyping a reality. As the user manipulates reusable modules, two problem areas are encountered: understanding module behavior and incorporating them into a graph. In general, reusable modules will differ from custom components because they are more generic and so require some customization for each use, which complicates both understanding and incorporation.

5.2.1 Identifying Reusable Modules

Assuming that a reasonable variety of modules will be available, the user has a potentially large problem in determining the appropriate module for a specific application. There are two aspects of this - finding a module and understanding its behavior relative to the target application. Many modules will be somewhat generic and can be adapted to a specific application, but the result of that adaptation may not be obvious.

5.2.1.1 Finding Reusable Modules.

Our approach to finding candidate modules within large library is to rely on the human's demonstrated ability to scan a list of candidates and quickly discard uninteresting items. That is, we do not feel it is necessary for a classification system to uniquely pinpoint a suitable module; indeed it is probably futile to try to do this. In fact, the user may often browse a collection of modules looking for inspiration for how to accomplish a task.

Two simple techniques for classifying modules are likely to be used. First, descriptive terms indicating module functionality (e.g. sort routine, database object) would be used, where several such terms apply per module. These are used in the obvious way to do a first-cut reduction in the candidate module list. Second, data associated with module's inputs and outputs will often be typed with distinctive data types, so the available modules for a certain type can be quickly found. Both of these mechanisms could be used on a generic basic (e.g. using widely understood descriptive terms and types) or on a custom basis where an explicit collection of modules is desired by the user, using a specific term or type.

5.2.1.2 Selective Reusable Modules.

Our approach to module section is to 1) provide a range of descriptive information associated with each module, 2) provide sample test cases which illustrate behavior, and 3) provide the ability to dynamically execute a module to watch its operation under specified parameters. The first approach would consist of providing means to store and retrieve a) English text, b) a PSDL graph, and c) input-output data types for each module. The second approach involves storing data instances for module inputs which give illustrative typical data plus extreme cases used for functional testing. The third approach involves having the PSDL interpreter work effectively in executing components interactively, meaning that the interpreter can be invoked on any component with little overhead effort. By this collection of methods we hope to provide the user sufficient insight along some useful dimension of insight that detailed examination of module code will not be necessary.

5.2.2 Incorporating Reusable Modules

Utilization of an available module requires two kinds of activity: customizing it to the specific use in mind, and actual insertion into the graph being constructed. Both of these require certain editing capabilities.

5.2.2.1 Customizing a Module.

We must assume that existing modules must be somewhat modified for the new usage. This can occur in three ways. First, if the change was anticipated then there is a parameter value to be set. For example a sort routine might be selectable to determine if outputs are sorted in ascending order or descending order. In PSDL these parameters are connected to external inputs, and the editor can provide constant values to set the parameter. Second, input/output data elements may have to be adjusted to match the types of the modules. For example if the input data is a record of fields of which only one field is used by the module, then the selection of the proper field is done by an additional selection function inserted outside the reusable module in the graph. This common need for input/output type adjustment is often achieved by enclosing the existing module inside a customizing layer, namely a new graph with conversion routines, to create a modified module. Third, the internal structure of the existing module may need tweaking. For this, a copy is made under a new name and altered as needed, using graphical editing facilities. This is common when the input data has an extra data element which requires processing within the module.

5.2.2.2 Inserting a Module.

Connecting a module in place in a graph requires specifying input/output alignments between the ports of the module and the arcs of the graph. This is the normal binding of actual parameters to formal parameters, which serves to rename a data instance from its name in the graph to its name in the module. The second aspect of insertion is describing initial values. All data elements have default values associated with them, which can be set to achieve desired behavior when modules are initialized or to compensate for lapses in specification.

5.3 Using the Prototype

Once a system specification has been developed it will be refined in two types of processes. First it will be debugged, namely to make it run up to the analyst's expectations. Second, it will be used to carry out end-user experiments, to refine the prototype up to end-user expectations. The debugging activity will be carried out using a PSDL interpreter, so that the analyst can correlate internal system actions with external system results. The prototype experiment activity will be carried out with independent observers watching external outputs (e.g. user screens), to see if desired actions occur. In both activities, the prototype will be modified, extended, and permuted.

5.3.1 Interpreted Prototype Execution

Interpretation allows the analyst to watch execution of the prototype, to see internal operations and the progression of data transformations. Executing on a graphical language such as PSDL, an interpreter can provide good visual indications of program progress, and this is an advantage of a graphical syntax. Interpretation involves several human factors issues in actual execution, covered in section 5.4.2.

Interpretation also requires correct implementation of a number of features which are peculiar to prototyping: timing interpretation, interpreting incomplete specifications, interpreting generic modules, developing an execution sequence for a declarative language description, and dealing with poor performance.

5.3.1.1 Interpretation of Timing.

By its character, a functional prototype will not exhibit the same timing as a later optimized implementation, so clocking the execution of a prototype will not indicate target system performance. Also, by its character, an interpreter will run slower than a translated prototype, so the sense of time is further distorted. Two types of timing must be dealt with: 1) explicit programmed delays, which affect the logical behavior of the program, and 2) implicit performance constraints which affect the valid operation of an implementation (in real-time systems).

Two approaches might be taken to providing time interpretation; the choice between these two depends on implementation difficulty and has not yet been made. First, an explicit simulated-time clock can be introduced which causes asynchronous events at specified times. The rate of simulated time can be adjusted, relative to actual computation time in interpretation, to cause different combinations of explicit timed events in the system. This approach provides no insight into the performance aspects of non-programmed implicit constraints. Second, a time-based scheduling strategy can be built into the interpreter to supplement its precedence-based scheduling. For this to work, each library module would be given estimated execution times, and overall execution times could be calculated during interpretation. This could be done assuming that all processes are executed concurrently when logically possible, so the resulting execution time gives a bound indicating the fastest possible execution.

5.3.1.2 Incomplete Specifications.

Prototyping will usually be carried out (at least initially) on critical parts of a large system, leaving uninteresting or easily envisioned parts of the system unspecified. It is desirable that the prototype run despite this lack of specification, with minimal effort on the part of the analyst.

The principal means for doing this is with default mechanisms. Unspecified data types should default to a text string. Unspecified functions need a variety of options but at least should cause some output activity. Unspecified data instances should use defaults associated with the data type. All of this is most easily accomplished by having a library of default modules ava'lable to the user: generate random numbers within an output type; go to the screen for guidance; pick randomly among the outputs available; etc. This is an illustrative case of how a standard set of library modules is an important component of a prototyping environment.

5.3.1.3 Generic Modules.

A library of reusable modules will have many generics included (generic in an Ada sense of being tolerant of variation of in/out data types). In Ada these are mostly specialized (instantiated) at compile time, but when an interpreter is being used there is no prior opportunity to specialize them. Therefore the interpreter must provide facilities for special treatment of generics.

Two alternatives are being evaluated. First, the interpreter might interrupt execution long enough to do a fast compile of the generic, knowing that it would need to be done only once per function (not each time the function is called). Second, the type information for the execution data would be available to the generic module so that it could interpret this data at runtime, but this means the interpretable version of a given generic module would be more difficult to write than the translatable version.

5.3.1.4 Operation Sequencing.

Since PSDL is not a procedural language, its sequence of executions must be inferred from precedence relationships. Unlike functional languages, it tolerates cycles and a variable number of output messages from a function's execution. The result is that some analysis of the PSDL graph is needed prior to interpretation, to determine relative priority of executions. Also the interpreter uses an internal queue of pending processes to handle the dynamic aspects of scheduling. While simpler methods might be used for many PSDL graphs, the interpreter has no prior knowledge of when some function may create multiple output messages, so the most general sequencing methods are used at all times.

The execution of a pre-execution analysis on a graph (called static analysis) occurs the first time a new graph is encountered (but not upon later re-entry to the graph). We are assuming that individual graphs will be small enough (say a dozen nodes) so that the static analysis step does not interfere with the performance of the interpreter.

5.3.1.5 Interpretation Performance.

While an interpreter is not expected to run in real time, its performance must not be so poor that it taxes the patience of the observers. There is a reasonable prospect that the PSDL prototype interpreter will not be unreasonably slow because of the tendency in most systems for the time consuming computations (e.g. FFT's, matrix inversions, compiles, etc.) to be performed inside existing optimized modules. The interpreter serves mostly to invoke those modules, and so represents a small percentage of overall execution.

When there are exceptions to that, performance can be improved by translating an internal frequently-used graph into compiled form either automatically or manually. A principal candidate for hand coding will be data storage objects which perform frequent updates and were directly coded in PSDL. If these objects perform their own space management they can save some unnecessary data copying, but this requires some design effort that is not generally needed.

5.3.2 Prototype Experimentation

When the analyst is presenting prototype results to a mission user, often with on-line displays, a sequence of modifications to the prototype are carried out in the course of fixing errors and trying alternative specifications. There are two aspects of interest in this: display of prototype results, and management of specification changes.

5.3.2.1 Displaying Prototype Results.

Effective experimental procedure will require doing more than just simulating a target system's functionality. It will often require making visible the internal results which are normally hidden, so that observers can better understand system behavior. For example, internal database contents, communications-line traffic, and overall system state are often of interest. Principally, this is a matter of experimental technique rather than a specific requirement on prototyping facilities, so there is no explicit support for this in PSDL. However, the library of reusable modules needs several capabilities that the analyst can insert for display purposes.

5.3.2.2 Specification Changes.

A prototype must be built to be changed. The employment of reusable modules almost automatically ensures some adaptability, because those modules are built for the explicit reason of being flexibly used. The support needed from the prototyping system is assurance that changes are easy to make and to manage.

Making changes usually involves modifying functions and data types. Changing functions often is done by plugging an alternative implementation into one position of a graph without otherwise modifying the graph. This concern helps shape the database representation of a graph and its relationship to components, so that few user actions are required to change between components. Changing data types should also be straightforward, but is complicated by the need to change the functions (methods) that interpret those types, when those functions are not generic. An implementation goal is to permit substitution of one type data object for another with a small number of changes. This is difficult in the case of heavily shared objects, such as a database, which are referenced in many places throughout a hierarchy of graphs.

Managing changes requires the ability to recover from undesirable changes. This is principally a database versioning and auditing problem, but requires the ability for the user to name and retrieve partially edited versions of a specification.

With these editing and versioning capabilities, the PSDL/SBMS system can be viewed as a graphically-controlled configuration management system which permits the rapid assembly of modules into an alternative prototype for experimentation.

5.4 User Interface of Tools

The exact characteristics of a user interface such as this are very pliable, and are least subject to advanced specification since they depend on human reactions rather than computer science theory.

Our objective in implementing this system is to fully utilize the emerging technologies for letting the user make decisions by selection from options available, rather than requiring the typing of a command. This is usually faster, and generally eliminates the need for a "help" facility as it is commonly implemented. Of course the major problem in this strategy is to not give the user unnecessary (at that time) options which require multiple choices to do predictable things.

A second strategic aspect of the user interface is the need to display complex information without cluttering the display. The general solution is to utilize the rapid interactivity of modern workstations to let the user chose alternative views of a displayed object. In PSDL terms this implies the need to switch rapidly between, say, text and graph descriptions of a component.

A third strategic objective for user support is the ability to switch rapidly between tools used for manipulating a displayed object. For PSDL this means switching between the editor, browser, and interpreter without changing the set of objects on the screen. This essentially implies that tool switching is nothing more than putting a different population of options in the pop-up menus shown the user. This requires an effective data management system to preserve the system state across tool invocations.

The facilities needed for the specific PSDL tools are discussed within this framework.

5.4.1 Editor Interface

To create a PSDL specification, as a hierarchy of graphs, it is necessary to view and edit a variety of data elements: text descriptions, graph components and connections, formal to actual bindings, data types, and data values. These require the usual set of add/delete/copy/move capabilities. Each type of data requires a little different set of capabilities. One of the major costs in building such a system is the provision to add a specialized editing facility for each class of object. A particular concern for a common type of reusable module, code generators, is the need for a specialized interface for each one. Typical generators are in three classes: screen format definitions, parsers (e.g. YACC) and expression evaluators. These take user inputs and generate a module suitable for later execution, and each is complicated enough to require specialized user interface support.

A particular concern in the editing process is the definition of a hierarchy of graphs. That is, when a graph is being displayed and a component graph is also desired, what is the mechanism for juxtaposing them? Since there is no simple answer to this question, the solution is to provide the user with an extensive windowing capability for shaping the display to the character of the graphs being observed. This involves opening/moving/growing/panning/ closing windows.

Another concern in the editing process is to minimize the number of user actions necessary to construct a given image. This is best done by anticipating what the user will do next after a given action. For example, after a user has inserted a component in a graph, will the next set of actions concern connecting that component to other components or will they deal with further defining the details of that component? In this example either sequence might happen, so a user interface that favors one sequence over the other will cause dissatisfaction. The solution to these problems is a lot of experimentation, which is one reason why we build prototypes.

5.4.2 Interpreter Interface

During execution of the interpreter the user wishes to see an animation of the activities as they move from graph to graph. This requires facilities to view activations, see intermediate results, set up an execution, and control the execution.

Dynamic Views During Interpretation.

As each function is executed on a graph its corresponding bubble can be highlighted and, if appropriate, its internal graph opened up to show execution. Some types of debugging favor an approach where every graph is opened as it is executed so the program progress is shown by graphs flashing on and off the screen. Alternatively, the user wishes to focus on a certain graph and to view activity only as it is seen in that graph.

In addition to watching activations, the user will want to observe certain data conditions. This can be done by setting a "tracepoint" on an arc, which displays each data item traversing that arc. Open issues exist in where to position the data display and how to best display complex data structures.

Static Views During Interpretation.

When interpreter execution stops for some reason, usually at a break point, the user may wish to view many aspects of the program state and perhaps to alter state data. The full power of the browser and editors should be available at this point for looking at data and graphs, and modifying parameters, data, and graphs. Of course continued execution of the interpreter after modifications may not be coherent, in which case the user would have to restart interpretation from the beginning.

Interpreter Set Up.

The interpreter can be used to execute any PSDL component, but certain adjustments need to be made initially for successful execution. First, input data values are needed to stimulate the system. Second, it is useful to set breakpoints (to stop operation), watchpoints (to conditionally stop operation), and trace points (to display a data stream) at selected points in the graph or its components. Third, if a prolonged execution is anticipated then the session should be named so that it can be restarted later at various times in the course of experimentation.

Interpreter Execution Control

As the interpreter is executing, the user needs certain controls: start, stop, single-step, faster, and slower. The speed controls refer to the size of delay inserted for each function executed to make the animation understandable to the user. All other control can be exercised by the user when the execution is stopped, and so do not need dynamic controls.

5.5 Human Interface Summary

The discussion here has focused primarily on support for the analyst in constructing and running the prototype, as opposed to end-user viewing of results. The end-user view depends on the skill of the analyst, but the analyst's productivity depends on the skill of the prototyping system developers.

Many aspects of this discussion were given in the form of requirements for necessary features. Sometimes it is obvious how such features might be implemented, and other times it is not clear. This is characteristic of user interface specifications. The actual functionality and format of presentation in the delivered system requires experimentation under any circumstances, because human reactions are difficult to predict. Fortunately, these aspects of a system design are often flexible and can be tuned. That is why prototyping has value.

6. Conclusions

While this initial effort was only intended to create a design for PSDL, SBMS, and support tools, it is reasonable to evaluate the system status at this point. We believe there has been enough success in attacking each of the major challenges confronting prototyping to be optimistic that an effective system can be developed. However open issues remain in each area as to whether or not people will really find the proposed facilities operationally useful. Only experimentation will answer that question. In addition, a PSDL environment needs to be extended to address design aspects (e.g. performance) of the target system, in order to be a widely effective tool.

6.1 Successes

The major challenges facing this prototyping facilities development effort are 1) providing an effective environment for reuse of standardized code modules, 2) developing an executable interconnection language for simplified specification of a prototype, and 3) bringing the tools of hierarchical structuring and graphical programming into the user's hands.

6.1.1 Support of Reuse

The principal support of reuse is a) SBMS facilities for finding modules plus associated data which facilitates understanding those modules, and b) an interpretation capability which helps understanding of how a module fits into a given problem. The SBMS aspect was anticipated, but the realization gained during this effort was that large amounts of peripheral information about a module, such as test sets and illustrations of prior use, can be necessary selection aids. The value of the interpretation facility is yet to be tested, but it appears promising, if the user finds it easy to set up and interpret execution experiments.

6.1.2 System Descriptions

Configurations of components are assembled using PSDL. The language serves to describe the interaction of those components in a simple semantics which is executable. We feel that a language has been defined which in fact eliminates many control and structuring decisions, and that this is an effective general-purpose language. Whether or not its use provides a substantial savings in effort on the part of the prototype specifier is open to experimentation, but we feel there are good prospects for that to be the case when the proper support tools are in place.

Our experience with doing example programs in PSDL gives some further insights about its strengths. We might compare it to a configuration management capability (such as "Make" in Unix systems); PSDL is more than that because it describes the semantics of interactions of the components being brought together. PSDL "externalizes" the control structure so that the individual components need not be aware of the overall program logic. We might compare PSDL to functional languages which provide a control-free description of a system (such as FP); PSDL imposes less serialization on the description and so simplifies the programming effort required for applications where there is natural asynchrony or concurrency. We might compare PSDL to dataflow or other diagrams used in design specification and documentation; PSDL has less flexibility in terms of displaying multiple dimensions of a system, but its executability means that the specification involved can be animated and checked.

6.1.3 User Interface

The opportunity to use interactive graphics for dealing with complexity in design information is becoming an economical reality, but the techniques for doing that are still evolving. We have developed a graphical syntax for our structures and have made hierarchical decomposition an effective tool. These techniques serve to achieve higher effective bandwidth to the user. Our hierarchical structuring, in contrast with that available in block-structured languages, in that each level can be given a useful abstraction function, so that the user can read and program each level as a program in itself.

Of course the overall user interface will evolve significantly with future use, but the promise for high effective bandwidth seems achievable.

6.2 Open Issues

As we gain experience in using and presenting PSDL, several concerns are raised which can only be resolved by usage experience. These occur in the same areas as the technical challenges: reuse, language, user interface.

6.2.1 Reuse of Components

Prior attempts at software reusability have not always been wildly successful. This is due in part because existing software is not written for reuse and because potential users have trouble learning about candidates for reuse. The hope for improved reuse in a prototyping system occurs in several forms. First, functional level components have fewer design idiosyncrasies to disqualify them from reuse. Second, components in a prototype will be built with standard interfaces and within a common data-type dictionary so the probability of their plugging together is increased. Third, the use of generic facilities in Ada permits modules to be more tolerant of data types, and so may have wider applicability. Fourth, components written in PSDL will be relatively easier to modify because they have not been optimized for high performance and so exist in a more easily understood form.

Despite these improvements, effective reuse may prove difficult to achieve outside specialized application domains. Operational experience will be necessary to evaluate this.

6.2.2 Description Language

The desire to describe systems at the functional specification level is adventuresome because it has not happened before. System analysts are accustomed to working at that level of description, but not to the degree of precision necessary for executability. Programmers are not accustomed to dealing with abstract functionality separated from implementation concerns, and they may require a significant reorientation to become effective users of PSDL. Thus, while functional specification has many attractions, it may be difficult to get people to use it, particularly when it is unfamiliar.

6.2.3 Graphical Syntax

Is programming in a graphical syntax more or less efficient than in a textual syntax? The experience with graphical interfaces in some other application areas (e.g. MacIntosh programs) indicates that graphics is good for beginners but can get in the way of experienced users. That depends on the type of application and on the degree of development of the user interface, namely how well the user can bypass non-productive actions. Again, usage is necessary to gain insight into these issues.

6.3 Extensions: Design Prototyping

A very desirable prototyping capability is to evaluate performance properties of a system design, and that is a very natural next step after functional prototyping. To accomplish this, PSDL would be extended to describe system concurrency, and transformations would be added through which the design decisions could be modified without changing functionality. Other capabilities would be needed for optimizing data management and inter-process synchronization.

PSDL as it is now defined appears to be a good vehicle to support these extensions and some of these capabilities may be added with minimal effort.

International Software Systems expects to continue to develop prototyping concepts and associated support tools, so the work performed under this contract should prove of value in later contract efforts.

7. Bibliography

1. Balzer, R., Konrad, M., et al., "RADC Requirements Engineering Testbed Research and Development Program Panel Recommendation", Technical Report for Contract No. F30602-85-C-0129.

2. Booch, G., Software Engineering with Ada, Benjamin/Cummings, 1983.

3. Honeywell Computer Sciences Center, "RaPIER (Rapid Prototyping to Investigate End-user Requirements), Final Scientific Report to the Office of Naval Research", Contract No. N00014-85-C-0666.

4. IEEE Transactions on Software Engineering, Software Reusability (special issue), September 1984, Volume SE-10 #5, pp. 474-609.

5. International Software Systems, Inc., "Research in Software Reusability, Final Report", Final Report for Contract No. DASG60-83-C-0004, TR-P106/83/0004-3.

6. Welch, T.A., "SBMS Software Base Management System", ISSI Technical Report, January 30, 1986.

7. Welch, T.A., "PSDL Prototype System Definition Language", ISSI Technical Report, January 30, 1986.

8. Yeh, R.T., Roussopoulos, N., Chu, B., "Management of Reusable Software", IEEE COMPCON 84, September 16-20, 1984, pp. 311-320.

9. Yeh, R.T., "A Framework for Software Evolution and Reuse", submitted for publication.

8. Contract Summary and Related Efforts

8.1 Financial Summary

STARS funding for this effort, for the period June 1985 through March 1986 was $148,661.

8.2 Schedule

This work was intended as the first phase of a multi-year effort. It was originally scheduled for June 1985 though December 1985. Due to delay in funding approval for the continuation work, the first phase work was stretched out into March of 1986 in the hopes of providing continuity with the subsequent work. When the decision not to fund the continuation was made in late March 1986, we faced the choice of filing a final report describing a fragmentary study or delaying the report in order to make it a self-sufficient document. The latter approach was used, so this document is quite late because much of the work on it has been done outside contracted hours.

8.3 Other Related Efforts

Concurrent effort has also been carried out on a Rome Air Development Center contract #F30602-85-C-0129, "VHLL System Prototyping Tool". This contract entails delivery of an operational prototype of a prototyping system, utilizing many of the design features identified in the STARS contract. The RADC work is being focused on command and control applications.

Appendix

A.1 PSDL Example

In order to illustrate the use of PSDL, a standard example is developed here. The example is Booch's Environment Monitor Problem [2] in which sensor signal values are read, logged, and compared to preset limits. This example shows some elements of real-time programming, and is used by Booch to illustrate an object-oriented programming approach in Ada.

The intent of the illustration here is to show the style in which PSDL might be used. Unfortunately, we have not yet done a good job of figuring out how to present PSDL structures on paper, since the language is oriented toward electronic interactive presentation. Normal usage of PSDL would employ two or more levels of description, where a top-level graph has components which are defined by lower-level graphs. Electronically it is easy to browse through the upper level graph and look at subgraphs, text descriptions, data types, etc. On paper we have the choice of having too many disconnected diagrams, or of having overly cluttered diagrams, or of omitting some useful information. The compromise need here is to try to express the whole application in one level of diagram, with only primitive elements separated, and to omit much of the information which defines actual application details. The resulting diagram is not at this point usefully executable because data type information is missing; this is straight forward detail except for the limits check function, where expressing constraints require a little thought.

Included with this Monitor Problem example are descriptions of the basic library modules which would be needed to make the system executable. Most of these are standard reusable modules which would be readily available Three of them are application-specific, involving access to specialized external devices, and that code would have to be written by the prototype developer. This is indicative of the general case that a prototype run in a specialized environment will need custom interfacing code.

A.2 Problem Description

The diagram in Figure A-1 has three major data flows, which are horizontal. At the top is the mechanism for end-users to adjust parameters in the monitoring process. The user has a command interface by which sensors can be enabled/disabled, limits set, and values selectively logged. After each command a confirmation is printed back on the terminal. The user interface, with error checking and prompts, is encapsulated in a standard module. The middle row shows a periodic reporting mechanism, by which sensor status is logged. The bottom row shows a mechanism which brings sensor values into the internal data structures periodically. The period for doing this is unspecified in the requirement, but would depend on how frequently the limits should be checked. The apparatus in the lower right is a time-out mechanism which causes as alarm if a sensor fails to respond in a given time period. (We believe that the time-out does not fit well in this problem, but was grafted on to illustrate the concept.)

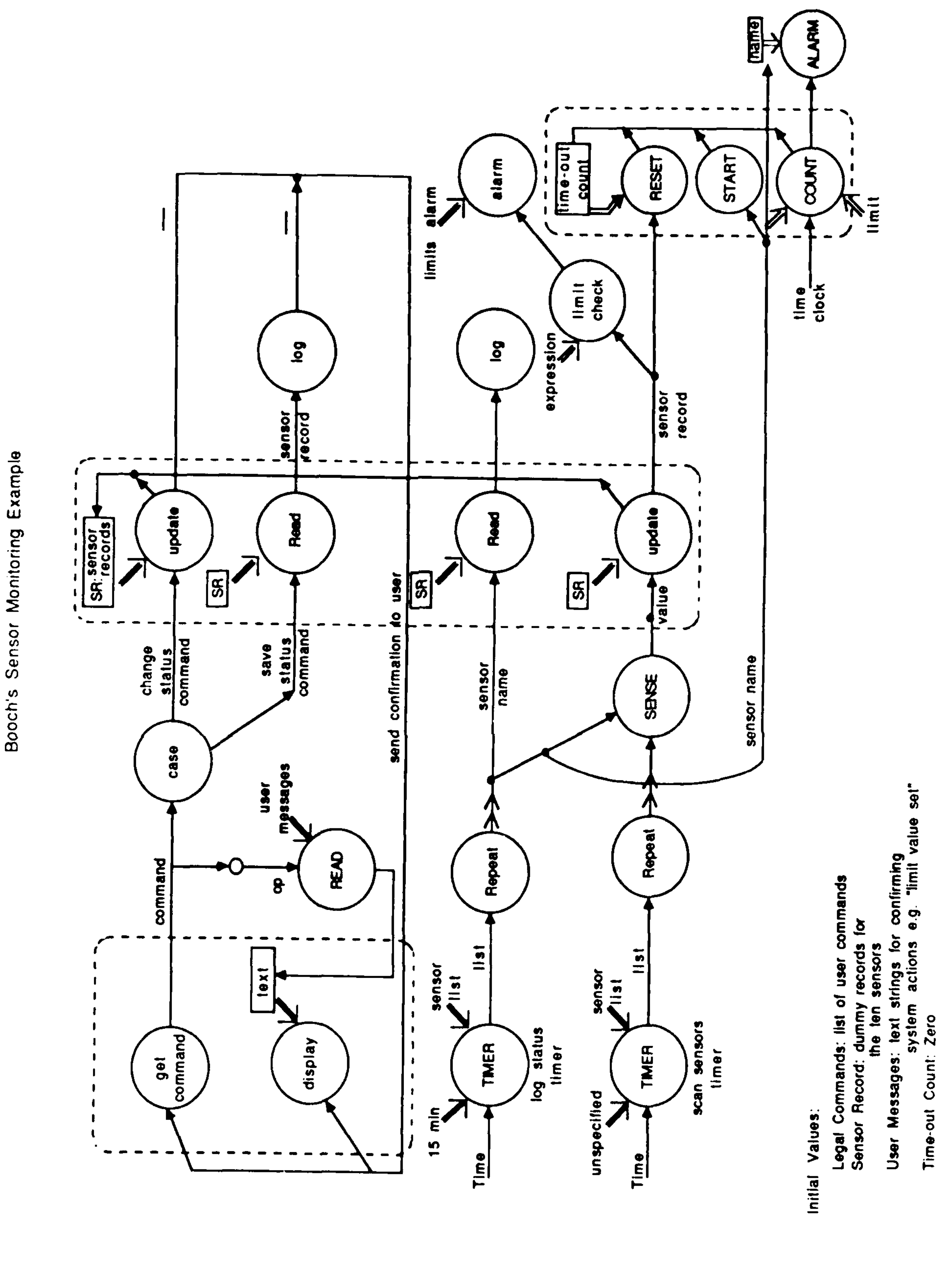
What is not presented in this formulation is the data structure of the stored data, the command set, or the confirmation texts. Also not shown are the constraints expressions which make up the limits check. These would be necessary to understand the details of the system but not the overall logic flow.

This formulation matches Booch's requirements [2], which are ambiguous. It does not match Booch's code in two aspects which we think are improvements: 1) the time-out check is made a little more useful, and 2) the sensor scanning function is explicit in PSDL, whereas it was an implicit use of idle time in Booch's code. The latter change marks one distinction between a functional specification and an implementation.

A.3 Analysis

It is true that reading this PSDL diagram does not make the system functionality immediately obvious. However, after a little study, the overall system logic becomes much clearer than it does reading Booch's code (which is good code but still obscure). Most evident is that the interaction between subsystems is much clearer in the PSDL form. When code is automatically generated from this PSDL graph, however, it will likely be less efficient than Booch's.

Note that this example illustrates two features which we find valuable in any specification language. First, the intermixture of dataflow and object structures permits the viewing of simple transactions (data flows) and also the interactions of different types of transactions (via objects). Second, all of the reusable components used here are general-purposed in character, and so are applicable to a wide range of applications.

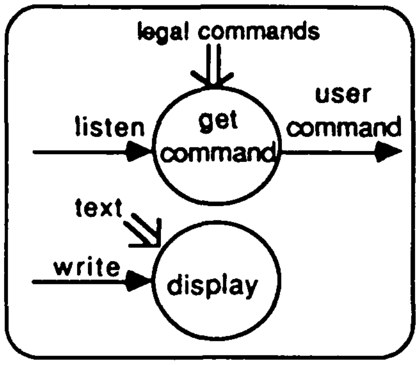


Initial Values:
Legal Commands: list of user commands
Sensor Record: dummy records for the ten sensors
User Messages: text strings for confirming system actions e.g. "limit value set"
Time-out Count: Zero

Figure A-1 - Example PSDL Program

Reusable Modules

Methods	Data types



write: ____

listen: ____

legal commands: **array of** user_command

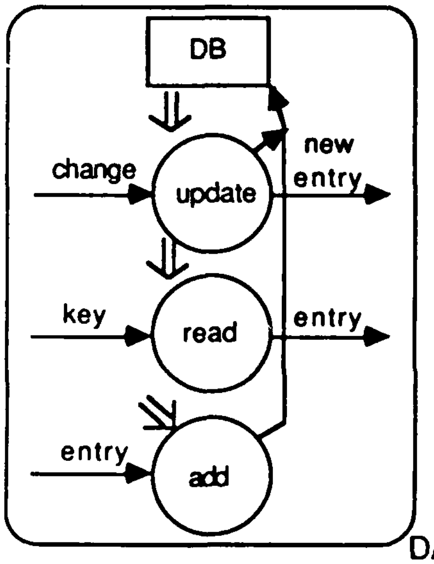
user_command: [op: integer; **array of** field]

field: number|text

text: character string

USER INTERFACE OBJECT

Specification: A text style user interface is provided; in a listen activity, every legal command typed by the user is output; text is displayed back to the user.



DB: **array of** entry

new entry: entry

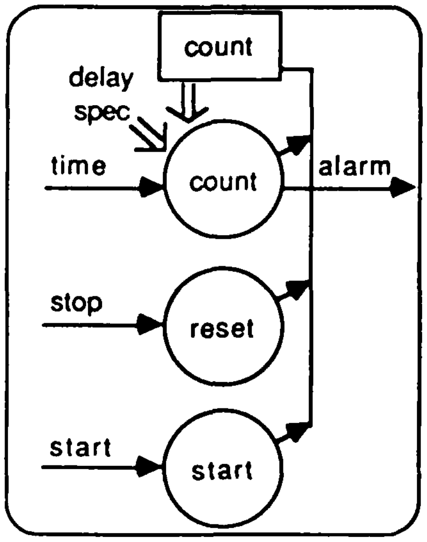
entry: [key: integer; **array of** field]

field: anything

change: [key: integer;
fieldname: integer;
value : anything]

DATA STORAGE OBJECT

Specification: An array of records is updated, added-to, and read from in convertional ways.



count: [elapsed_time: integer;
on/off: boolean]

delay spec: integer

time: ____

stop: ____

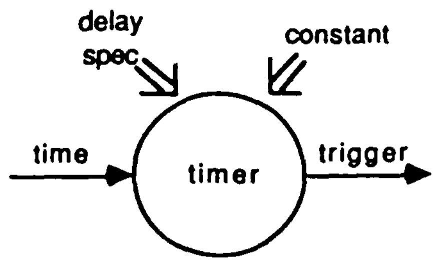
start: ____

alarm: ____

TIME OUT OBJECT

Specification: When the timer is started, it produces an alarm after the delay unless first reset.

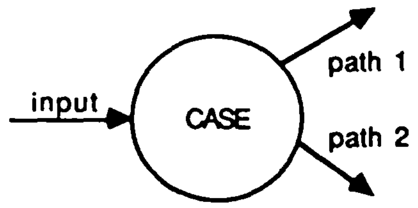
continuation of resuable modules



time:	____
delay spec:	integer
constant:	X : anything
trigger:	X

TIMER MODULE

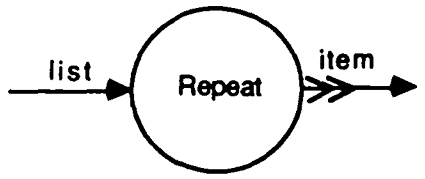
Specification: Every [delay spec.] time units a trigger message is sent whose contents equal the constant provided.



input:	message
path 1:	message
path 2:	message
message:	[op: integer; anthing]

SELECTION MODULE

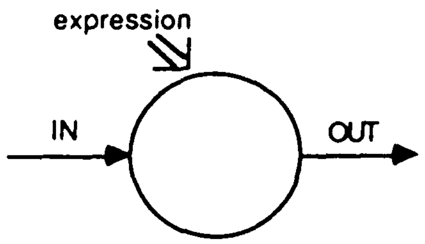
Specification: An output path is selected based on the first field of the input: outputs are mutually exclusive.



list:	**array** of item
item:	anything

REPEAT MODULE

Specification: When a list of items is received, each item is sent out as a separate message. This performs the function of interation in many cases.

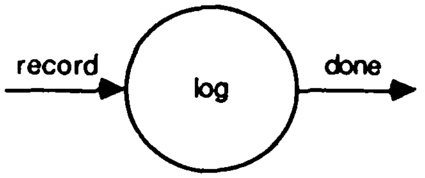


IN:	anything
OUT:	anything
expression:	character string

EXPRESSION MODULE

Specification: This module interpretively executes an arbitrary arithmetic expression to calculate output values as a function of input values.

Application - Specific Modules (encapsulate outside events)

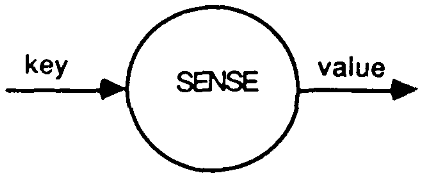


done: ______

record: anything

LOG THE STATUS

Specification: Has access to an external file, and writes the input data onto the file.

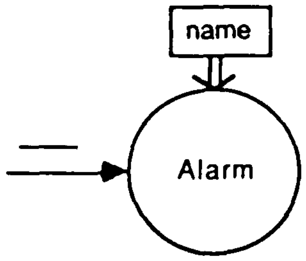


value: [key; integer]

key: integer

READ SENSOR VALUE

Specification: Has access to externally derived sensor values, and reads them.



SET OFF ALARM

Specification: Has access to external alarms, and activates the alarm designated by name.

END
8-87
DTIC