1988

Languages for Specification, Design, and Prototyping

Berzins, V.; Luqi

Naval Postgraduate School

V. Berzins and Luqi, "Languages for Specification, Design, and Prototyping", Technical Report NPS 52- 88-038, Computer Science Department, Naval Postgraduate School, 1988.

https://hdl.handle.net/10945/65280

T30

NPS52-88-038

NAVAL POSTGRADUATE SCHOOL

Monterey, California

LANGUAGES FOR SPECIFICATION, DESIGN, AND PROTOTYPING

Valdis Berzins

Luqi

September 1988

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

The work reported herein was supported in part by the National Science Foundation and the Naval Postgraduate School Research Foundation.

This report was prepared by:

[redacted]

LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]

KNEALE T. MARSHALL
Dean of Information
and Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-038	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b. OFFICE SYMBOL *(If applicable)* 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA. 93943	7b. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct Funding
8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA. 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
LANGUAGES FOR SPECIFICATION, DESIGN, AND PROTOTYPING (U)

12. PERSONAL AUTHOR(S)
BERZINS, Valdis, LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Annual	FROM 87/09 TO 88/08	1988, Sept	73

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer aided software engineering, rapid prototyping, specification, real-time software, embedded systems, software design, reusability

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
This report is about specificaiton, design and prototyping languages supporting new paradigms for software development. The languages used in the Computer Aided Software Engineering paradigms differ from the ones in traditional software development because of the need for supporting a higher level of automation at the early stages of software development. Analysis and comparisons among these languages are discussed in detail in this report.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE *(Include Area Code)* (408)646-2735; 22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

UNCLASSIFIED

Languages for Specification, Design, and Prototyping[1]

Valdis Berzins

Luqi

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

This report is about specification, design, and prototyping languages supporting new paradigms for software development. The languages used in the new CASE paradigms differ from the languages used in traditional software development because of the need for supporting a higher level of automation at the early stages. The traditional software life cycle consists of a series of phases sometimes called requirements analysis, functional specification, architectural design, module design, implementation, testing, and evolution (Fig. 1). The result of each phase is a document serving as the starting point for the next phase, or an error report requiring reconsideration of the earlier phases. Traditionally the phases before implementation have been carried out largely by manual processes, and the resulting documents have been expressed in informal notations. The implementation phase produces a document expressed in a programming language. Programming languages are formal notations that can be processed by a variety of automated tools, such as compilers, static analyzers, debuggers, execution profilers, etc. Most of the computer-aided design in traditional software development environments is

[1]This research was supported in part by the National Science Foundation under grant number CCR-8710737.

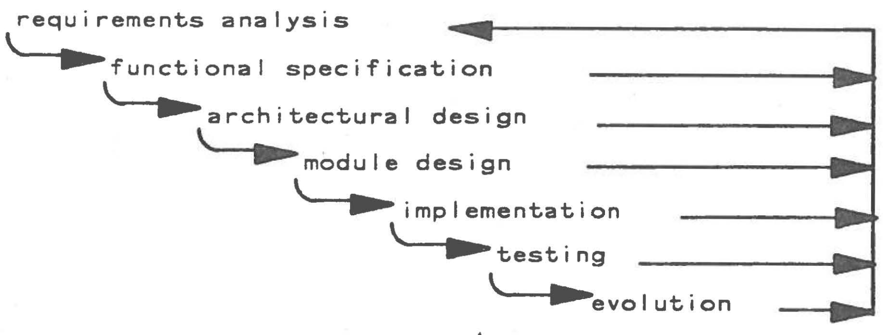

Fig. 1 Traditional Software Life Cycle

applied in the implementation and later phases.

A formal language is a notation with a clearly defined syntax and semantics. Formal languages are critical components of a CASE environment because they are needed to achieve significant levels of computer-aided design with currently feasible technologies. Automated tools are capable of detecting structure in a notation only if the structure has been formally defined, and responding to aspects of its meaning only if the meaning of the aspect has been formally defined. The tools applicable to informal notations usually treat them as uninterpreted text strings, which limits the tools to bookkeeping functions such as version control. Notations with a formally defined syntax but an informal semantics can support tools sensitive to the structure of the syntax, such as pretty printers and syntax-directed editors. If both the syntax and semantics of a special purpose language have been fixed and clearly defined, it becomes possible to create automated tools for analysis, transformation, or execution of the aspects of the software system captured by the language and its conceptual model.

The new software development paradigms are a response to problems with traditional development methods, which take a large effort to produce systems that do not meet the customer's needs very well. These problems are caused largely by labor intensive tasks at the early stages of the development. Currently there are several ways to approach the problem. One approach is to automate some of the tasks in the early stages of the traditional life cycle. Examples of this approach are work on executable specification languages and formal verification. Another CASE approach introduces the prototyping software life cycle.

A software prototype is an executable model or a pilot version of a proposed system. A prototype is usually a partial representation of the proposed system, used as an aid in requirements analysis and system design through an iterative process of negotiation between the systems analyst and the customer. The construction activity leading to such a prototype is called prototyping. The customer describes the requirements, while the analyst interprets them and builds a prototype. The analyst then demonstrates the execution of the prototype to the customer. The requirements are adjusted based on feedback from the customer and the prototype is modified accordingly until both the customer and the analyst agree on the requirements. This process is illustrated in Fig. 2.

Formal languages for the specification, design, and prototyping of software systems are needed to support the new CASE paradigms, since they involve computer-aided analysis and design from the earliest stages of software development. In the new paradigm, the goals and required behavior of the intended system are negotiated in the con-

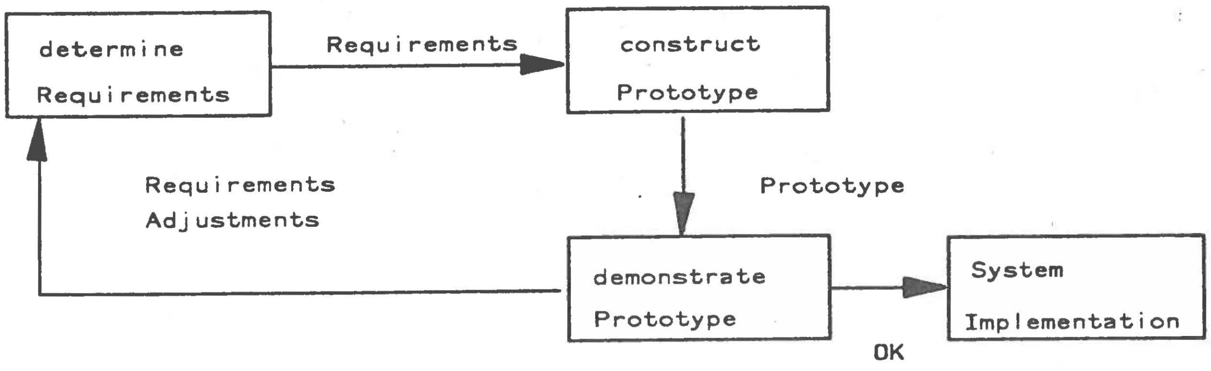

Fig. 2 The Prototyping Life Cycle

text of a computer-aided analysis of the customer's problem.

A CASE environment with knowledge-based assistants for each phase of development starting with requirements analysis is an example of this approach. The computer-aided aspects of the process include completeness and consistency checking, displaying descriptions of the system from various viewpoints, demonstrations of prototypes, concurrency and configuration control for the design data, and information retrieval functions. The CASE tools in such an environment depend on each other, and must be integrated together to meet this goal. Such integration depends both on formal languages and emerging technologies for managing engineering databases [27-29, 38].

The purpose of a specification language is to record a specification. A specification is a black-box description of the behavior of a software system or one of its components. A black-box description explains the behavior of a software component in terms of the data that crosses the boundary of the box, without mentioning the mechanism inside the box. A specification language should allow simple abstract descriptions of complex behaviors that can be easily understood by people and mechanically analyzed.

The purpose of a design language is to record a design. A design is a glass-box description of a software system or component. A glass-box description gives the decomposition of a component into lower level components and defines their interconnections in terms of both data and control. A design language should allow simple abstract descriptions of system structure that can be easily understood by people and mechanically analyzed.

The purpose of a prototyping language is to define an executable model of a system, using both black-box and glass-box descriptions. Some meta-programming and functional programming languages have similar properties. However, a prototyping language has no obligation to give detailed algorithms for all components of the system as long as it is descriptive and executable. Prototyping languages and programming languages have different evaluation criteria: a prototyping language is optimized to allow an analyst to create and modify a working system as quickly as possible, while a programming language is optimized to allow a programmer to produce a time and space efficient implementation. A prototyping language supports simple and abstract system descriptions, locality of information, reuse, and adaptability at the expense of execution efficiency. A prototyping language should have facilities for recording specification and design information, subject to the constraint that the final product must be executable.

The difference between specification and design languages is the difference between interface and mechanism: a specification says what is to be done, and a design says how to do it. An important purpose of specification and design languages is to serve as a precise medium of communication between the members of a development team working on a large system. The evaluation criterion for both specification and design languages is the ability to support simple, concise, and humanly understandable descriptions of complex behavior. It is useful for specification and design languages to be executable, but simplicity of expression takes precedence when the two considerations conflict. It is important to be able to determine the properties of a specification and to certify that a design realizes a specification. Execution can help attain these goals, but it is not the only way to do so, and it is not necessarily the most effective way.

Prototyping languages are used in requirements analysis for the purpose of requirements validation via early demonstrations to the customer. They are also useful for evaluating competing design alternatives, validation of system structure, and feasibility studies. Prototypes can be used to demonstrate the feasibility of real-time constraints and to record and test interfaces and interconnections. Specification languages are used for recording external interfaces in the functional specification stage and for recording internal interfaces during architectural design at the highest levels of abstraction. They are also used in verifying the correctness and completeness of a design or implementation. Design languages are used for recording conventions and interconnections during architectural design and module design.

It is useful to briefly examine the history of language development, because the terminology for describing languages has been changing dramatically along with implementation technology. Originally any compiled programming language was a very high level language. As systems became more complex, the meaning of the term shifted towards design languages which can describe system structure without introducing low level implementation details and generalized components that can be adapted to many different situations. Eventually technology improved to the point where programming languages could support abstraction and generalization (e.g. Ada and Smalltalk). Systems became even larger, and the meaning of the term shifted again, towards languages describing what a system is supposed to do, without specifying how the system is to accomplish its goals. Technology is advancing to the point where some of the languages in these categories are getting to be executable as well (e.g. Prolog, Refine and PSDL). The concept of a very high level language is a moving target that depends on the current state of compiler technology and the speed, memory capacity, and cost of available hardware.

This report presents languages for specification, design, and prototyping. We discuss these classes of languages one at a time to simplify the presentation. Many of the existing languages for software development described in the literature combine aspects from several of these categories. We describe the characteristic properties and restrictions for the languages in each category, examine ways to use the languages, and compare them with other kinds of languages. This report is limited to general purpose languages that can span a range of applications.

1. Specification Languages

The purpose of a specification language is to describe the interfaces of a software system or component. Specification languages are used for formulation, analysis, communication, and retrieval. A specification language provides a set of concepts and notations which allow the analyst or designer to formulate an interface for a system or component. Notations are important for inventing large systems because people are limited in the number of items they can consider at the same time. Considering each aspect of such a system in isolation and recording the result before proceeding to the next is a common way of overcoming this limitation. The language used is important because it influences the analyst's thinking and determines which things are easy to express, and which are impossible or impractically difficult. A specification language should help the analyst to construct a simpler conceptual model for the intended system and to establish and maintain its conceptual integrity.

A formal specification language allows the proposed interface to be analyzed with respect to many different kinds of properties. At a structural level, the language can be used to help the analyst organize her thoughts and to determine which pieces of information are still missing. At a semantic level, the language can be used to determine many properties of the description and the behavior of the proposed interface. Examples of such properties include type consistency, correctness of a particular response for a particular input, the set of correct responses for a particular input, freedom from deadlock for multistep protocols, coverage of all possible input values, satisfiability, uniqueness of outputs, and consistency with a proposed design. None of these semantic properties can be determined without a precise specification.

The specification language is a medium for communication between the analysts, the development team, and the customers. The specifications often form part of the contract agreement governing a development project, and act as a primary source of information about what the designers and implementors are supposed to accomplish. Large systems involve many people over a long period of time. In large organizations oral communication is ineffective, and decisions have to be written down and circulated. Written specifications avoid the need for repeating the same information to different audiences, getting everyone together at the same time, or relying on imperfect human memories. The information in the specification is also the basis for customer review, although it may be necessary to paraphrase the information [25,52] and to provide summaries and simplified views.

Specifications also have an important role in retrieving reusable software components. To find an existing component that can perform a given task, it is necessary to describe the required behavior and match it to descriptions of the behavior of existing components. For large component libraries this is a major task that can benefit from mechanical assistance [37,38], suggesting reusable components should be stored with formal specifications. The CASE tool performing this function is known as a software base management system [55].

The primary benefits of using a formal specification language are precision and the potential for automation, which lead to better software products. The consequences of not using a formal specification language are miscommunication, a manual working style, and software that is hard to use and understand. Miscommunication is caused by ambiguity and incompleteness, which allow the author of a document to have a different interpretation than the readers of the document. Miscommunication leads to system faults. A manual working style leads to larger numbers of faults because people make more random errors than programs do and people do not have enough time and patience to do exhaustive error checking. Since informal languages do not guide the analysts' thinking or support simplifying transformations very well, systems developed without a formal specification language are often more complex than they have to be.

A specification language should have the following properties:

Precision

Each statement in the language should have a single well defined meaning.

Abstractness

It should be possible to completely define interface behavior without considering mechanisms and low level details.

Expressiveness

The language should allow brief descriptions of common system behaviors which are understandable as they stand. Abbreviations that must be expanded before they can be understood are not expressive in this sense. In addition to existing, the brief descriptions must be constructible by people in a natural way.

Simplicity

The rules describing the meaning of the language should be simple, without exceptions or interactions between multiple components. This is important both for ease of learning and ease of automation. It is also important to avoid misunderstandings, because situations where extensive reasoning is required to determine the meaning of a statement provide opportunities for people to make errors of interpretation.

Locality

The language should support localized description units with limited interactions and the dependencies between the units should be mechanically detectable. This reduces the amount of information needed to understand or modify a given aspect of a specification to a humanly manageable level, and supports mechanical aid in assembling and displaying the information needed for a single specification step.

Tractability

It should be possible to implement a wide variety of automated aids for analyzing, transforming, and implementing subsets of the specification language. While the subsets of the language that can be handled by the tools should be as large as possible, it may not be possible to cover the entire language without compromising the abstractness and expressiveness of the language.

Adaptability

The language should support the description of general purpose components and the adaptation of those components to particular situations. Generic modules and inheritance mechanisms are two well known ways to support adaptability.

Specification languages are designed for CASE paradigms following the traditional software life cycle. A specification language is used in functional specification to define external interfaces of the system and in architectural design to define internal interfaces of the system.

The relation between requirements and specifications is controversial, and there has been no clear agreement on the distinction between the two [30]. We have found the following formulation useful.

A specification defines a set of disjoint interfaces. Formally, an interface is a predicate on a subset of the possible observable behaviors of a system indicating which behaviors are acceptable and which are not.

Requirements consist of behavioral goals for the system and constraints on its development. The constraints include limits on schedule and budget. Formally, a goal is a function from interfaces to a set of utility values. In the simplest case the utility set can consist of two values, **acceptable** and **not acceptable**, in which case goals become predicates on interfaces. This corresponds to the view of a goal as an acceptance test, which does not completely capture current practice in requirements analysis. It is more realistic to view the utility set as an ordered interval of values which indicate the relative usefulness of different interfaces. This corresponds to the view of a goal as an objective function in an optimization problem.

From this point of view, requirements analysis is the process of determining the constraints and the objective function, while functional specification is the process of solving the optimization problem. The solution to the optimization problem is an interface, represented in the specification language.

In current practice the developers have only informal and approximate descriptions of the goals, which are used to guide intuitive design tradeoff decisions producing an approximate solution to the optimization problem. Requirements analysis and functional specification often overlap in time, because the design tradeoff decisions being made require more information about some aspects of the goals, in the form of more accurate approximate descriptions. It is a matter of research whether this optimization process can be usefully automated and whether classical results from optimization theory can be applied. One of the difficulties is calculating accurate estimates of the budget and schedule needed to implement a particular interface or class of interfaces. Another is that the descriptions of the goals available at any given time are incomplete and uncertain.

A validation step is required to demonstrate that the proposed interface resulting from the functional specification effort meets the real needs of the customer, given that it optimizes the formal model of the constraints and goals resulting from requirements analysis. This step is needed because of the uncertainty associated with the formal model, which is usually incompletely understood by the customer. To carry out the validation step, it is necessary to demonstrate the characteristics of the proposed interface to the customer. Since most interfaces are capable of infinitely many concrete behaviors such demonstrations are inherently incomplete, with statistical rather than an absolute conclusions.

The relation between specifications and programs is more traditional. A program determines a set of algorithms and data structures to be used to calculate the responses of a software system or component. The correctness of a program with respect to a given interface can be demonstrated by showing that all possible behaviors of the proposed mechanism are acceptable with respect to the interface (proof of correctness), and it can be refuted by exhibiting a particular behavior of the mechanism that is not acceptable with respect to the interface (testing). The specification also tells the programmer and the program generation tools what they are supposed to accomplish (implementation). In the current state of CASE technology, it is reasonable to expect that implementation will not be entirely manual or entirely automatic, but the result of the cooperation between skilled programmers and a set of computer-aided design tools. This imposes a dual burden on specification languages: the need for effective communication with both people and programs.

Specification languages are used in specification-based software design [7]. The goal of architectural design is to decompose a system into a set of simpler modules. Specification languages are used to define the interfaces of these modules. Decomposition into simpler tasks is necessary for implementing large systems whether the design is created by people or CASE tools. Precise specifications are needed to guide implementation, especially if the process is to be computer-assisted.

There has been increasing interest in executable specification languages, motivated by two main considerations:

(1) automated prototyping for validating requirements and specifications, and

(2) automated implementation of production quality software.

The main distinction between the two versions of the problem is that the first version relaxes performance constraints while the second does not. If the specification language is strong enough to be interesting, both versions of the problem are algorithmically unsolvable in the general case. The practical impact of an "executable" specification language can be judged by considering the expressiveness of the entire language, the expressiveness of its executable subset, and the relative difficulty of transforming simple but non-executable specifications into executable equivalents.

Many specification languages use some form of predicate logic for describing the constraints and properties of input and output of a black box in the system independently of the algorithms and data structures used for calculating the outputs of the box. This has both advantages and disadvantages. Quantifiers are convenient to use because they allow many problems to be specified in a simple, compact, and natural way. This allows the systems specifier to work at the black box level, concentrating on behavior of a system rather than the mechanisms of implementing the system. Quantifiers can also lead to implementation difficulties. Specification languages that include unrestricted integers and quantifiers can specify functions that are not computable. Such functions are impossible to implement perfectly, since any partially correct implementation will have some input values for which execution will fail to terminate even though the specified function has a well defined value. An implementation is partially correct if it never produces an output that conflicts with the specification. While a plausible response to this difficulty is that customers will not specify non-computable functions in practical projects, there are related difficulties that are less easily avoided.

Consider a function with an output y subject to the following specification.

```
if for all(x: integer :: f(x) = g(x)) then y = 0 else y = 1
```

This is an example of a conditional with a universal quantifier in the test predicate, in the syntax of Spec 87. The output y is to be zero if the functions f and g have the same value for all integer arguments and one otherwise. Any compiler that can handle all specifications of this form solves the equivalence problem for recursive functions, which is well known to be undecidable. According to Rice's theorem, examples of specification forms with this property are plentiful [48]. This means any specification compiler will have many specifications for which compilation will fail to terminate or will produce an implementation that either produces incorrect results or fails to terminate for some inputs. An example of the first case is an implementation strategy where the compiler tries to both prove the theorems "f = g" and "f ≠ g" in parallel, producing the constant 0 or the constant 1 as an implementation if the corresponding goal succeeds and taking forever to compile if it cannot decide. An example of the second case is an implementation strategy where the compiler produces a program that tries the equivalence of f and g on particular integers until it discovers a difference and produces a 1, and fails to terminate if there are no differences.

Neither of the above alternatives is very appealing. More practical approaches either impose a time limit and report failures for compilations that are too difficult, or restrict the specification language to forms that can be successfully compiled. The second alternative is less attractive because it is difficult to impose syntactic restrictions that will guarantee successful compilation without damaging the abstractness, expressiveness, and simplicity of the specification language. Under the first alternative the designer can initially work with the simplest formulation, and later help the specification compiler over difficult spots by adding annotations or giving interactive advice. The annotations can be removed in a mechanically produced summary view when the specifications are used for communication rather than execution, thus regaining the initial simplicity.

One well established category of specification languages is based on heterogeneous algebras. Some of the specification languages in this category include Larch [19, 20] and Clear [8]. The languages in this class are geared towards specifying abstract data types, and many of them support correctness proofs for programs written using the data types [45]. In algebraic approach data types are specified by giving axioms for the primitive operations of the type in the form of conditional equations. By adding restrictions on the form of the axioms, algebraic specifications can be made executable [12, 17].

One set of restrictions that will suffice to make a set of algebraic specifications executable is the following:

(1) The axioms must be orientable so that the right hand side of each equation is strictly less than the left hand side with respect to some well founded ordering on symbolic terms.

(2) The oriented axioms must be confluent [22].

(3) The set of axioms must be sufficiently complete [17].

(4) The left hand side of each axiom must contain at least one instance of a constructor operation.

These conditions allow the axioms to be treated as rewrite rules. The first condition ensures that all rewrite sequences terminate, so that each expression has a normal form. The second condition ensures that the result of a rewrite sequence is independent of the order in which the axioms are applied, so that all equivalent expressions have the same normal form. The Knuth-Bendix algorithm can be used to check for confluence and to transform some sets of axioms without the property into equivalent sets with the property [22]. The third condition ensures every non-constructor operation applied to variable-free terms is provably equivalent to a constant of another type with respect to the axioms. An operation is a constructor if its range is the type being defined. Since a constant of another type contains no constructor operations, it must be in normal form, and since the normal form is unique, all rewrite sequences for a non-constructor expression must result in a constant. This ensures that all variable free terms of other types can be evaluated by applying the rewrite rules.

An example of an algebraic specification with these properties is given below.

```
type set[t]
  empty(): set[t]
  add(t, set[t]): set[t]
  in(t, set[t]): boolean
  subset(set[t], set[t]): boolean
  equal(set[t], set[t]): boolean
axioms
  in(x, empty) = false
  in(x, add(y, s)) = equal(x, y) or in(x, s)
  subset(empty, s) = true
  subset(add(x, s1), s2) = in(x, s2) and subset(s1, s2)
  equal(s1, s2) = subset(s1, s2) and subset(s2, s1)
end
```

The free variables in each equation are implicitly universally quantified. Equations in this form are equivalent to recursive definitions of the non-constructor operations, if values of the type are represented as symbolic expressions in terms of the constructor operations. Consequently, writing specifications in the restricted form is much like programming. Sometimes it is necessary to introduce auxiliary operations to define the operations we are really interested in [42]. In the example, it is difficult to define the "equal" operation on sets in terms of the "in" operation without introducing an auxiliary operation such as "subset". If the problem does not require a "subset" operation, then introducing one complicates the specification by adding unnecessary details.

Another approach to specifications is based on logic and the event model. Some languages in this class are MSG 84 [4] and Spec 87 [6, 7]. This approach uses predicate logic to define the responses to an event, where an event consists of the arrival of a message at an interface boundary. The major emphasis of these languages has been on abstractness and expressiveness. Both MSG 84 and Spec 87 have facilities for defining functions, state machines, and iterators as well as abstract data types. Experience in a series of software engineering courses [5] indicates that MSG 84 is useful in practice on software developments of appreciable size (five people teams working twenty weeks). The languages support consistency checking of many kinds, and tools for automating the checking are under investigation.

Spec 87 is more advanced than MSG 84 with respect to expressiveness and simplicity. An example of a Spec 87 fragment defining the equal operation on sets is shown below.

```
MESSAGE equal(s1 s2: set{t})
REPLY (b: boolean)
WHERE b <=> FOR ALL(x: t :: in(x, s1) <=> in(x, s2))
```

This says two sets are equal if they have the same elements. This definition is simpler than the corresponding algebraic definition, since three axioms have been replaced by one and the auxiliary concept "subset" has been eliminated. This axiom cannot be expressed in the conditional equation form used by the algebraic techniques because its prenex normal form contains an existential quantifier, and the conditional equation form admits only universal quantifiers. The definition is not executable as it stands because the bound variable x ranges over a potentially infinite type t, but it is subject to the following meaning-preserving transformations.

```
WHERE b <=> FOR ALL(x: t :: in(x, s1) => in(x, s2))
            & FOR ALL(y: t :: in(y, s2) => in(y, s1))

WHERE b <=> FOR ALL(x: t SUCH THAT in(x, s1) :: in(x, s2))
            & FOR ALL(y: t SUCH THAT in(y, s2) :: in(y, s1))
```

The first transformation expands the equivalence into a conjunction of two implications, and decouples the universal quantifiers on the two conjuncts. The second transformation turns the implications into restricted range quantifications. The resulting specification is executable by enumeration because the bound variables x and y have been restricted to finite sets. Informally, the transformed specification says two sets are equal if all of the elements of the first are contained in the second and vice versa.

The Gist language follows a different approach to specifications [1,24]. Gist is based on an extended entity-relationship model of a global state, and the behavior of a system is viewed as a sequence of states. This choice is motivated by the philosophy that the functions of a proposed system should first be defined in a global model, and should be allocated to particular internal or external subsystems in a later step. Unlike the entity-relationship model common in database work [10], the version of the model used in Gist allows relations with infinitely many tuples. Relationships are treated as predicates and are defined using a first order logic with unbounded quantifiers and modal operators for time references. The behavior of the system can be characterized by state invariants and demons that can trigger state changes when stated conditions are met. The language has facilities for introducing boundaries which can be used for creating black-box descriptions. It also allows global references and imperative statements that can be used to describe mechanisms.

The tools associated with Gist include a paraphraser, which generates English narrative texts from the formal specifications [52]. The paraphraser was originally motivated by the need to support review sessions with customers who could not read the formal notation. The paraphraser has also been found useful for locating faults, because it presents the specifications from a different viewpoint than the formal text. Both a symbolic and a concrete execution tools are under development for subsets of the language. The symbolic execution facility describes sequences of states resulting from a given situation using predicates, while the concrete execution facility works with particular data values.

An approach for generating production quality implementations from specifications is embodied in the Psi project [14, 15, 26], Chi project [16, 50], and the Refine language. This work has concentrated on ways to automatically implement behavior specified in first order logic, and on choosing efficient algorithms and data structures for some common general purpose data types, including sets, sequences, Cartesian products, mappings, and relations. This approach has been influenced by work on implementing the SETL language [49].

These problems have been attacked by assembling sets of rules for transforming logical specifications of behavior into algorithm fragments for realizing the behavior. The work has been done using a wide spectrum language with the capabilities for describing both specifications and programs. Such a language is needed for the approach because the transformations produce intermediate results where logical specifications are mixed with program fragments. A specification becomes executable when it is transformed into a program without any specification fragments. The goal of efficiency is pursued by using performance estimates as a guideline for choosing between data representations and algorithms in cases where more than one transformation is applicable [13]. Work on extending the approach to a wider variety of data types is in progress. An active research direction in this area concerns application of the technology to user-defined abstract data types.

Specification languages are useful for simplifying the conceptual design of large systems and for certifying the correctness of critical properties of such systems. Many people are working on automating aspects of the process of producing working systems from formal specifications. Much progress has been made, and it is reasonable to expect the future work in the area will lead to practical benefits that include higher quality software and more efficient software development. Progress on increasing the size and expressive power of the executable subsets of specification languages is possible and useful results are expected from future work in this direction.

The most powerful specification languages available should be used in the analysis and design of large systems to control conceptual complexity. Such specification languages do not have computable compilation functions, making it unlikely that implementation of large software systems can be completely automated. A more realistic goal is implementation by creating and refining annotations for high level specifications. Many practitioners are currently reluctant to accept formal specification languages because they see extra work: an additional language must be learned and a formal document must be produced that does not contribute directly to the program they have to write. They are reluctant to spend much effort on a document that will be produced and discarded. This will change if a specification can be made automatically executable by adding pragmas containing only irredundant compiler directives, especially if pragmas not needed for the easy but tedious parts of the implementation. Other potential paths to acceptance are automatically producing documentation or automatically generating and evaluating test data by means of tools based on the specifications. Even if the pragmas have the effect of choosing between correctness-preserving transformations, testing will still be needed because the transformations may depend on potentially incorrect assumptions about the actual operating environment.

2. Design Languages

The purpose of a design language is to describe the architecture or internal structure of a software system or component. The architecture of a software system consists of a hierarchically structured set of components. The description of a system architecture involves both a design language and a specification language. The design language is used to define the structure of the hierarchy and to describe the interconnections between the components. The specification language is used to define the interface of each component.

The difference between a design and a program is the difference between a plan and a finished product: a design records the early decisions that determine an implementation strategy, while a program contains all the details necessary to get an efficiently executable system. The primary goal of a design is documentation rather than execution. Designs should describe justifications, assumptions, and conventions as well as algorithms and data structures.

Design languages are used for formulation, communication, analysis, and planning. A concise and powerful notation is important for inventing, recording, and communicating designs as well as specifications. The design language is an important medium of communication between the designers, the managers of the project, the implementors, and the CASE tools supporting implementation. A design language can be analyzed with respect to many different properties, such as correctness, performance, and development cost.

Managers are interested in estimating, planning, and tracking a development project. Each of the software components determines a number of tasks that must be scheduled, such as the design, implementation, and verification of the component. These tasks must all be identified before accurate estimation, planning, and task assignments become possible. The managers and the CASE tools for supporting management functions are concerned with extracting task descriptions from the system design and estimating the effort required to do each task.

A design language should have the following properties.

Expressiveness

The language should allow brief and natural descriptions of implementation strategies and justifications. The most powerful known control and data structuring concepts should be included.

Abstractness

It should be possible to determine the essential properties of algorithms, data structures, and subtasks without going into the low level details.

Incompleteness

The language should support descriptions with a controlled degree of incompleteness. Details that must be filled in later should be sufficiently clear to be locatable by a mechanical procedure.

Correspondence

A design language need not be executable, but it should have an executable subset that can be automatically mapped into the implementation language. The non-executable features should be subject to automatic transformations into the implementation language if augmented by pragmas explaining how to implement them in each case. There should also be an automatic mapping from the specification language to the design language for generating the interface description part of the design.

A traditional idea is that design languages should be extensible [31]. This idea should be re-examined in the context of a CASE paradigm. It is desirable to incorporate powerful new ideas in control and data structuring as they come along, since new ideas are rare and it is easier to extend the design language than it is to convert to a new programming language. However, since CASE tools depend on the language, it is desirable to limit the frequency of language changes. If a new design language is going to be designed for the CASE environment, it should include the currently known types of constructs for defining program objects, with emphasis on those that are powerful enough to cover open-ended sets of applications. Examples of such mechanisms include user-defined abstract data types, user-defined loop sequencing abstractions, generic modules, multiple inheritance, parallel loops, atomic transactions, nondeterministic wait (for responding to the first observed instance of a set of asynchronous events), and demons (processes activated whenever a specified predicate becomes true). The mechanisms chosen should be orthogonal or nearly so. Including many variations on a theme can increase rather than decrease the designer's intellectual burden. A single more general mechanism should be sought if a language appears to be sprouting a whole family of similar mechanisms with small variations.

Another traditional justification for extensible design languages is supporting application specific constructs while allowing the aspects of the design language common to all applications to be standardized. With the advent of the powerful and flexible mechanisms listed above, application-specific constructs can be supplied by standard libraries of specialized operations, data types, looping constructs, generic modules, and inheritable generalized module fragments without changing the design language. The desire to simplify the language by dropping the constructs that are not needed in a particular application is consistent with this approach, and can be supported by tools providing simplified subset views of the underlying general purpose language. Some subsets of the language can be certified to have special properties. For example, the CASE tools may know that the functional subset of the design language is side-effect free, so that unsynchronized and unprotected concurrent references can be used in the implementation mapping for the subset language. The remaining advantage in changing the language is to allow the use of special syntactic forms familiar in an application domain. Such advantages are cosmetic, and can be provided by preprocessors or structure editors that support multiple concrete syntactic forms for the same abstract syntax without affecting the structure of the language as seen by the CASE tools.

Another consequence of the CASE paradigm is that design languages should be formal. Informal descriptions can be included as comments, which are not interpreted by the tools. Informal descriptions have been used in the past for two main purposes: to support abstraction, and to make it easier to express designs. Abstraction in this context refers to the ability to capture the essential elements of a design without getting into low level details. Some formal ways to achieve this capability depend on predicate logic and on shared community knowledge. For example, a predicate with quantifiers can be used to describe a complex condition on a data structure serving as the test in a conditional or loop statement. State changes can be described either by explicitly introducing and specifying a black-box component for a lower-level component, or by a transition predicate, which specifies the relation between the initial and final states of a transition without describing the details of how to implement the transition. An example of the use of shared community knowledge is a reference to a "sort" function with a pragma "use quicksort". In such a case, "sort" refers to a general class of modules described in a design library, and "quicksort" refers to a specialization of that class with a particular algorithm known by the designer to have the properties needed in a particular context.

Formal notations are used to gain the advantage of automated processing, possibly at the expense of some extra effort in formulation. One approach to design has been to start from natural language descriptions and to transform them into more formal designs. Such an approach is based on the premise that either detailed natural language descriptions of the processes to be performed are already available from the customer, or that the system designer can sketch an implementation strategy in natural language more quickly than in more formal notations. The process of transforming the natural language descriptions into a formal design language can be partially automated. Some of the more interesting tools attempt to identify abstractions by locating repeated phrases in the natural language text and to identify data types and program objects by locating common noun phrases. A detailed description of this approach and references to related work can be found in [3].

The idea that designs should be accompanied by justifications is motivated by the desire to make changes easier when the system must evolve to meet changing requirements. Some justifications are easiest to record as informal comments, but doing so implies checking will be done manually, perhaps at a design review meeting. Examples of some kinds of justifications and conventions that are important to record are preconditions, data invariants, loop invariants, bounding functions, and termination orderings.

Preconditions are assumptions on the inputs to a module that must be met for a limited implementation to produce correct results. Preconditions in the design usually come from the specification, because they are black-box properties. However, sometimes resource constraints motivate the implementors to introduce stronger preconditions in the design than were originally specified. Such stronger preconditions have to be reflected back to the specification [47], and the places in the design where the module is used have to be checked to make sure they respect the stronger precondition, or adjustments must be made in case they do not. Mechanical aid for such checking is desirable.

Data invariants are restrictions on data structures that must be respected by all programs creating or modifying its instances. Data invariants usually apply to the implementation structures for abstract data types, serving as hidden internal properties specified in the design of a type module. Many of the well known data structures for efficiently implementing common data types gain their efficiency from elaborate data invariants that have been crafted to avoid recomputation of various properties of the data structure. The data invariants constitute the assumptions shared by the implementations of all operations of a type. Since they are not local to a single procedure they can be a vehicle for unwanted interactions, especially for types so large that it is not practical for the same person to implement the operations. Bugs caused by procedures damaging invariants are common and are difficult to diagnose based on fault symptoms because they involve interactions between pieces of code that are separated both spatially and in execution time. This justifies expending a fair amount of effort on documentation and checking.

Loop invariants are properties of the state variables of a loop that hold both before and after every execution of the loop body. Many of the more efficient algorithms depend on carefully constructed loop invariants to achieve their efficiency. While loop invariants are local to a single procedure, they should also be documented to avoid inadvertent damage when the code has to be modified due to a requirements change. Loop invariants as well as data invariants are often difficult to reconstruct from the code, so that they should be recorded as they are introduced in the design process. This is especially important for implementations of critical functions whose correctness will be subject to correctness proofs, because there are automatic procedures for constructing the required theorems which will operate without designer interaction if the loop invariants are given along with the desired preconditions and postconditions.

Bounding functions and termination orderings are justifications for believing that the loops and recursions in the program will terminate. A bounding function gives an upper bound on the number of loop iterations still left for given values of the state variables of the loop, or an upper bound on the depth of any remaining recursive calls for given values of the formal parameters of a recursive subprogram. A terminating program will strictly reduce the bounding function after each execution of the loop body or upon each recursive call. Checking the termination of a program becomes easy if the bounding functions are given. The bounding functions are also useful for performance analysis, because they give worst case estimates of the running times. Termination orderings are useful for establishing termination in some cases where bounding functions yielding natural numbers are difficult to construct. The range of a bounding function can be any well founded set. Some well known termination orderings are the lexicographic and multiset orderings on sequences. These orderings can be used to construct sequence-valued bounding functions for loops or recursive functions whose progress is governed by the interaction of several different parameters.

The kinds of justifications described above can be used in the process of formally or informally verifying the correctness of a design with respect to a given specification. Other kinds of justifications include priorities for different design goals, such as "optimize space".

Design languages are used by experienced and highly skilled people to determine the overall system architecture and to make the key design decisions. Traditionally the more mundane decisions have been left up to less experienced and less skillful people. As CASE technology improves, a larger fraction of the software engineering community will be concerned with architectural design, which will become less tedious with mechanical aid, and the routine aspects of programming will be gradually taken over by automated tools. This trend will be driven by demands for larger and more sophisticated computer systems.

3. Prototyping Languages

The purpose of a prototyping language is to support rapid prototyping. Rapid prototyping is a promising approach to evolutionary software design that was proposed in the early 1980's to solve problems with productivity and reliability in software development [54]. More specifically, prototyping is a method for constructing executable models of software systems rapidly. Such models are known as software prototypes.

Prototyping was distinguished from simulation to emphasize that it should be applied to the early stages of software development and that its goal should be speedily accomplished by an environment containing state of the art software tools. Simulation can be used at any level, including assembly language. The goal of simulation is usually to determine the properties of a specific program or system. Rapid prototyping refers to the activities of constructing software prototypes using CASE design tools at the requirements analysis, functional specification, and architectural design stages of software development. The goal of prototyping is to design, tailor, define, test, document and implement a system (Fig. 3). The prototyping life cycle has two stages, prototyping and system generation. In the prototyping stage, a prototype version of the system is designed and repeatedly tested and modified until the customer is satisfied with it. In the system generation stage, the prototype is used to define and document the architecture of the intended system. The system is implemented by filling in missing details and reworking key modules as needed to achieve adequate performance. Prototyping is most useful for systems that are difficult to built directly, quickly, and correctly, such as software systems with hard real-time constraints and systems large enough to require

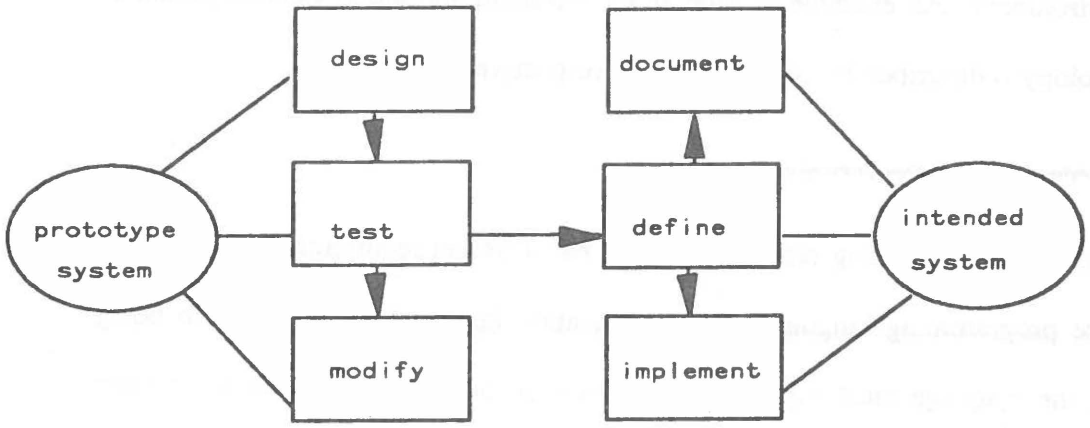

Fig. 3 The Use of Prototypes for Software Development

multiple man-years of design effort.

As shown in Fig. 2, the prototyping process integrates the early stages of the traditional life cycle (Fig. 1) and the evolution stage into the prototyping cycle, which tests the evolving prototype system through execution. The programming level details of the system can be completed after the analyst and customer are satisfied with the behavior of the prototype. The capability for rapid prototyping can best be realized in the context of a high level prototyping language. A prototyping language should have the properties of both a specification language and a design language. The algorithmic level characterizing most current programming languages is not appropriate for supporting rapid prototyping because too many details must be specified. A high level view aids the prototype developer to cope with the complexity of typical software systems, and supports more effective computer-aided systems e.g. reasoning from a design data base or retrieving reusable software components. A prototyping language containing constructs for expressing descriptions of specifications and designs is crucial as well as an automated support environment. An example of such an environment and the associated prototyping methodology is described in [38] and [35, 39] respectively.

3.1. Requirements for Prototyping Languages

A language for supporting rapid prototyping has different requirements from a general purpose programming language or a specification language. In addition to being executable, the language must support the specification of requirements for the system and functional descriptions for the component modules. Since rapid prototyping involves many design modifications, the language must make it easy for the system designer to create a prototype with a high degree of module independence [51], and to preserve its good modularity properties across many modifications. The prototyping language has to be sufficiently easy to read to serve as design documentation, and also has to be formal enough for mechanical processing in the rapid prototyping environment.

The design of a prototyping language should be motivated by the reasons mentioned above and by the requirements listed below:

(1) A prototyping language should be executable, so that the customer can observe the operation of the prototype.

(2) A prototyping language should be simple and easy to use. The language should be based on a simple computational model and should be integrated with a computer-aided prototyping method. The language should support a good designer interface with graphical summary views.

(3) A prototyping language should support hierarchically structured prototypes, to simplify prototyping of large and complex systems. The descriptions at all levels of a prototype should be uniform. The underlying computational model should limit and expose interactions between modules to encourage good decompositions. The language should harmoniously support data abstraction, function abstraction, and control abstraction.

(4) A prototyping language should apply at both the specification and design levels to allow the designer to concentrate on designing the prototype without the distraction of transforming one notation into another.

(5) A prototyping language should be suitable for specifying the retrieval of reusable modules from a software base, to avoid creating multiple descriptions of each module.

(6) A prototyping language should support both formal and informal module specification methods, to allow the designer to work in the style most appropriate to the problem.

(7) A prototyping language should contain a set of abstractions suitable for the problem area for which the prototyping language is designed, e.g. timing for real-time and embedded systems.

When looking for a language meeting such a set of requirements, the designer or analyst may find his choices are limited. It is not hard to convince someone that high level abstractions and brief and powerful language structures are needed to simplify the design at a conceptual level. Many requirements specification and conceptual modeling languages are at a suitable high level, but unfortunately most of them are not executable. Many of the existing programming languages are too inflexible and too difficult to use. Many kinds of coupling problems between modules of a system are not preventable in a programming language because conventional programming languages are required to execute efficiently on conventional machines. Strong coupling can make a rapid prototyping effort fail because modifications get progressively more difficult and error prone, so conventional programming languages cannot be adequate for prototyping. Consequently the design of a special purpose language for rapid prototyping has to be considered.

A prototyping language must have the characteristics of a good design language, because the structure of a prototype must be understandable and easy to modify. Early design languages [31, 33, 51] were not executable, although more recent work has promise in this direction [9]. Some design languages work at the design and specification levels, but are not executable [2]. Other work on executable specifications [34] has taken the automatic transformation of specifications into running systems as a distant long term goal, and has concentrated on generating run-time checks from the specifications in the short run. These approaches are not sufficiently well developed to produce results applicable to rapid prototyping in the near future.

Many informal versions of data flow diagrams have been used extensively to model the data transformation aspects of software systems. Data flow diagrams are easy to read, revealing the internal structure of a process and the potential parallelism inherent in a design, making dataflow attractive to designers. A language based on dataflow makes it easier for a prototyping environment to provide graphical capabilities for displaying and updating the structure of the prototype. However, data flow diagrams or other informal notations do not provide a unified mechanism to represent the relevant attributes of software systems and are not sufficient to be executable. A more precise model of a dataflow computation has been developed in the context of hardware design [11]. Lucid [53] is a good dataflow based programming language. These models and languages support execution, but not specification and design. These languages are not sufficient for specification, design and prototyping in a CASE environment, since the requirements are more complex.

A language supporting both good modularity and good control is needed to support system decomposition [46]. System decomposition is a central issue in the design of any large system. Good modularity is a key factor for increasing productivity, since it reduces the debugging effort for producing a correct executable system, and improves the understandability, reliability, and maintainability of the developed system. A powerful set of control abstractions are needed for simple glass-box descriptions. These features are especially important in rapid prototyping.

Each method for decomposing a software system is associated with a family of computational models. Two well known system decomposition methods are based on data flow and control flow. The components of a data flow decomposition are independent sequential processes that communicate via buffered data streams, while the components of a control flow decomposition are procedures that are called by and return to a main procedure with a single thread of control. Neither of these decomposition methods offers both good modularity and good control.

Iwamoto et al [23] suggest circumstances in which each of the two kinds of decomposition is preferable and give some restrictions sufficient to guarantee that the computed results are independent of scheduling decisions. However, their system is subject to many confusing restrictions and is not sufficient as a base for a CASE prototyping environment. They use a data flow decomposition in cases where there is a mismatch between the structures of the input data stream and the output data stream of an operator, introducing an intermediate data stream of lower level data elements to resolve the structure clash. They use a control flow decomposition in cases where the data stream forks into several branches and is rejoined, or where the operators on the branches influence each other's results by means of state changes, because in these cases a data flow decomposition will result in computations whose results can depend on the unpredictable behavior of the process scheduler. An example of the first case is a decomposition with a dispatch operator that recognizes several alternative kinds of inputs and routes them to the appropriate special purpose operator. A data flow decomposition for such a structure requires extra sequencing information in the data elements to make sure that the result streams do not get out of order when they are merged, since the relative speeds of independent processes are not predictable under the usual interpretation of dataflow. An example of the second case is a transaction with multiple updates to a shared database, where the final state of the database may depend on the arbitrary order of the updates performed by operators on parallel branches of the dataflow graph.

To avoid the problems with data flow decompositions mentioned above we have developed a new underlying model of computation for PSDL [35, 40], which is based on dataflow and guarantees that the results of a computation do not depend on undetermined properties of the schedulers. Control constraints are combined with the dataflow model to achieve the best modularity with sufficient control information. Dataflow is used to simplify the interactions between modules, eliminating direct external references and communication via side effects. The first problem with data flow decompositions mentioned above does not arise in our model because of a rule in PSDL which says that a composite operator cannot fire again until all of the internal activity associated with the previous firing is complete. This rule provides a kind of mutual exclusion that prevents interference between successive actions by the same operator without preventing concurrent execution of the components of a composite operator. The second problem with data flow decompositions does not arise in PSDL prototypes because there is no implicitly shared mutable data.

3.2. Execution of Prototyping Languages

There are two approaches to making a prototyping language executable, one based on meta-programming and the other on executable specifications. The meta-programming approach views the prototyping language as a means for adapting and interconnecting available software components. The processor for such a language generates the skeleton of an implementation, with empty places for component modules. These empty places can be filled in with reusable components drawn from a software base, with stubs for roughly simulating the expected behavior of a module, or by handcrafted code. The prototyping language PSDL described in section 3.3 uses this approach.

The executable specification approach uses black-box specifications in the executable subset of a specification language for realizing the behavior of a system. For expressive notations, this amounts to implementation by enumeration, since this is one of the few known implementation techniques powerful enough to realize arbitrary computable specifications. Such techniques can produce slow implementations, even with sophisticated approaches such as the one taken in logic programming. The execution mechanism of a logic based programming language such as Prolog is a symbolic version of enumeration. This is more efficient than enumeration by brute force, because each logical step considers a potentially unbounded class of individuals rather than a single individual. However, the number of classes to be considered can still be unbounded and is often very large if the logic program contains only the abstract essence of a specification, without any extra information to help narrow down the search. Since executable specifications run slowly without special implementation guidance, this approach must be strengthened to be applicable to the prototyping of very large systems. One way to strengthen the approach is to add annotations for speeding up execution to the pure specifications. Another way is to combine executable specifications with the first approach, and use them to realize only small and simple components of a larger system in cases where efficient implementations for the components are not already available.

3.3. PSDL: An Example of a Prototyping Language

In this section, the concepts and constructs of the prototyping language PSDL (Prototype System Description Language) [40] are used for explaining and analyzing the design principles for prototyping languages mentioned in sections 3.1 and 3.2. Some other languages that have been used for rapid prototyping include SETL, Prolog, Refine, and Kodiyak [21]. These languages are suited for prototyping because they are capable of executing abstract descriptions of processes.

PSDL supports the prototyping of large systems with hard real-time constraints [43, 44]. The language is based on a simple computational model that is close to the designer's view of real-time systems. The model integrates operator, data, and control abstractions (section 3.3.2), and encourages hierarchical decompositions based on both data flow and control flow. More details are described below.

3.3.1. Computational Model

The PSDL computational model contains operators that communicate via data streams. Each data stream carries values of a fixed abstract data type [32]. Each data stream can also contain values of the built-in type "exception". The operators may be either data driven or periodic. Periodic operators have traditionally been the basis for most real-time system design, while the importance of data driven operators for real-time systems is recognized [41].

To provide a small and portable syntax with a clear semantics it is necessary to have a mathematical model behind the language constructs. Formally the computational model is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the maximum execution time for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an operator and each edge is a data stream. The first three components of the graph are called the enhanced data flow diagram.

An **operator** is either a function or a state machine. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams. The output objects produced when a function fires depend only on the current set of input values. The output values produced when a state machine fires depend only on the current set of input values and the current values of a finite number of internal state variables. Operators of these two types are useful for prototyping real-time systems.

Operators are either atomic or composite. Atomic operators cannot be decomposed in the PSDL computational model. Composite operators have realizations as data and control flow networks of lower level operators. If the output of an operator A is an input to another operator B, then there is an implicit precedence relationship between the two, which says that A must be scheduled to fire before B. A composite operator whose network contains cycles is a state machine. In such a case, one of the data streams in each cycle is designated as the state variable controlling the feedback loop, and an initial value is specified for the state variable. State variables serve to break the circular precedence relationships among the operators which would otherwise be implied by the data flow relationships.

A **data stream** is a communication link connecting exactly two operators, a producer and a consumer. Communication links with more than two ends are realized using copy and merge operators. Each stream carries a sequence of data values. Streams have the pipeline property: if a and b are two data values in data stream Y and the data value a is generated by op-1 before the data value b is generated then it is impossible for a to be delivered to op-2 after b is delivered.

There are two types of data streams - **dataflow** streams and **sampled** streams. A dataflow stream guarantees that none of the data values is lost or replicated, while a sampled stream does not make such a guarantee. A data flow stream can be thought of as a fifo queue, while a sampled stream can be thought of as a cell capable of containing just one value, which is updated whenever the producer generates a new value. Since real-time systems must often operate within a (small) bounded memory, the finite queue length imposes a restriction on the relative execution rates of two operators communicating via a dataflow stream. A sampled stream imposes no such constraint, since it can deliver a value more than once if the consumer demands more values before the producer has provided a new value, and it can discard the previous value if the producer provides a new value before the consumer has used the previous one.

Dataflow streams must be used in cases where each data value represents a unique transaction or request that must be acted on exactly once. For example, the transactions in a system for electronic funds transfer would be transmitted along a dataflow stream. Sampled data streams are often used for simulating continuous streams of information, where only the most recent information is meaningful. For example, an operator that periodically updates a software estimate of the system state based on sensor readings would use a sampled stream. In PSDL the stream type is determined from the activation conditions for the consumer operator, rather than being explicitly declared.

3.3.2. Abstractions

Abstractions are an important means for controlling complexity [5], which is especially important in rapid prototyping because a system must appear to be simple to be built or analyzed quickly. PSDL supports three kinds of abstractions: operator abstractions, data abstractions, and control abstractions.

An **operator abstraction** is either a functional abstraction or a state machine abstraction. Both functional and state machine abstractions are supported by the PSDL constructs for operator abstractions. PSDL operators have two major parts: the SPECIFICATION and the IMPLEMENTATION. The specification part contains attributes describing the form of the interface, the timing characteristics, and both formal and informal descriptions of the observable behavior of the operator. The attributes both specify the operator and form the basis for retrievals from a reusable component library or software base. The size and the content of the set of attributes may vary depending on the specific usage, underlying language, or the type of the modules specified, e.g. GENERIC PARAMETERS, INPUT, OUTPUT, STATES, and EXCEPTIONS.

A PSDL operator corresponds to a state machine abstraction if its specification part contains a STATES declaration, otherwise it corresponds to a functional abstraction. The STATES declaration gives the types of the state variables and their initial values. The state variables of a PSDL state machine are local, in the sense that they can be updated only from inside the machine. This restriction prevents coupling by means of shared state variables, and is one of the features of PSDL that leads to good modularization. It is also important for making the correctness of distributed implementations independent of the number of processors.

The implementation part determines whether the operator is atomic or composite. Atomic operators have a keyword specifying the underlying programming language followed by the name of the implementation module implementing the operator. Composite operators have the attributes COMMUNICATION GRAPH, INTERNAL DATA, CONTROL CONSTRAINTS, and INFORMAL DESCRIPTION.

Data abstractions are an important concept for language design. Data abstractions decouple the behavior of a data type from its representation. This is especially important in prototyping because the behavior of the intended system is only partially realized, capturing only those aspects important for the purposes of the prototype. The behavior of the prototype data is also a partial simulation of the data in the intended system, so that the data representations in the prototype and the intended system are likely to be different. Data abstraction allows the data interfaces to be described independently of the representation of the data, so that the interfaces for the operations on the data can be the same in the prototype and in the intended system. Aspects of the data not included in the prototype will be reflected in extra operations on the type, which appear in the intended system but not in the prototype. It is important to have common interfaces between the prototype and the intended system because it makes comparisons easier during the validation of the intended system, and because it enables the structure of the prototype design to be reused in the intended system where appropriate.

All PSDL data types are immutable, so that there can be no implicit communication via side effects. Both mutable data types and global variables have been excluded from

PSDL to help prevent coupling problems in large prototype systems. If many modules communicate implicitly via a shared data structure or global variable, then it is easy to inadvertently interfere with a module by making an apparently unrelated change to another module. Repairing such faults is too time consuming to be tolerated in a rapid prototyping effort.

The PSDL data types include the immutable subset of the built-in types of the underlying programming language, user-defined abstract types [18], the special types time and exception, and the types that can be built using the immutable type constructors of PSDL. The PSDL type constructors were chosen to provide data modeling facilities with a small set of semantically independent structures [40]. For example, finite sets, sequences, tuples, mappings, and relations correspond to the usual mathematical concepts.

The definition of an abstract data type in PSDL contains two parts: SPECIFICATION and IMPLEMENTATION.

Control abstractions are important for simplifying the design of real-time systems, because much of the complexity of such systems lies in their control and scheduling aspects. The control abstractions of PSDL are represented as enhanced data flow diagrams augmented by a set of control constraints. As a common property of real-time systems, periodic execution is supported explicitly. The order of execution is only partially specified, and is determined from the data flow relations given in the enhanced data flow diagrams, based on the rule that an operator consuming a data value must not start until after the operator producing the data value has completed. This constraint applies only if the operators have the same period or if neither is periodic. If the order of execution for two operators is not determined by this rule, then both can run concurrently if sufficiently many processors are available. Conditional execution is supported by PSDL triggering conditions and conditional outputs.

3.3.3. Control Constraints

The control aspect of a PSDL operator is specified implicitly, via control constraints, rather than giving an explicit control algorithm. There are several aspects to be specified: whether the operator is PERIODIC or SPORADIC, the triggering condition, and output guards. The stream types for the data streams in the enhanced data flow diagram are determined implicitly, based on the triggering conditions.

PSDL supports both **periodic** and **sporadic** operators. Periodic operators are triggered by the scheduler at approximately regular time intervals. The scheduler has some leeway: a periodic operator must be scheduled to complete sometime between the beginning of each period and a deadline, which defaults to the end of the period. Sporadic operators are triggered by the arrival of new data values, possibly at irregular time intervals.

A PSDL operator is periodic if a period has been specified for it and sporadic otherwise. A period can be specified explicitly, or it can be inherited from a higher level of decomposition in a hierarchical prototype.

There are two types of **data triggers** inside PSDL operators.

OPERATOR p TRIGGERED BY ALL x, y, z

OPERATOR q TRIGGERED BY SOME a, b

In the first example the operator p is ready to fire whenever new data values have arrived on all three of the input arcs x, y, and z. This rule is a slightly generalized form of the natural dataflow firing rule [11], since in PSDL a proper subset of the input arcs can determine the triggering condition for an operator, without requiring new data on all input arcs. This kind of data trigger can be used to ensure that the output of the operator is always based on fresh data for all of the inputs in the list, and can be used to synchronize the processing of corresponding input values from several input streams.

In the second example, the operator q fires when any of the inputs a and b gets a new value. This kind of activation condition guarantees that the output of operator q is based on the most recent values of the critical inputs a and b mentioned in the activation condition for q. If q has some other input c, the output of q can be based on old values of c, since q will not be triggered on a new value of c until after a new value for a or b arrives. This kind of trigger can be used to keep software estimates of sensor data up to date.

Every operator must have a period, a data trigger, or both. If a periodic operator has a data trigger, the operator is conditionally executed with the data trigger serving as an input guard.

A **timer** is a special kind of abstract state machine whose behavior is similar to a stopwatch. Timers are used to record the length of time between events, or the length of time the system spends in a given state. This facility is needed to express sophisticated aspects of real-time systems, such as timeouts and minimum refresh rates. The state of a timer can be modeled as a time value and a boolean run switch. The value of the timer increases at a fixed rate reflecting the passage of real-time when the run switch is on, and remains constant when the run switch is off.

There are four primitive operations for interacting with timers: read, start, stop, and reset. The read operation returns the current value of the timer without affecting the run switch. The start operation turns the run switch on without affecting the value of the timer. The stop operation turns the run switch off without affecting the value of the timer. The reset operation turns the run switch off and sets the value of the timer to zero.

Timers are treated specially in PSDL because they provide a nonlocal means of control for hard real-time systems. The PSDL declaration

TIMER t

creates an instance of the generic state machine described above, with the fixed name t. The name of a timer can be used like a PSDL input variable, whose value is the result of the read operation of the timer. The value of a timer can be affected by PSDL control constraints of the forms

START TIMER t,
STOP TIMER t, and
RESET TIMER t.

These control constraints can appear anywhere the name t is visible, with the effect of invoking the start, stop, and reset operations of the abstract timer t.

PSDL supports two kinds of **conditionals:** conditional execution of an operator and conditional transmission of an output. These constructs handle the controlled input and output of an operator.

PSDL operators can have a TRIGGERING CONDITION in addition to or instead of a data trigger for **conditional execution.** Two examples of operators with triggering conditions are shown below.

```
OPERATOR r TRIGGERED BY SOME x, y IF x: NORMAL AND y: critical
OPERATOR s TRIGGERED IF x: critical
```

The first example shows the control constraints of an operator with both a data trigger and a triggering condition. The operator r fires only when one or both of the inputs x and y have fresh values, x is a normal data value, and y is an exceptional data value with the exception name "critical". This example illustrates exception handling in PSDL.

The second example shows the control constraints of an operator s with a triggering condition but no data trigger. In this example s must be a periodic operator with an input x since sporadic operators must have data triggers, and triggering conditions can only depend on timers and locally available data. In this case the value of x is tested periodically to see if it is a "critical" exception, and the operator s is fired if that is the case. Both of these examples illustrate ways of using PSDL operators to serve as exception handlers.

In general, the triggering condition acts as a guard for the operator. If the predicate is satisfied, the operator fires and reads its inputs. If the predicate is not satisfied, the input values are read from the input data streams without firing the operator. If a periodic operator has a data trigger or a triggering condition, then the guard predicate is tested periodically, and if found true, the operator is fired. The guard predicate of an operator can depend only on the input values to the operator and on the values of timers. The predicate can make use of the operators of the abstract data types carried by the input streams, allowing a structure similar to a guarded command, where different operators handle an input depending on some computable properties of the input value.

An example of a control constraint specifying a **conditional output** is shown below.

OPERATOR t OUTPUT z IF 1 < z AND z < max

The example shows an operator with an output guard, which depends on the input value max and the output value z.

In general an output guard acts as if the corresponding unconditional output had been passed through a conditionally executed filtering operator with the same predicate as a triggering condition. The filtering operator passes the input value to the output stream unchanged if the predicate evaluates to TRUE. If the predicate evaluates to FALSE, the filter removes the value from its input stream without affecting its output stream. An output predicate can depend only on the input values to the operator, the output values of the operator, and values of timers.

Output guards are convenient but they do not strictly increase the expressive power of the language, since they can be simulated by adding an explicit filter operator, at the cost of some additional output streams to the original operator since the output guard can depend on the INPUTS of an operator as well as on its outputs.

PSDL **exceptions** are values of a built in abstract data type called exception. This type has operations for creating an exception with a given name, for detecting whether a value is an exception with a given name, and for detecting whether a value is normal (i.e. belongs to some data type other than exception). PSDL provides a shorthand syntax for the latter two operations, as illustrated in the following example of a PSDL predicate

x: overflow AND y: NORMAL

which is true if the input value x is the exception value with the name "overflow" and the input value y is normal, as indicated by the PSDL keyword "NORMAL".

Values of type exception can be transmitted along data streams just like values of the normal type associated with the stream. Exceptions are encoded as data values in PSDL to decouple the transmission of an exceptional result from the scheduling of the actions for handling the exception, and to provide a programming language independent interface between atomic operators. This makes it possible to use atomic operators realized in several different programming languages in the same PSDL prototype.

Exceptions can be produced and handled in PSDL. For example, the control constraint

OPERATOR f EXCEPTION e IF x > 100

transmits the exception value named e on all output streams of f instead of the values actually computed by f whenever the input value x is greater than 100. Exceptions can be handled by operators with triggering conditions selecting only input values of type exception, as illustrated in a previous example. Exceptions can be suppressed either by a PSDL output guard of the form

OPERATOR g OUTPUT Y IF Y: normal

or a PSDL input guard of the form

OPERATOR h TRIGGERED IF Y: normal.

The data trigger of an operator determines the **stream types** of its input streams by the following rules.

(1) If a stream is listed in an ALL data trigger, then it is a **dataflow stream.**

(2) All streams not constrained by the first rule are **sampled streams.**

These rules are motivated by the fact that an operator must be executable whenever its triggering conditions are satisfied. In particular, values of streams that are not mentioned at all, or are mentioned in SOME data triggers can be demanded at arbitrary times, which is inconsistent with the fact that dataflow streams cannot allow the consumer to read more values than the producer has written. Consequently rule (1) captures the most general situation where dataflow streams make sense.

In the following example, the operator op has the input streams x, y, z and the output stream w.

```
                                 ---------
                      x ----> |         |
input streams   y ----> |    op    | ----> w   output stream
                      z ----> |         |
                                 ---------
```

Under the following control constraint

OPERATOR op TRIGGERED BY ALL x, y

x, y are dataflow streams while z is a sampled stream. Under a different control constraint

OPERATOR op TRIGGERED BY SOME x, y

x, y, z are all sampled streams. In either case, the stream type of w is not affected by the control constraint associated with its producer operator op.

3.3.4. Timing Constraints

Timing constraints are an essential part of specifying real-time systems. The most basic timing constraints are given in the specification part of a PSDL module, and consist of the MAXIMUM EXECUTION TIME, the MAXIMUM RESPONSE TIME, and the

MINIMUM CALLING PERIOD. The maximum execution time is an upper bound on the length of time between the instant when a module begins execution and the instant when it completes. The maximum execution time is a constraint on the implementation of a single module, and does not depend on the context in which the module is used.

The last two constraints are important for sporadic operators. The maximum response time for a sporadic operator is an upper bound on the time between the arrival of a new data value (or set of data values for operators with the natural dataflow firing rule) and the time when the last value is put into the output streams of the operator in response to the arrival of the new data value. The maximum response time for a periodic operator is an upper bound on the time between the beginning of a period and the time when the last value is put into the output streams of that operator during that period. The maximum response time includes potential scheduling delays, while the maximum execution time does not.

The minimum calling period is a constraint on the environment of a sporadic operator, consisting of a lower bound between the arrival of one set of inputs and the arrival of the next set. In a PSDL specification every sporadic operator with a maximum response time constraint must have a corresponding minimum calling period constraint.

3.3.5. Hierarchical Constraints

PSDL operators are defined in a hierarchical structure, which induces some consistency constraints on the language. The most fundamental constraints are concerned with interface consistency. Every input stream of a component of a composite operator must either be an input of the composite or must be produced by a component of the composite. Similarly every output stream of a component operator must also be an output stream of the composite operator if it is consumed by an operator that is not a component of the composite. Every exception produced by the components of a composite must also be produced by the composite unless it has been handled inside the composite. Each input of a composite operator must be an input of at least one of its components, and each output of the composite operator must be an output of at least one of its components. If the consumer of a data stream is a composite operator, then both the composite and all of its components consuming the same data stream induce constraints on the stream type. PSDL timing constraints also impose some consistency requirements between the various levels of a hierarchical design. The maximum execution time and the maximum response time of a subnetwork must be no larger than those of the composite operator realized by the subnetwork. The minimum calling period of a composite must be no larger than the minimum calling period of any of its components.

3.3.6. Execution of PSDL Prototypes

The prototyping language PSDL uses the meta-programming approach for execution (see section 3.2). PSDL prototypes are executable if all required information is supplied, and the software base contains implementations for all atomic operators and types. To simplify the design of the PSDL translator, Ada is used for implementing both the PSDL reusable components in the software base and the PSDL execution support environment [36]. The PSDL execution support system contains a static scheduler, a translator, and a dynamic scheduler. The static scheduler produces a static schedule for the operators with real time constraints. The translator augments the implementations of the atomic operators and types with code realizing the data streams and activation conditions, resulting in a program in the underlying programming language that can be compiled and executed. Execution is under the control of a dynamic scheduler, which schedules the operators without real-time constraints and provides facilities for debugging and gathering statistics. More details can be found in [35].

4. Conclusions

As compiler and hardware technology improves, the distinctions between prototyping languages, specification languages, design languages, and programming languages are getting smaller and may eventually disappear. Programming languages are getting more expressive and more flexible, and are supporting more abstract descriptions of the processes to be carried out, while specification and design languages are getting to have larger executable subsets. A prototyping language must have the capabilities of both a specification and a design language while still remaining executable. In the short run these four kinds of languages will remain distinct to more effectively support different classes of powerful CASE tools. Programming languages will support optimizing compilers whose main objective is to produce efficient implementations. Specification and design languages will support CASE tools for requirements analysis and for proving the correctness of designs and implementations. Prototyping languages will support tools for prototype demonstrations and implementation planning.

Since the completely automatic and totally correct implementation of powerful specification languages is an algorithmically unsolvable problem, research on CASE technology should investigate ways in which people can most effectively guide tools for computer-aided implementation. A promising approach for applying CASE technology to rapid prototyping is augmenting abstract specifications with annotations or pragmas giving hints about ways to implement them. An important problem is finding concepts and notations that can naturally express such information in an abstract and orthogonal way. Abstractness is desired to simplify the problem of guiding the tools by avoiding as many details as possible. Orthogonality is desired to avoid repeating information that is already contained in or implied by the abstract specification. It is desirable to keep the abstract specification separate or easily mechanically separable from the annotations to provide simplified views of large system models.

Progress on automatically generating prototypes or efficient implementations from abstract specifications is going to depend on a knowledge-based approach. The size of the required knowledge bases depends on the range of problems the language is attempting to address. For this reason, the most powerful systems appearing in the near term will be those with narrow application areas, because such tools can be built with smaller knowledge bases. For a general purpose system, the knowledge base will have to include a large fraction of currently available knowledge about classes of efficient algorithms and data structures, along with the restrictions on their use and measures of their performance. This part of the knowledge is known as the software base. Other kinds of knowledge that may turn out to be necessary include knowledge about ways of adapting and combining the structures in the software base, properties of the application domain and properties of the CASE environment.

1. R. Balzer, ‘‘A 15 Year Perspective on Automatic Programming’’, *IEEE Trans. on Software Eng.*, Nov. 1985.
2. F. W. Beichter, O. Herzog and H. Petzsch, ‘‘SLAN-4 A Software Specification and Design Language’’, *IEEE Trans. on Software Eng. SE-10*, 2 (Mar. 1984), 155-162.

3. D. M. Berry, N. Yavne and M. Yavne, ''Applciation of Program Design Language Tools to Abbott's Method of Program Design by Informal Natural Language Descriptions'', *to appear in Journal of Software and Systems.*, .

4. V. Berzins and M. Gray, ''Analysis and Design in MSG.84: Formalizing Functional Specifications'', *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985), 657-670.

5. V. Berzins, M. Gray and D. Naumann, ''Abstraction-Based Software Development'', *Comm. of the ACM 29*, 5 (May 1986), 402-415.

6. V. Berzins and Luqi, ''Specifying Large Software Systems in Spec'', *submitted to IEEE Software*, 1987. Also NPS 52-87-033, Computer Science Department, Naval Postgraduate School.

7. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

8. R. M. Burstall and J. A. Goguen, ''An Informal Introduction to Specifications using Clear'', in *The Correctness Problem in Computer Science*, Springer Verlag, New York, 1981, 185-213.

9. T. Cheatham, J. Townley and G. Holloway, *A System for Program Refinement*, McGraw-Hill, 1984.

10. P. P. Chen, ''The Entity-Relationship Model - Toward a Unified View of Data'', *Trans. Database Systems 1*, 1 (Mar. 1987), 9-36.

11. J. B. Dennis, G. A. Boughton and C. K. C. Leung, ''Building Blocks for Dataflow Prototypes'', in *Proc. Seventh Symposium on Computer Architecture*, La Baule, France, May 1980.

12. J. A. Goguen and J. J. Tardo, ''An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Specifications'', in *Proceedings IEEE Conference*

on Specifications of Reliable Software,, Apr. 1979, 170-189.

13. A. Goldberg, "Technical Issues for Performance Estimation", in *Proc. Second Annual RADC Knowledge-based Assistant Conference*, RADC(COES), Grifiss AFB, NY, 1987.

14. C. Green, "The Design of the Psi Program Synthesis System", in *Proceedings of the Second International Conference on Software Engineering*, 1976.

15. C. Green and D. Barstow, "On Program Synthesis Knowledge", *Artificial Intelligence Journal*, Nov. 1978.

16. C. Green and S. Westfold, "Knowledge-Based Programming Self-Applied", in *Machine Intelligence*, vol. 10 , Wiley, 1982.

17. J. V. Guttag, "The Specification and Application to Programming of Abstract Data Types", CSRG-59, Ph. D. Thesis, University of Toronto, 1975.

18. J. V. Guttag, E. Horowitz and D. R. Musser, "Abstract Data Types and Software Validation", *Comm. of the ACM 21*, 12 (1978), 1048-1064.

19. J. V. Guttag and J. J. Horning, "A Larch Shared Language Handbook", *Science of Computer Programming 6* (1986), 135-157.

20. J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language", *Science of Computer Programming 6* (1986), 103-134.

21. R. Herndon and V. Berzins, "The Realizable Benefits of a Language Prototyping Language", *IEEE Trans. on Software Eng. SE-14*, 6 (June 1988), 803-809.

22. G. Huet, "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *J. ACM 27*, 4 (Oct. 1980), 797-821.

23. K. Iwamoto and O. Shigo, "Unifying Data Flow and Control Flow Based Modularization Techniques", in *Proceedings of the Fall COMPCON Conference*, IEEE, 1981, 271-277.

24. L. Johnson, ''Overview of the Knowledge-Based Specification Assistant'', in *Proc. Second Annual RADC Knowledge-based Assistant Conference*, RADC(COES), Grifiss AFB, NY, 1987.

25. L. Johnson, ''Turning Ideas into Specifications'', in *Proc. Second Annual RADC Knowledge-based Assistant Conference*, RADC(COES), Grifiss AFB, NY, 1987.

26. E. Kant, ''On the Efficient Synthesis of Programs'', *Artificial Intelligence Journal*, 1983.

27. M. Ketabchi and V. Berzins, ''Generalization Per Category: Theory and Application'', *Proc. Int. Conf. on Information Systems*, 1986. also Tech. Rep. 85-29, Computer Science Dept., University of Minnesota.

28. M. Ketabchi and V. Berzins, ''Modeling and Managing CAD Databases'', *IEEE Computer 20*, 2 (Feb. 1987), 93-102.

29. M. Ketabchi and V. Berzins, ''Mathematical Model of Composite Objects and its Application for Organizing Efficient Engineering Data Bases'', *IEEE Transactions on Software Engineering*, Jan 1988.

30. J. C. Leite, ''A Survey on Requirements Analysis'', RTP-070, Department of Information and Computer Science, University of California at Irvine, 1987.

31. R. C. Linger, H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison Wesley, Reading, MA, 1979.

32. B. Liskov and S. Zilles, ''Programming with Abstract Data Types'', *Proc. of the ACM SIGPLAN Notices Conference on Very High Level Languages 9*, 4 (Apr. 1974), 50-59.

33. G. Luckenbaugh, ''The Activity List: A Design Construct for Real-Time Systems'', Master's Thesis, Department of Computer Science, Univ. of Maryland, 1984.

34. D. Luckham and F. W. Henke, ''An Overview of Anna, a Specification Language for Ada'', *IEEE Software 2*, 2 (Mar. 1985), 9-22.

35. Luqi, ''Rapid Prototyping for Large Software System Design'', Ph. D. Thesis, University of Minnesota, 1986.

36. Luqi and M. Ketabchi, ''A Computer Aided Prototyping System'', in *Proceedings of ACM First International Workshop on Computer-Aided Software Engineering*, vol. 2 , Cambridge, Massachusetts, May 1987, 722-731.

37. Luqi, *Normalized Specifications for Identifying Reusable Software*, Proc. of the ACM-IEEE 1987 Fall Joint Computer Conference, Dallas, Texas, October 1987.

38. Luqi and M. Ketabchi, ''A Computer Aided Prototyping System'', *IEEE Software 5*, 2 (March 1988), 66-72.

39. Luqi and V. Berzins, ''Rapidly Prototyping Real-Time Systems'', *IEEE Software*, Sep. 1988, 25-36.

40. Luqi, V. Berzins and R. Yeh, ''A Prototyping Language for Real-Time Software'', *IEEE Trans. on Software Eng.*, October, 1988.

41. L. MacLaren, ''Evolving Toward Ada in Real Time Systems'', *Proc. ACM SIGPLAN Notices Symp. on the Ada Programming Language*, Nov. 1980, 146-155.

42. M. E. Majster, ''Limits of the 'Algebraic' Specification of Abstract Data Types'', *SIGPLAN Notices Notices 12*, 10 (1977), 37-42.

43. A. K. Mok, *The Design of Real-Time Programming Systems Based on Process Models*, IEEE, 1984.

44. A. K. Mok, ''The Decomposition of Real-Time System Requirements into Process Models'', *IEEE Proc. of the 1984 Real Time Systems Symposium*, Dec. 1984, 125-133.

45. D. Musser, "Abstract Data Type Specification in the AFFIRM system", *IEEE Trans. on Software Eng. SE-6*, 1 (June 1980), 24-31.

46. D. Parnas, "On the Criteria to be Used in Decomposing a System into Modules", *Comm. of the ACM 15*, 12 (Dec. 1972), 1053-1058.

47. D. L. Parnas and P. C. Clements, "A Rational Design Process: How and Why to Fake It", *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 251-257.

48. H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw Hill, 1967.

49. E. Schonberg, J. Schwartz and M. Sharir, "An Automatic Technique for the Selection of Data Representations in SETL Programs", *Trans. Prog. Lang and Systems 3*, 2 (Apr. 1981), 126-143.

50. D. Smith, G. Kotik and S. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Trans. on Software Eng.*, Nov. 1985.

51. W. Stevens, G. Meyers and L. Constantine, "Structured Design", *IBM Systems Journal 13*, 2 (May 1974), 115-139.

52. W. Swartout, "GIST English Generator", in *Proceeedings of the National Conference on Artificial Intelligence*, AAAI, 1982.

53. W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, New York, NY, 1985.

54. R. T. Yeh, R. Mittermeir, N. Roussopoulos and J. Reed, "A Programming Environment Framework Based on Reusability", *Proc. Int. Conf. on Data Engineering*, Apr. 1984, 277-280.

55. R. T. Yeh, N. Roussopoulos and B. Chu, "Management of Reusable Software", *Proc. COMPCON*, Sep. 1984, 311-320.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	2
Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20340	2
Commander Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue NW Washington, D.C. 20390-5290	2
Commander Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374-1662	1
Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	1
Director, Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20363-5100	1
Space and Naval Warfare Systems Command Attn: Dr. Knudsen, Code PD50 Washington, D.C. 20363-5100	1

Ada Joint Program Office OUSDRE(R&AT) The Pentagon Washington, D.C. 230301	1
Naval Sea Systems Command Attn: CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
Office of the Secretary of Defense Attn: CDR Barber The Star Program Washington, D.C. 20301	1
Naval Ocean Systems Center Attn: Linwood Sutton, Code 423 San Diego, CA 92152-5000	1
National Science Foundation Division of Computer and Computation Research Washington, D.C. 20550	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	150