1988

Specification Languages in Computer Aided Software Engineering

Luqi

Naval Postgraduate School

T13

NPS52-88-005

NAVAL POSTGRADUATE SCHOOL

Monterey, California

SPECIFICATION LANGUAGES IN
COMPUTER AIDED SOFTWARE ENGINEERING

LuQi

March 1988

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

K. T. Marshall
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School.

This report was prepared by:

[redacted]

LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[redacted]

for JAMES M. FREMGEN
Acting Dean of Information
and Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-005	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 Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943
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*)
Specification Languages in Computer Aided Software Engineering (U)

12. PERSONAL AUTHOR(S) LuQi

13a. TYPE OF REPORT	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
		March 1988	36

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	

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

One of the most important goals in CASE is to automate the design effort at the early phases of software development. The only way to achieve a high degree of automation is to create mechanically processable documents at the specification level. Thus formal specification level. Thus formal specification languages are the basis of CASE.

This paper examines a number of specification languages, including MSG(MeSsaGe), PSDL (Prototype System Description Language), and Spec(Specification Language). It describes the general principles, language features and basic structures for each of them. The purposes and benefits of using a formal specification language are discussed. The differences among these languages and the areas where each of them apply are explained to help designers choose the right specification language and CASE tools for their applications.

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

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

Specification Languages in Computer Aided Software Engineering

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

One of the most important goals in CASE is to automate the design effort at the early phases of software development. The only way to achieve a high degree of automation is to create mechanically processable documents at the specification level. Thus formal specification languages are the basis of CASE.

This paper examines a number of specification languages, including MSG(MeSsaGe), PSDL(Prototype System Description Language), and Spec(Specification Language). It describes the general principles, language features and basic structures for each of them. The purposes and benefits of using a formal specification language are discussed. The differences among these languages and the areas where each of them apply are explained to help designers choose the right specification language and CASE tools for their applications.

1. Introduction

Requirements analysis, functional specification and architectural design are very important early phases in the software design life cycle. In these phases, the user and designer agree on the requirements for the intended system, its external interfaces and its overall structure. The effort in software system design and implementation is completely dependent on the decisions made and conceptual foundations established at these early phases.

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. 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.

In traditional software development environments CASE is applied only in the module design and programming phases. Automation is mostly based on programming languages. The tools in such environments include compilers, static analyzers, debuggers, and execution profilers. There are many existing pro-

gramming languages available for describing algorithms or writing programs. Traditionally the phases before implementation have been carried out largely by manual processes, and the resulting documents have been expressed in informal notations. The designer has limited choices among existing tools and methods, especially with respect to formal specification languages for aiding the process of firming up requirements and specifying the internal and external interfaces for the designed system.

CASE is a response to problems with traditional development methods. The problems are caused largely by labor intensive tasks at the early stages of development. There are several feasible ways to approach the problem. One approach involves work on executable specification languages and formal verification. 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, concurrency and configuration control for the design data, and information retrieval functions. Another CASE approach introduces the prototyping software life cycle [18]. In this approach, the goals and required behavior of the intended system are negotiated in the context of a computer-aided analysis of the customer's problem, coupled with demonstrations of prototypes.

Formal specification languages are the basis of CASE. The only way to automate the earlier phases of the software development is to create mechanically processable documents at these phases. The reliability and productivity of the process are determined by the specification language used. A good specification language will have maximum expressive capability and a complete formalism for supporting mechanical processing. The systematic use of the formal specification language Spec (see section 3) in the early stages integrated with the use of Ada in the later stages is described with detailed examples in [4].

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 handling only formally defined aspects of notations. 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 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.

Formal languages are therefore key components in a CASE environment. A good CASE language integrates the functions of a specification language, a design language, and a prototyping language [5]. Such a language can serve as the basis for integrating the tools in a CASE development environment. The CASE tools in such an environment depend on each other, and must be integrated together for effectiveness. Such integration depends both on formal languages and emerging technologies for managing special purpose databases, e.g. design databases [22], software bases [14], and engineering databases [10-12, 18]. The formal languages in the environment should support specification, design, prototyping, and programming.

The purpose of a specification language is to define an interface. 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 define the structure of a system. 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 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.

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.

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, validating system structures, and establishing feasibility. 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.

This paper examines a number of specification languages, including MSG (MeSsaGe) [1, 2], PSDL (Prototype System Description Language) [13, 17], Spec (Specification Language) [4]. It describes the general principles, language features and basic structures for each of them. Through the discussion of these languages, the purpose and benefits of using a formal specification language will be illustrated. The differences among these languages and the problems and areas where each of them should apply are explained to help designers choose the right specification language for their own problems and their application area in the new paradigm of computer aided software engineering.

2. Specification Language MSG

A precisely defined specification language such as MSG is very useful for functional specifications, because the language determines what can be said. Early experience with MSG in the research on software development tools [1] as well as use by student teams in a software engineering course [2] indicates that the formal and precise nature of the notation is an aid rather than a barrier to human designers, once the underlying concepts are mastered. A mathematical semantics for the language is essential in order to support computer aided design at the early stages of development.

2.1. General Principles of the Language

The design goals for MSG are the following.

(1) The notation must have a precise and unambiguous semantics.

(2) The notation must be based on a clean and coherent model of computation.

(3) The notation must be sufficiently powerful to express all relevant aspects of system behavior.

(4) The notation must support a convenient set of abstractions for building models of software systems. Abstractions are vital for suppressing detail without sacrificing precision. A well chosen set of building blocks can considerably ease the burden of the analyst.

(5) The notation must not be redundant, to avoid update inconsistencies.

(6) The notation must be modular and support effective searching.

(7) The notation must support partial descriptions, so that it may aid in developing a model and completeness checking.

(8) The notation must be easy to understand and modify. Small conceptual changes should correspond to small changes in the text.

(9) The notation must be relatively easy to learn. Since it will be used by professionals, some specialized training can be required.

(10) The notation must apply to a broad range of applications, so that many different notations are not needed.

(11) The notation must apply to a large part of the life cycle, so that translation from one form to another is minimized. A closely related set of notations can span the functional specification and architectural design stages.

(12) The notation must support mechanical error checking and view extraction. A realistic system cannot be accurately and efficiently specified without a set of computer aided software tools because the specifications get too large for reliable manual processing.

MSG has a precise semantics and a formal definition based on the actor model. Its notation is sufficiently powerful to conveniently express all functional aspects of system behavior and provides a useful set of built-in abstractions and facilities. Modular descriptions are supported, and components in a specification have a fixed order, to make it easier to find things. Concepts defined in other actors must be explicitly imported, providing cross references and disambiguating nonlocal names. Its notation has been expressly designed to allow message types to be defined independently of assertions describing finer details, and to allow the existence of actors or messages to be recorded even if details are not yet available, so that partial descriptions are supported. Such a notation supports mechanical processing at early stages of design, when the specifications are not yet complete.

2.2. Special Features and Applications of the Language

A specification language should support the kinds of abstractions commonly used in describing software systems. The abstractions needed for functional specification are transforms, state machines, data types and iterators. Abstractions in MSG are modeled as actors in the underlying actor model of the language.

MSG is based on the actor model of computation, where actors are independent active elements that interact solely by sending each other messages. All communication between MSG actors modules is done by SENDing messages (also actors) which will be RECEIVEd by some other module in an event. For instance if a message is received that requests an array is to be sorted on some key, the section of an MSG specification that responds to that message will describe the properties of the output array (i.e. keys in ascending order), and not some sorting algorithm like "quicksort".

MSG specifications provide an outline for programmers to follow, and via a series of assertions relates expected output (post conditions attached to SEND, REPLY, UPDATE) for each certain input (preconditions attached to RECEIVE) (i.e. expected output from a procedure call with inputs meeting certain criteria). MSG doesn't supply the algorithms to deliver this output, it just states what conditions the output must meet.

The decomposition of concepts is supported by defined predicates and functions, which can be used in the assertions describing the behavior of an actor. The concepts introduced by definitions are needed to *recognize* when a proposed output value is correct, but they are generally not needed to *construct* those outputs. Consequently the concepts introduced in **define** sections of MSG specifications will probably appear in the user's manual of the proposed system, but probably will not be implemented as running code. Concepts introduced by **define** can depend on other defined concepts, leading to a concept hierarchy orthogonal to the usual action hierarchy.

By the time a MSG specification is finished, an experienced programmer should be able to code any module of the system, using an algorithm of his choice. Previous experience with novices indicates that programmers can learn to read the notation in a few weeks, and can learn to write it in ten weeks.

A number of the tools associated with MSG were developed and experimentally used in both teaching Software Engineering courses and many research projects. For example, an MSG parser/error checker and a database were used to support a number of meaningful consistency checks. The database was linked to the language by means of a bi-directional translator. A syntax-directed editor for MSG and a graphics editor for the associated diagrams were developed.

MSG is used in functional specifications and in architectural design. MSG can also be used in the subset of requirements definition involving defining interfaces to external systems. Such notation will support mechanical processing.

In summary, specification language MSG is a formal tool allowing system designers to abstract specifications, and to communicate design decisions to each other without getting lost in implementation details. In other words, actors are the basic building blocks in MSG for constructing software specifications. A functional specification is a conceptual model of the proposed software system, which presents only those aspects of the system relevant to the users of the system. These building blocks are

sufficient to describe most current software systems that do not involve timing constraints.

2.3. Basic Structures of the Language

Any specification of a software system will consist of a number of modules (or building blocks). Each MSG module defines an actor. MSG offers four module types: MACHINE, TYPE, TRANSFORM, and ITERATOR.

Both machines and types have a **MODEL** as part of their MSG specification. The operations of a machine are described in terms of a conceptual model of its internal state, while the operations of a type are described in terms of a conceptual model of its instances. A conceptual model is a representation of the information content of an object in terms of the built-in types of MSG and the abstract data types defined by the designer.

2.3.1. State Machines

A *state machine* is an actor that exhibits internal memory, in the sense that the events activated by a message depend on earlier messages. Many software systems contain databases or provide commands that allow users to modify internal states in ways that affect the behavior of the system. Examples are inventory control systems, operating systems, and flight simulators. Such systems are naturally modeled as state machines, where user commands or transactions correspond to messages received by the state machine, and the answers produced by the system correspond to the reply messages sent by the state machine. The key word MACHINE is used for state machines in MSG.

An MSG MACHINE is a single instance entity with internal memory, which is abstracted by the MACHINE's MODEL. The MODEL of a MACHINE describes the state of the system or sub-system the MACHINE represents. The basic format of an MSG MACHINE is provided by the following syntax:

```
MACHINE  name     ! Binds a name to a machine for reference
model             ! The conceptual model of the machine
init              ! An assertion describing the initial
                  ! state of the machine, optional
import            ! Allows this machine to use definitions
```

```
                    ! from other actors, optional
export              ! Allows other modules to use any definitions
                    ! explicitly listed, optional
define              ! Macros for commonly repeated definitions,
                    ! and for exportation, optional
BEHAVIOR            ! keyword for beginning of the operations
action              ! The operations
END       name
```

2.3.2. Transforms

A *transform* is an actor that computes a mathematical function, in the sense that the events activated by a message depend only on that message, and not on the contents of earlier messages. A transform has no memory or internal state. There may be more than one response to a single request, in which case the responses are not causally ordered, and the potential for a parallel implementation exists. Responses can either be sent to the actor that originated the request, as happens when a subroutine returns a value, or they can be sent to other actors, as happens in a system with a pipeline structure.

MSG TRANSFORM's are like mathematical functions. They have no internal memory which means the output is only a function of the input parameters. The REPLY depends only on the most recently received message. Examples include: square root, sort, compilers, etc. The basic format of an MSG TRANSFORM is defined by the following syntax.

```
TRANSFORM           name
import              ! optional
export              ! optional
define              ! optional
BEHAVIOR            ! see above machine and
action              ! type for descriptions
END       name
```

2.3.3. Types

MSG TYPE modules are used to describe **abstract data types.** An abstract data type may have several instances, unlike the machine module which has a single instance. For example, an operating system (single instance, machine) can keep track of several processes (multiple instance, type). An abstract data type is one whose instances can be changed, or accessed in any way only via a set of operations associated with that type. There may be many instances of an MSG TYPE, just as many variables of some user defined type may be declared in Pascal. A common example is a stack with the operations: create, is_empty, push, top, and pop. In this example a suitable conceptual model is a sequence containing the elements in the order they were pushed onto the stack. In general the MODEL of a type represents an instance of the abstract type. Calls to the operations for an MSG TYPE are of the format:

<TYPE name>$<operation name>(<parameters>)

Such calls may be used in the assertions describing other operations. It is important to note that the assertions only state how to recognize correct outputs and do not state that the implementation will necessarily have such a call. The basic format of a MSG TYPE is given by the following syntax:

```
TYPE        name    ! Bind name to actor
model               ! Conceptual model of the type
import              ! optional
export              ! optional
define              ! optional
BEHAVIOR            ! see model above
action
END         name
```

2.3.4. Iterator

MSG ITERATORs are control abstractions. They YIELD all elements in a particular instantiation of an abstract data TYPE one at a time, to be used by body of a foreach loop. Iterators are defined in an architectural design, and called only in the algorithms in module designs. For example an ITERATOR called

Depth_First on a binary tree would return each node of the tree in depth first order. This could be used to find the sum of all the elements of an integer tree, or preform a depth first search to see if an element is in the tree. Such an iterator is defined in the following example.

```
TYPE tree
MODEL oneof{leaf: int, node: tuple: {left right: tree}
BEHAVIOR
CASES
  RECEIVE DEPTH_FIRST: tuple {x: tree}
  CASES
    WHERE x = {leaf: a}
    YIELD : sequence{int}
    WHERE YIELD = [a]
  OR ELSE
    WHERE x = {node: n}
    YIELD : sequence{int}
    WHERE YIELD = [!tree$depth_first(n.left), !tree$depth_first(n.right)]
  END CASES
OR ELSE
  RECEIVE ... % other tree operations such as insert and delete
END CASES
END tree
```

3. Specification Language Spec

Spec [4] is a language for writing black-box specifications of interfaces defined in the functional specification and architectural design of software systems. The purpose of the language is to simplify the design and description of large systems by allowing the behavior of an interface to be described without introducing details of the internal structure. This is important because the structural details of most large systems are too complex to be readily understood. Spec provides abstraction mechanisms for simplifying

the description of behavior by hiding structural details.

The critical early stages of software development are dominated by the tasks of building tractable conceptual models of the proposed software and defining its interfaces. These conceptual models consist of abstractions chosen to reduce the complexity of the system. Black-box specifications are essential for realizing the benefits of abstractions in the software development process [2]. An abstraction is useful for controlling complexity only if it has a black-box specification simpler than its implementation. The specification must enable the use of the abstraction without any additional information. A precise notation for black-box specifications is needed for constructing and communicating conceptual models and for supporting computer-aided design. The Spec language provides one such notation.

3.1. General Principles of the Language

Spec is a newly designed specification language based on previous experiences [4]. It closely follows the general design principles of specification languages described with details in [5].

The purpose of a specification language is to describe the interfaces of a software system or component. Its concepts and notations should allow the analyst or designer to formulate an interface for a system or component. It 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. Notations are important for inventing large systems because people are limited in the number of items they can consider at the same time. 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.

Spec was designed to have the following general 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.

Spec is based on the event model of computation, and uses predicate logic for the precise definition of the desired behavior of modules. The most important ideas of this language are modules, messages, events, parametrization, and defined concepts. Spec also has a number of features that become important only for specifying very large systems, such as import/export controls for defined concepts, and view and inheritance mechanisms.

The Spec language uses the event model to define the behavior of black box software modules. The event model has been influenced by the actor model [8]. The main differences from the actor model are the treatment of time and temporal events, and the treatment of multi-event transactions [1]. In the event model, computations are described in terms of modules, messages, and events. A module is a black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another. An event occurs when a message is received by a module at a particular instant of time.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of messages it accepts, along with its response to each kind of message it accepts. Each response consists of the messages sent out by the module in response to the most recent incoming message and the destinations of those messages.

Any module accepts messages one at a time, in a well-defined order that can be observed as a computation proceeds. Message transmission is assumed to be reliable, which means every message sent eventually arrives at its destination. While restrictions on message delay and ordering can be added for particular applications, these are not inherent in the event model. Unless explicitly stated otherwise, messages can have arbitrarily long and unpredictable transmission delays. The order in which messages arrive is not

normally under the control of the designer. The order in which a module accepts messages can be constrained by means of atomic transactions.

Events can also be triggered at absolute times. Such events are called **temporal events**. Formally a temporal event occurs when a module sends a message to itself at a time determined by its local clock. Temporal events are the means by which modules can initiate actions that are not direct responses to external stimuli.

3.2. Special Features and Applications of the Language

The Spec language contains a number of features that are needed mostly for specifying large systems. Some of these features include parameterized modules, defined concepts, and an inheritance mechanism.

The Spec language has an inheritance mechanism which can be used for specifying constraints common to the interfaces of many modules and for view integration. Specifying constraints common to many interfaces is essential for achieving interface consistency in very large systems. For example, the standard properties and operations associated with equality, partial orderings, and total orderings are defined by components in the standard type library. These properties and operations can be inherited by any user-defined type with an "=" or "<" operation, providing an easy way to ensure these operations have their standard meanings. The interface of a system to each class of users can be a separate view of the system, and different interfaces can be specified by different designers or analysts. A total picture of the system is formed by expanding the definition of a module that inherits all of the individual views. The inheritance mechanism and the rules for combining different versions of messages and concepts inherited from multiple parents are described in more detail in [3].

In Spec, atomic transactions are used to restrict the sequence of events in protocols involving chains of events. Such restrictions are sometimes necessary to prevent interference between concurrent processes. Once a module starts an atomic transaction, the order of events at the module is constrained by the transaction until the transaction is completed.

Atomic transactions are defined in terms of sequencing (action ";" action), parallel combination (action action ... action), choice ("IF" action "|" ... "|" action "FI"), and repetition ("DO" action "|" ... "|"

action "OD"). Sequencing implies one instance of the the action on the left must occur before one instance of the action on the right. Parallel combination implies one instance of each action listed must occur in some unspecified order. A choice implies exactly one instance of the actions listed must occur. A repetition implies a sequence of zero or more instances of the actions listed may occur. The actions can be either message names or transaction names. Message names preceded by "EXCEPTION" match only events with the given name and the condition *exception*, while message names without "EXCEPTION" match only events with the given name and the condition *normal*. The transaction names used in actions can have mutually recursive definitions. if more than one atomic transaction is defined, then the first one listed controls the module, and must contain direct or indirect references to the others. The set of legal event sequences for a module forms a context-free language. The definition of a set of atomic transactions can be represented as a recursive transition graph similar to the diagrams commonly used to define the syntax of Pascal..

Spec is a specification language with a broad range of applications. The language is primarily intended for recording black box interface specifications in the early stages of design. The language is most useful in the functional specification and architectural design stages, in which interfaces are proposed and specified. The language has a precise semantics and a simple underlying model. Experience has shown that it is sufficiently powerful to allow the specification of many kinds of software systems, and sufficiently flexible to allow software designers to express their thoughts without forcing them into a restrictive framework. The language is sufficiently formal to support mechanical processing. Some tools for computer-aided design of software that are currently under investigation are syntax-directed editors, consistency checkers, design completion tools, test case generators, and prototype generators.

3.3. Basic Structures of the Language

The basic module types in Spec are the following: FUNCTION, MACHINE, TYPE, DEFINITION, and INSTANCE. Functions, machines, and types are similar to the transforms, machines, and types of MSG. Iterators are replaced by GENERATE clauses in messages, which can appear in any of the first three module types. Definition modules contain only concept definitions. Definition modules are used for defining concepts that cannot be attributed to any single module and are used in many different modules of a system. Instance modules are used for defining instances of generic components that need global names,

such as replicated external systems interacting with a proposed software system.

An example of a specification for a generic square_root function is shown below.

```
FUNCTION square_root {precision: real}
 WHERE precision > 0.0

 MESSAGE (x: real)
  WHEN x >= 0.0
   REPLY (y: real)
   WHERE y >= 0.0 & approximates(y * y, x)
  OTHERWISE
   REPLY EXCEPTION imaginary_square_root

 CONCEPT approximates(r1 r2: real)
  VALUE (b: boolean)
   WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

The example shows the definition of a response with two different cases. One of the cases produces a normal response and the other produces an exception. The concept definition introduces and defines a predicate symbol "approximates" that is used to define the properties of the normal response. An example of a definition for a mutable type is shown below.

```
TYPE queue{t: type}
 -- mutable version
 MODEL (e: sequence)
  -- The front of the queue is at the right end.
 INVARIANT true
  -- Any sequence is a valid model for a queue.

 MESSAGE create
```

```
    -- A newly created empty queue.
   REPLY (q: queue{t})
    WHERE q.e = [ ], new(q)

  MESSAGE enqueue(x: t, q: queue{t})
    -- Add x to the back of the queue.
   TRANSITION q.e = append([x], *q.e)

  MESSAGE dequeue(q: queue{t})
    -- Remove and return the front element of the queue.
   WHEN not_empty(q)
    REPLY (x: t)
     WHERE *q.e = append(q.e, [x])
    TRANSITION FOR SOME (y: t :: *q.e = append(q.e, [y]))
   OTHERWISE REPLY EXCEPTION queue_underflow

  MESSAGE not_empty(q: queue{t})
    -- True if q is not empty.
   REPLY (b: boolean)
    WHERE b <=> (q.e ~= [ ])
END
```

In this example, the "create" operations creates a new instance of the queue type each time it is called, which "enqueue" and "dequeue" change the states of previously existing instances. Expressions preceded by a "*" refer to the previous state. An example of a module with timing constraints is given below.

```
MACHINE ticket_system
 STATE(outstanding: set{ticket_id})
 INVARIANT true
 INITIALLY outstanding = { }
```

```
  MESSAGE ticket(violator: person, ticket_id: integer)
   SEND check_on_payment(violator: person, ticket_id: integer)
   TO ticket_system
    WHERE (30 DAYS) <= DELAY < (31 DAYS)
   TRANSITION outstanding = *outstanding U {ticket_id}

  MESSAGE payment(violator: person, ticket_id: integer)
   TRANSITION outstanding = *outstanding - {ticket_id}

  MESSAGE check_on_payment(violator: person, ticket_id: integer)
   WHEN ticket_id IN outstanding
    SEND letter(s: string) TO violator WHERE warning(s)
    SEND check_on_payment(violator: person, ticket_id: integer)
    TO ticket_system
     WHERE (30 DAYS) <= DELAY < (31 DAYS)
   OTHERWISE -- do nothing

  CONCEPT warning(s: string) VALUE(b: boolean)
   -- True if s is a letter requesting immediate payment of the fine.
END
```

This example illustrates explicitly specified delays for responses. The keyword "DELAY" refers to the time interval between two events, the stimulus and the response. This interval includes both computation time and message transmission time. In this case, both an upper bound and a lower bound on the delay are specified. Time units can range from nanoseconds to weeks.

4. Prototyping Language PSDL

The purpose of a prototyping language is to support rapid prototyping. 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. The goal of prototyping is to design, tailor, define, test, document and implement a system (Fig. 1). 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 multiple man-years of design effort.

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. A high level view can aid the prototype developer to cope with the complexity of typical large software systems, or to support more effective computer-aided systems e.g. reasoning

Fig. 1

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 [18] and [13, 16] respectively.

4.1. General Principles of the Language

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 [23], 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. Consequently a special purpose language for rapid prototyping is needed.

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. PSDL is based on dataflow to make it easier for a prototyping environment to provide graphical capabilities for displaying and updating the structure of the prototype. There are some dataflow based hardware design models or programming languages available. These models and languages support execution, but are not sufficient for specification, design and prototyping in a CASE environment, since the requirements are more complex.

Good modularity and good control are needed to support the central issue of system decomposition [21] 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.

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.

PSDL is based on a unified dataflow and control method, which guarantees that the results of a computation do not depend on undetermined properties of the schedulers. Dataflow is used to simplify the interactions between modules, eliminating direct external references and communication via side effects. Control constraints are combined with the dataflow model to achieve the best modularity with sufficient control information.

4.2. Special Features and Applications of the Language

PSDL supports the prototyping of large systems with hard real-time constraints. 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 and encourages hierarchical decompositions based on both data flow and control flow.

The PSDL computational model contains operators that communicate via data streams. Each data stream carries values of a fixed abstract data type. 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 [16]. 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.

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.

There are two types of data streams - **dataflow** streams and **sampled** streams. A dataflow stream guarantees that none of the data values lost or replicated, while a sampled stream guarantees the most recently generated data value is always available.

Abstractions are an important means for controlling complexity [2], 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. **Data abstractions** are 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. **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 non procedural control constraints. Periodic execution is supported explicitly since it is a common property of real-time systems. Conditional execution is supported by PSDL triggering conditions and conditional outputs.

A set of special constructs in PSDL are designed to support high level real-time specifications, e.g., maximum execution time, data synchronization, minimum calling period, maximum response time, periodic execution, timer control, etc.

Prototyping languages support executable specifications. There are two approaches to making a prototyping language executable one based on meta-programming and the other on executable specifications [5]. The prototyping language PSDL uses the first approach.

PSDL prototypes are executable if all required information is supplied, and the software base contains implementations for all atomic operators and types. Ada is used for implementing both the PSDL reusable components in the software base and the PSDL execution support environment [18]. The PSDL execution support system [15] contains a static scheduler [9, 20], a translator [19], and a dynamic scheduler [7]. 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 an Ada program that can be compiled and executed. Execution is under the control of the dynamic scheduler, which also schedules the operators without real-time constraints.

4.3. Basic Structures of the Language

There are two kinds of building blocks for constructing a PSDL prototype: OPERATOR and TYPE. Basic structures for each of them are given below briefly:

OPERATOR	name
SPECIFICATION	GENERIC_PARAMETER
	INPUT/OUTPUT
	STATES
	EXCEPTION
	TIMING
	REQUIREMENTS_TRACE
IMPLEMENTATION	REUSABLE_COMPONENT
	ENHANCED_DATAFLOW_DIAGRAM

STREAMS

CONTROL_CONSTRAINTS

TYPE	name
SPECIFICATION	OPERATORS_SPECIFICATION
IMPLEMENTATION	OPERATORS_IMPLEMENTATION

The language uses enhanced dataflow mentioned in previous section as a basis combing control structure. Control constraints and timing constraints are the fundamental means to aid the execution.

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.

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 [6], 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.

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.** In general, the triggering condition acts as a guard for the operator. 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.

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.

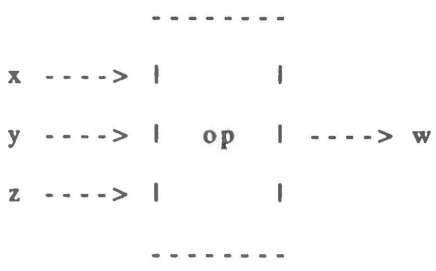
In general an output guard acts as if the corresponding unconditional output had been passed through a conditionally executed filtering operator with the same predicate as a triggering condition.

The 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.**

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



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.

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 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 minimum calling period is a constraint on the environment of a sporadic operator, consisting of a lower bound on the time between the arrival of one set of inputs and the arrival of the next set.

5. Conclusions

Spec has evolved from the specification language MSG [1] and a rapid prototyping language for the design of large real-time systems [17], guided by extensive classroom experience in using formal specifications in multi-person projects [2]. The most important advances over MSG are the integration of time into the underlying model, the development of an inheritance mechanism [3], and improved locality of information. Messages are treated as independent subunits of a specification, and control state considerations are separated from the event-level interfaces of a module by means of explicit definitions for atomic transactions.

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. Specification languages can be used for

formulation, analysis, communication, and retrieval. 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 is augmenting abstract specifications with annotations 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.

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

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

3. V. Berzins and Luqi, *The Semantics of Inheritance in Spec*, Computer Science, Naval Postgraduate School, 1987. NPS 52-87-032.

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

5. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

6. 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.

7. S. Eaton, "An Implementation Design of A Dynamic Scheduler for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

8. C. Hewitt and H. Baker, "Actors and Continuous Functionals", in *Formal Description of Programming Concepts*, North-Holland, New York, 1978, 367-387.

9. D. Janson, "A static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

10. 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.

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

12. 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.

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

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

15. Luqi, "Execution of Real-Time Prototypes", in *ACM First International Workshop on Computer-Aided Software Engineering*, vol. 2 , ACM, Cambridge, Massachusetts, May 1987, 870-884. Technical Report NPS 52-87-012.

16. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems", *to appear in IEEE Software*, 1988.

17. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE TSE*, 1988.

18. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software*, March 1988.

19. C. Moffitt, "Development of a Language Translator for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

20. J. O'Hern, "A Conceptual Design of a Static Scheduler for Hard Real-Time Systems", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

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

22. S. Simmel and V. Berzins, "A Software Management System", in *Proc. First International Workshop on Computer-Aided Software Engineering*, Cambrdge, MA, May 1987, 767-783.

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

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
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
Chief of Naval Research 800 N. Quincy Street Arlington, VA 22217	1