A Proposed Design for a Rapid Prototyping Language

Luqi; Berzins, V.; Kraemer, B.; White, L.

Naval Postgraduate School

T48

NPS52-89-045

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A PROPOSED DESIGN FOR A
RAPID PROTOTYPING LANGUAGE

LUQI
VALDIS BERZINS
BERND KRAEMER
LAURA J. WHITE

MARCH 1989

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin	H. Shull
Superintendent	Provost

This report was prepared in conjunction with research conducted for the National Science Foundation and funded by the Naval Postgraduate School.

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 Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-89-045	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 & ONR Sponsored Navy Direct Funding
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Washington, D. C. 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)
A PROPOSED DESIGN FOR A RAPID PROTOTYPING LANGUAGE (U)

12. PERSONAL AUTHOR(S)
LUQI, VALDIS BERZINS, BERND KRAEMER, LAURA J. WHITE

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Progress	FROM Sept 88 TO Mar 89	1989 March	38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Aided Software Engineering, Rapid Prototyping, Speciification, real-time Software, Embedded Systems, Software Design, Reusability

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
We propose to develop a new rapid prototyping language (RPL) for a wide range of software including real-time, parallel, distributed, and knowledge-based systems. RPL is intended as a tool for aided the development of large Ada systems. Rapid development will be achieved by a combination of a clear and simple language with a suitable computational model, graphical design representations, executable logical specifications, reusable software components, automated code generation, and specialized scheduling algorithms.

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

UNCLASSIFIED

[illegible]

A PROPOSED DESIGN FOR A RAPID PROTOTYPING LANGUAGE

Luqi
Valdis Berzins
Bernd Kraemer
Laura White

Rapid Prototyping Research Group
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

May 1989

ABSTRACT

We propose to develop a new rapid prototyping language (RPL) for a wide range of software, including real-time, parallel, distributed, and knowledge-based systems. RPL is intended as a tool for aided the development of large Ada systems. Rapid development will be achieved by a combination of a clear and simple language with a suitable computational model, graphical design representations, executable logical specifications, reusable software components, automated code generation, and specialized scheduling algorithms.

CONTENTS

1. Introduction

We propose a rapid prototyping language (RPL) for a wide range of software, including real-time, parallel, distributed, and knowledge-based systems.

The Naval Postgraduate School provides a unique environment for RPL design with faculty, fully funded graduate students, and military instructors who have been technically trained and are familiar with DoD application areas. The RPL design proposal has the full support of the superintendent and all levels of administration at NPS.

RPL will benefit from much previous work in rapid prototyping, specification languages, and CASE tools, which are described in more than 50 theses, 60 papers and 100 reports produced by our group. Previous work in design and implementation of formal specification languages includes an event model based language SPEC [9], an actor model based language MSG [4], a Petri net based language SEGRAS [31], and a prototyping system description language PSDL [33] and its computer aided prototyping system CAPS [39].

RPL aids the designer of embedded systems and compensates for some of the problems of Ada. The semantic concepts of RPL support specification and design of real-time embedded and knowledge based systems. RPL prototypes can represent software systems executable on parallel and distributed architectures. RPL is the basis for an integrated tool set supporting iterative prototyping of complex software systems [33, 35, 58].

Key features of RPL are its:

- simplicity and clarity via high level abstractions and graphical notations
- integration of designs, specifications, and requirements to support evolution
- expressive executable subset
- support for modular construction and decomposition of large scale designs
- amenability to formal analysis and verification
- representations for hard and soft real time constraints

Key features of the tool set which will accompany RPL are:

- graphical designer interface
- prototyping database with reusable software components
- automatic Ada code generation from RPL designs
- execution and dynamic debugging support
- optimization and transformation to final implementation
- systematic documentation and version control facilities
- formal analysis, testing, proofs of correctness and consistency

An overview of the technical approach for the development of RPL is described in section 2. Section 3 describes the RPL development plan which includes the organization, technical criteria and the development schedule. Section 4 describes the previous related experience of the key personnel who will be closely involved in the design of RPL.

2. Technical Approach

RPL combines second-order logic specifications supporting verification with an augmented dataflow representation for design and interconnection of prototype components. The design graph is augmented with special pre/post conditions to express real-time constraints and adjust component behavior to each application context [44]. Execution is based on automatically generating code for combining reusable software components or simulating component behavior via the executable subset of the specification logic. Real-time constraints are guaranteed by automatically constructed schedules. Verification is based on a proof system for the logic, which provides a simplified treatment of parallelism due to the atomic execution semantics of RPL modules. Iterative modifications of prototypes are supported by localized information in RPL and its computational model, component behavior modification via logical constraints, and facilities of the tool set for code and design reuse, requirements tracing, and static analysis. The logic and the proposed computational model provide the basis for integrating these facilities into a coherent language and tool structure.

The proposed rapid prototyping language RPL aids rapid construction, evaluation, and adaptation of prototype designs via a clear and simple syntax, support for abstractions, reusable software components, and an integrated system of supporting software tools. Our approach to speeding up the process is based on flexible prefabricated software components.

2.1. Basic Semantic Concepts of RPL

A RPL prototype consists of *modules* communicating via *data streams*. Modules and data streams are collectively called *components*. RPL provides specifications, graphical interconnections, constraints for adapting and controlling prototype behavior, and data format definitions.

2.1.1. Prototype Building Blocks

Modules are classified as *functions, state machines,* and *types,* all of which can be generic. Every module has a black-box specification and a realization. Modules can be realized using four different mechanisms:

(1) retrieval and adaptation of reusable software components,

(2) decomposition into more primitive modules,

(3) simulation of specifications, and

(4) custom implementation in a programming language.

Streams represent both local and long haul communication links between modules. Streams participating in cycles represent state components of state machines, and can be used to model persistent data storage. A stream can carry instances of any designated abstract data type. Exceptions and hardware interrupts are represented as instances of a pre-defined data type. Abstract data types can be either *immutable* or *mutable*. Instances of immutable types are constants. Instances of mutable types have internal states, and can be used to represent prototypes with dynamically created components.

2.1.2. RPL Component Specifications

RPL specifications consist of a form and an optional behavior. The form identifies the operations and the information flows, and is required for execution. The behavior specifies the intended meaning of the module and is needed for verification or simulation. Behavioral specifications are expressed in a *typed second order logic* with two distinguished executable subsets: 1st order equational Horn clauses and 2nd order conditional equations. These subsets can be executed via interfaces to existing languages [25, 60].

The second order logic supports description and analysis of generic modules with type and function parameters. Second order quantifiers are used mostly to express and prove general properties of the reusable generic modules in the prototyping database. Designs implementable directly in Ada have static declarations for all instances of generics. Such designs do not include an interpreter for symbolic simulations. This class of designs covers most applications except for dynamically reconfigurable systems [54] and certain AI computations. Correctness proofs use second order quantifiers to adapt the general specifications of the generics to the particular instances used in the design by substituting the actual generic parameters for universally quantified type and function parameters.

An example of the use of second order logic in specifications is shown in Fig. 1. The generic reduction operator has a second order specification, which is used to derive a first order specification for a particular instance of the generic. In the example, d and f are generic type and function parameters. Generic actual parameters are shown in "[ ]". Existing theorem proving systems can be applied to subsets of the logic [12], and the parts of the system that must be verified can be specified in subsets of the logic amenable to available theorem proving tools.

(a) Generic Specifications for Reduce

```
        All d: type,  f: function[(d, d), d], x: d, s: sequence[d]
            reduce[f](insert(x, s)) = f(x, reduce[f](s))
        All d: type,  f: function[(d, d), d], x: d
            reduce[f](insert(x, nil)) = x
```

(b) Instantiated Specification for sum = reduce[plus], d = real

```
        ALL x: real, s: sequence[real] sum(insert(x, s)) = plus(x, sum(s))
        ALL x: real sum(insert(x, nil)) = x
```

Fig. 1 Example of a Second Order Specification

Second order logic has been avoided previously because there are no complete proof systems. Since the standard proof systems for first order logic are sound for second order logic, all first order proofs are feasible, and some second order theorems can be established using the same proof systems. The incompleteness of these systems does not affect their practical use in verification because termination and many other program properties of practical significance are undecidable in the general case.

RPL component specifications support: verification, fault location by monitoring preconditions and postconditions during prototype execution, computer-aided retrieval and composition of reusable components [37], simulation of prototype modules before they are implemented, and software evolution [41, 57].

2.1.3. RPL Prototype Architecture

RPL graphical interconnection facilities support large scale prototyping via hierarchical decomposition of prototypes. Decomposition is used to express and verify the structure of a software design, and to gain efficiency by using different mechanisms for implementing different parts of the system. This allows the use of existing efficient software components in conjunction with less efficient simulations of parts without available implementations.

Interconnection structures can be represented using icons and arrows on a graphical display to help designers visualize proposed designs. Such graphical network representations are familiar to software developers and are widely recognized as easy to use. We propose to extend icon and arrow representations with optional generic parameters and constraints. Generic graphs can be used to express interconnections in systems containing many instances of a generic module, as illustrated in Fig. 2. Concise descriptions of complex interconnection patterns such as trees, butterfly networks, hypercubes, etc. can be given graphically via subranges and recursion.

2.1.4. Controlling and Adjusting Prototype Behavior

The behavior of modules and data streams is adjusted to the context in which they are used by constraints. Constraints acting as preconditions are called *guard constraints* and those acting as postconditions are called *completion constraints*.

Module constraints - Guard constraints determine the execution conditions for a module, which can depend on the current absolute time, the length of time the data has been in the data stream, the length of time since the last execution of the module, and properties of data values in input data streams [7]. Applications of guard constraints for modules include representing maximum execution rates and demons. Completion constraints determine when and how the results of a component must be delivered to the rest of the prototype. This mechanism can selectively block, delay, or transform outputs, and can represent timed aborts, maximum response times, safety checks, and exception handling.

Stream Constraints - The constraints associated with data streams determine the properties of the communications protocol. Guard constraints on data streams affect the conditions under which data values can be put into a stream or read from a stream. Applications of guard constraints for streams include representing bandwidth constraints, communications delays, first-in/first-out ordering constraints, and limits on storage capacity. Stream completion constraints can represent the conditions under which data values are removed from a stream. These conditions can depend on read and write operations on streams.

Guaranteed Service and Overload Resolution - RPL supports the concepts of *critical* and *non-critical* components for expressing overload resolution policies, especially in the context of real-time systems. Critical modules must terminate with results that meet their postconditions, while non-critical modules must meet their postconditions if they terminate. Thus critical modules carry a guarantee of service, while non-critical modules do not. Critical modules with time bounds have *hard* real-time constraints: the system must guarantee completion by the deadline. Non-critical modules with time bounds have *soft* real-time constraints: they must be aborted if they cannot be completed by the deadline. Streams are also critical or non-critical. Critical streams must guarantee delivery of data within specified constraints, while non-critical streams must meet constraints only if data can be delivered. Non-critical streams with time bounds must discard data if it cannot be delivered on time. Critical streams can represent *stable storage* in fault tolerant distributed systems [32]. Non-critical components can have *priorities*, which determine which module executions and data transmissions are delayed if there are not sufficient resources to meet all constraints.

2.1.5. User Interface Support

The data format definitions of RPL are used to define text formats and graphical icons for instances of abstract data types. The data definition facilities of RPL are based on attribute grammar technology [45]. Defined text formats are used to represent constant expressions and to define input/output formats for abstract data types. Output formats can only be defined if the type has a distinguished set of primitive constructor operations, e.g. cons and nil in lisp, along with corresponding domain test operations, e.g. listp and null in lisp, and partial inverse operations, e.g. car and cdr in lisp. Instances of the formats defined via this facility can be used for:

(1) entering and displaying text representations for abstract input types of interpreters or compilers for special purpose languages;

(2) defining visual gauges for monitoring or animating prototype execution;

(3) specifying initial values for RPL data streams. This supports initializing special databases, such as knowledge bases. An expert system can be represented by a graph in which the entire knowledge base is passed as a value in a data stream;

(4) specifying actual parameters for generic reusable components. Reusable components can have complex options, such as a generic syntax-directed editor that takes an attribute grammar as a parameter.

Lack of support for compact input/output formats for user defined abstract data types has been a major weakness of previous languages, and has impeded the development of convenient debugging facilities for systems with abstract types.

Guard and completion constraints can be represented in abbreviated forms which can be chosen from menus, so that the designer does not need to write predicates explicitly. RPL supports definitions of abbreviated formats for classes of constraints which make it easy to specify common constraints by menu selection, pointing to graphical representations of modules and streams, and entering attribute values in tables [43].

These formats define mappings from the menus and tables to logical specifications in an executable subset of the logic. RPL provides a standard predefined set of constraint formats, and has an open design which allows users to add formats tailored to particular applications.

2.2. Computational Model

RPL is based on an augmented dataflow model. Each RPL module is conceptually an independent process, and different modules can potentially be executed concurrently on different processors. Modules can interact only via data streams. Data streams can be shared among modules. The behavior of a module consists of a discrete sequence of *firings,* where different firings of a module are independent activities. Each time a module fires, it reads a sequence of data elements from each of its input streams, and then writes a sequence of values into each of its output streams. The lengths of the data sequences on each stream associated with each firing are determined from the conjunction of the specification of the module and its constraints. These sequences have length one unless explicitly specified. The time interval associated with each firing begins at the start of the first read operation from the input data streams, and ends at the completion of the last write operation on the output streams. The read operations associated with a module firing are performed as an atomic action, and the write operations are performed as a separate atomic action.

The conditions under which a module fires are determined by the constraints associated with the module and the number of processors in the target hardware configuration. The number of modules firing at any given point in time must be less than or equal to the number of processors, and is not bounded if the number of processors is not specified. The conditions under which a module can fire are determined by the guard constraints of the module. In the absence of guard constraints, a module can fire whenever the required set of input data values is available. If guard constraints are given, the module can only fire if data values are available and the guard constraints are satisfied. Critical modules whose guard constraints are satisfied are guaranteed to be fired eventually. If the postcondition of a critical module includes a bound on the completion time, then firing of the module must be completed within the given bound. Computations with critical completion constraints that require the number of simultaneous firings to exceed the number of processors are not feasible. If a non-critical module has a postcondition that puts a bound on the completion time, and there are not enough processing resources to complete the firing of the non-critical module, then the specified number of input values are read from each input stream and no output values are produced.

Temporal events in an augmented dataflow computation occur at explicitly defined points of absolute time, which are often periodic. *Data events* in an augmented dataflow computation occur when a set of data values enabling the firing of a module becomes available. Completion constraints can limit the delay between a temporal event or a data event at a module and the completion of the next firing of the module. In the absence of constraints on the firing time, at least one module must start to fire whenever there is a module that can fire and the number of modules in the process of firing is less than the number of processors. A non-critical module can fire only if no critical modules can be fired and no higher priority modules can be fired.

2.3. Relationship to Requirements

The goals for RPL are rapid adaptability, design clarity, verification, and a broad range of applicability. We assess our approach with respect to these goals below.

2.3.1. Concepts and Range of Application

The concepts of RPL support a broad range of applications. The prototyping languages PSDL [36] and SSDL, which are based on similar concepts, have been used to prototype a hyperthermia system [33], a C3I system [42], and an inertial navigation system [22]. Experience shows graphical decompositions to be effective for exploring design alternatives [41]. The specification language Spec [10] which contains a logic similar to the one proposed for RPL, has been used to specify an airline reservation system [9], a fault-tolerant network [11], an inertial navigation system [22], and a software design database [15]. A related model for distributed computing has been used in the SEGRAS system to formally specify parallel and distributed computations such as concurrent merge-sort with restricted computation resources, dynamic reconfiguration, and distributed database management [31].

2.3.2. AI-Related Programming

RPL directly supports rule-based programming via guard constraints and modules [40]. Beyond the RPL facilities for general parallel and distributed computing, AI-related programming is supported via the reusable software components and the tools in the CAPS associated with RPL. The main sources of difficulty in developing an AI-related program are identifying, representing, and organizing the knowledge to be used by the application system. Representation and processing of the knowledge can be supported by suitable modules in the software base [17]. Some examples of the generic functions, state machines, and types which can be supplied by a software base include many varieties of inference engines, truth maintenance systems, constraint networks, inheritance networks, production systems, unification functions, fuzzy and Bayesian inference functions, equation solvers, term rewrite systems, frames, contexts, assertions, indexed assertion databases, and semantic nets. The facilities for prototyping special purpose languages also provide a mechanism to quickly construct and experiment with flexible interpreters, which provide the ability to simulate facilities that are difficult to implement directly in Ada, such as creating new types or functions and modifying task priorities as the system runs.

The data definition facilities of RPL can ease the problems of representing and organizing the knowledge in a knowledge base by supporting the definition of special purpose languages for concisely recording and displaying the knowledge. These facilities can also be useful for tailoring generic software components, since many of the more sophisticated reusable components can have complex generic parameters with lengthy structured external representations, such as the set of productions in a production system.

The tools in the CAPS should eventually include generic tool generation facilities, such as translator generators, syntax-directed editor generators [50], and browser generators, which can be used to create special purpose processors for the external knowledge representations used by knowledge engineers. We expect RPL and CAPS to be used to create some of these tools, in a bootstrapping operation. Some of the more advanced tools for assisting knowledge engineers in organizing a knowledge base should help locate conflicting or overlapping bits of knowledge, construct least common generalizations [2], and perform meaning-preserving simplification transformations.

2.3.3. Parallel and Distributed Systems

The computational model of RPL is naturally parallel, and provides atomic actions as primitives. This simplifies design of parallel programs, since concurrent firings of several RPL modules cannot interfere. Nested atomic transactions are naturally represented by hierarchical RPL decompositions.

The only source of interference between concurrent activities in RPL is contention for shared resources such as processing time and data streams. Contention for processors is addressed by the concepts of critical and non-critical components, and constraints induced by critical real-time constraints are addressed by the CAPS scheduling tools. All other resources are represented as instances of abstract types that travel down data streams.

Mutual exclusion for operations on state machines is ensured in RPL by having only one data value in the stream representing the state of the machine. More complex synchronization constraints are expressed locally as context constraints of modules and streams, providing natural representations for distributed designs with no centralized controllers.

It is impossible for two RPL modules to deadlock while seeking exclusive access to the same set of streams since the computational model ensures all data items required to fire a module are read in a single atomic action. A set of RPL modules can become permanently blocked only if they are waiting for input in a cyclic dependency. Tools for detecting design errors in this category are described in Section 2.3.4. RPL facilities for defining the regular interconnection patterns typical of parallel algorithms are discussed in sections 2.1.3 and 2.3.5.

In distributed computations the communications delays can be large compared to processing times. RPL stream completion constraints can specify transmission delays. Fault tolerant designs for distributed systems can be represented in RPL by using critical streams to indicate which parts of the data in a stream must be kept in stable storage to enable failed nodes to recover from processor failures.

2.3.4. Automatic Checking

RPL supports automated consistency checking and error prevention at various levels. The simplest checks are type consistency and interface consistency checks between the levels of a hierarchical design. Tools for preventing errors by automatically filling in implied details of a design are anticipated, and some of the simpler facilities have been demonstrated in the initial CAPS developed for PSDL [52, 62]. For example, skeleton definitions for interfaces of prototype components are generated automatically based on the information in the interconnection graph one level up, and syntax errors are prevented by a template-driven syntax-directed editor.

Effective debugging is supported on a conceptual level by the display facilities for abstract data types. If specifications are expressed in the executable subset of the logic, then code for monitoring preconditions, postconditions, and invariants can be automatically inserted to instrument the prototype for testing and assistance in fault location. The graphical representations for decompositions also provide a convenient framework for animations which can allow the designer or customer to watch the system run at reduced speeds. This capability is useful for demonstrating the prototype to customers. It is especially helpful for debugging purposes when coupled with a capabilities to run the system backwards as well as forwards and to display projections of the computation that filter out lower level events or events that do not match given preconditions. The computational model supports a simple way to run a computation backwards, by recording timed traces of the data sequences read from each data stream. Visualizing the behavior of a prototype in this way helps detect deadlocks and unsafe situations.

An aspect of consistency checking related to parallel and distributed systems is deadlock detection and prevention. A Petri net representation of the control flow defined by an RPL decomposition can be generated by simple mechanical means. Verification conditions for liveness and safety of elementary Petri nets have been generalized for high level Petri nets in [53]. For a subclass of these nets sufficient and necessary conditions exist which enable automatic checks by inspecting net elements and their adjacent arcs. A prototype of a corresponding analysis tool is available for elementary nets and is under development for high level nets.

Another net-theoretic approach relies on the computation of S- and T-invariants of high level net models [18]. The existence of such invariants is a sufficient condition for safety and a restricted form of liveness. The basic idea of this technique is to construct a matrix that shows the local effect of each transition to each state elements of the net and then solve a linear, homogeneous equational system for this matrix. The solutions of this system can be transformed into logic formulas which describe the desired invariance properties. Bounds on buffer sizes useful for optimization can also be determined.

An aspect of consistency checking related to real-time systems is timing feasibility. Static checking for timing constraints that are not realizable even with an unbounded number of processors can be realized by a tool for checking longest paths at the design level. Corresponding tools at the assembly language level are needed to verify maximum execution times for reusable software components relative to particular compilers and target architectures.

The feasibility of meeting the hard real-time constraints with a limited number of processors is checked by constructing a schedule, which can also be used for prototype execution. The initial CAPS for PSDL does this using a heuristic method for a single processor [24, 49]. An optimal polynomial time algorithm for the one-processor case can be constructed based on the approach of [55]. Constructing optimal schedules for the multiprocessor case is NP hard, and hence can only be done for small designs. A linear time heuristic method with good success rates for finding feasible schedules for multiple processors can be based on [51].

Real-time constraints for expert systems are difficult to supply at a high level. For example, the amount of time required for a single logical inference step depends on the size of the input assertion. Thus bounds on the worst case sizes of the input assertions, and bounds on the number of rules that must be applied to reach a conclusion in the worst case are needed to guarantee responses within hard real-time constraints. Such bounds need not exist at all, since many classes of inference problems that are undecidable in the general case, and they can be very difficult to derive if they do exist. The special cases likely to be tractable are expert systems with fixed sets of rules which are designed to answer fixed sets of questions. Such systems can be analyzed mechanically via exhaustive graph search algorithms, but the analysis can be very time consuming.

2.3.5. Flexibility and Reuse

RPL supports flexibility via component constraints, which make it easy to make small changes to the behavior of a prototype, and via its high level computational model, which provides a clear and simple view of the prototype design, before it has been complicated by optimizations and programming level details.

The associated CAPS supports software flexibility by automatically generating many of the details needed for execution, and by providing automated retrieval of reusable components from the prototype database. Such retrievals are based on the specifications of the required components. This retrieval is made feasible by fast approximate methods to narrow down the search space, using standardized forms [34] of the specifications as search keys and a special database organization [26, 39]. Limited logical inference methods and knowledge-based techniques are used to adapt and combine components that provide partial matches to the required specifications [37].

Another aspect of reuse is generic module interconnection patterns. Nontrivial interconnection patterns can be defined using recursive graphs, as illustrated by the simple pipeline structure shown in Fig. 2. The pair of graphs comprising the recursive definition are shown in (a). Generic actual parameters of generic software modules and generic interconnection patterns are shown in "[ ]". In general, a generic interconnection pattern represents a parameterized family of interconnection graphs of different sizes, which contain many instances of a small number of generic components. A particular design will use an instance of a generic interconnection pattern whose size and prototype component names are given as actual parameters, as shown in (b). This facility is especially useful in defining designs for special purpose parallel algorithms, which often

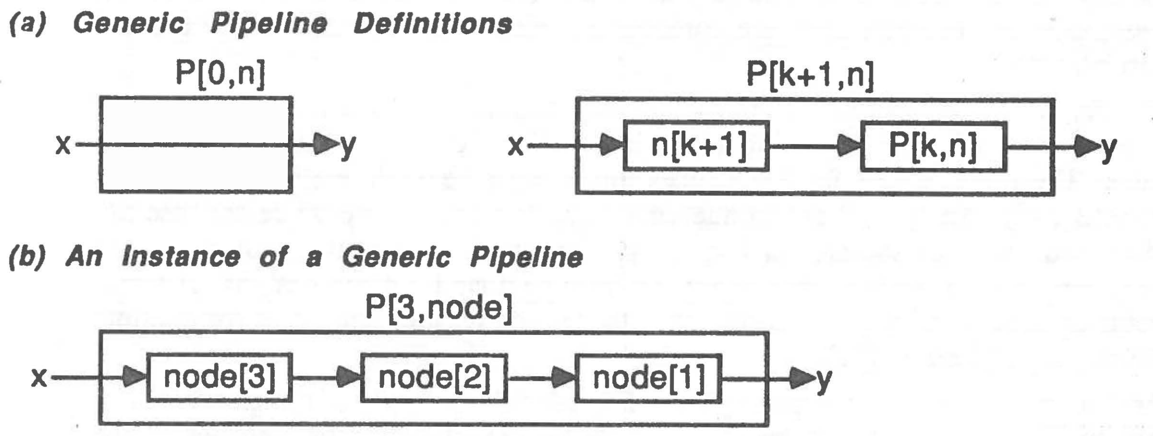

(a) Generic Pipeline Definitions
P[0,n]
x
y
P[k+1,n]
x
n[k+1]
P[k,n]
y
(b) An Instance of a Generic Pipeline
P[3,node]
x
node[3]
node[2]
node[1]
y

Fig. 2 Generic Pipeline Interconnection

consist of large numbers of identical nodes connected in regular patterns.

2.3.6. Formal Methods

The main type of verification associated with RPL is proving that a proposed interconnection of specified components will realize a specified behavior. This is significantly simpler than verifying code in a programming language because component behavior is already expressed in logic.

The theoretical foundations of verifying sequential systems are well known and have been turned into powerful support systems [19, 23]. Theorem proving techniques for mechanically generating proofs in first-order predicate logic will apply to specifications written in that sublanguage. Verification for parallel systems has been poorly understood until recently. The computational model for RPL has been carefully designed to make verification of parallel systems simple. The key property of the semantics of RPL is that firing a module is an *atomic action*. This prevents interference between parallel activities, reducing parallelism to sequential non-determinism. This allows proving safety properties by techniques developed for non-deterministic sequential programs.

Liveness properties in parallel systems depend on fair scheduling assumptions. A type of fair scheduling corresponding to the treatment of critical modules in RPL is *justice*, which ensures that each module will have a chance to fire infinitely many times. A complete[1] proof system based on the atomic action property and the justice assumption is given in[21], using temporal first-order logic. This system can prove both safety and liveness properties, and can easily be adapted to the critical modules of RPL, since these satisfy the justice assumption, provided that the termination of each critical module can be proved in isolation.

Non-critical modules do not satisfy the justice assumption, since they may starve under permanent overload conditions, where all of the available processing resources are needed for critical and higher priority modules. Thus liveness properties cannot be guaranteed for non-critical modules, and the E-rule of [21] does not apply to them. The other rules can be used to prove safety properties of non-critical modules. Streams without explicit constraints have the semantics of ordinary variables, and match the proof rules directly. Streams with constraints must be modeled as additional modules in verification.

The distinction between critical and non-critical modules is essential for the design of real-time systems, and cannot be captured by semantic models for parallel computation based on unmodified versions of the original Unity model [13], which imposes a blanket justice assumption.

2.3.7. Speed of Development

The graphical representations of RPL can make it easier for designers without a high degree of training to use the system. The learning curve is further reduced by an open design which supports alternative graphical icons and language formats to adapt the appearance of designs to the symbols common in particular applications.

[1]Relative to the completeness of the underlying first order logic.

2.3.8. Real-Time Systems

The semantic concepts described in Section 2.1.4 support real-time constraints directly. Specifications for reusable software components include maximum execution times. The context constraints for the components in a decomposition can specify upper and lower bounds on response times and triggering rates, and can directly represent both periodic and data driven computations. The distinction between critical and non-critical modules identifies which activities must have pre-allocated resources sufficient for worst case operating conditions, while priorities and soft real-time constraints (see Section 2.1.4) provide clear and simple mechanisms for formulating system responses to overload conditions without interfering with scheduling considerations for critical modules. The language is designed to support automatic construction of schedules by the CAPS. Sampled data streams (data replaced by write) provide the means for modeling asynchronous communication and dataflow streams (fifo, data removed by read) provide the means for modeling synchronous communication. These mechanisms provide a close match to common design concepts used in embedded real-time systems. Since RPL operations are atomic and independent[2] except for brief stream operations at the beginning and end of each firing, they can be scheduled preemptively by automatic procedures. This avoids the need for the manual effort currently expended on partitioning logical tasks into fragments of sizes suitable for constructing tight schedules.

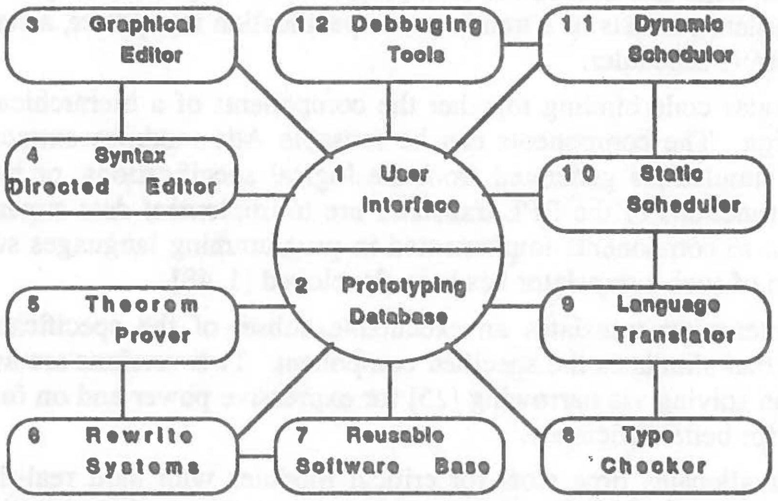
2.3.9. Transformation to Implementation

RPL supports the computer-aided transformation of prototypes into production quality implementations (Ada code for the target architectures) via automatic generation of interconnection code and schedules by the CAPS tools. Optimization transformations are guided by design-level pragmas, which contain high-level optimization advice. For example, an important optimization is in-place updating for streams representing state components with complex structures, such as databases. This optimization modifies a part of the data structure in place, rather than creating a whole new data structure to put into the output stream. The optimization is guided by the designer because it is valid only if the data instance can be contained in at most one data stream at a time. Manual parts of the process include creating Ada implementations for modules that are simulated from the black-box specifications or are implemented in other programming languages in the prototype version of the system. An optimization that can be performed without designer guidance is code merging. Code merging eliminates data stream operations between two modules by combining them into a single module. This optimization can be performed whenever both modules can be scheduled to fire consecutively on the same processor.

2.4. Tool Support

The design of the proposed prototyping language RPL is influenced by the need to support a computer-aided prototyping system (CAPS). This section discusses the aspects of CAPS that affect the design of RPL. An overview of the structure of CAPS is shown in Fig. 3. CAPS will be implemented as an integrated, interactive environment that

[2]Provided that critical modules avoid sharing instances of data types whose instances have internal states.



The designs and the results of feasibility studies of tools 1-12 are given in references [52, (15, 22), 59, 50, 28, 34, 17, 27, (48, 1), (24, 49, 47), 16, 62] respectively.

Fig. 3 CAPS Structure

supports all activities involved in creating, modifying, analyzing, managing, and documenting prototypes, their proofs, and executions. All software objects are stored in an prototyping database which keeps track of their development history and provides a common interface and communication medium to the environment tools.

2.4.1. Designer Interface

The designer interface of CAPS uses a bitmap display and combines a display-oriented text editor with a graphic editor, multiple windows, menus, and mouse input. To help beginners learn RPL by doing, the editor supports template editing, while experienced designers may use their favorite text editor which interacts with a syntax-directed parser to check syntactic correctness of specification fragments and insert them into the database. The graphic editor extends the syntax-directed editing paradigm to the graphical data flow notation of RPL. The database keeps track of both logical constraints and geometrical representations of data flow graphs. The interface provides computer aid for retrieving reusable software components [37] and maintains the link between the prototype design and the requirements. The debugger provides animation facilities for displaying the activity of the prototype in terms of the graphical design representation.

2.4.2. Execution Support

The RPL execution support system reduces prototyping effort by automatically generating Ada code and schedules which enforce timing behavior [38]. This enables RPL to support development of large and embedded Ada programs directly and easily. The RPL execution support system consists of a translator, a specification interpreter, a static scheduler [47], and a dynamic scheduler.

The translator generates code binding together the components of a hierarchically structured prototype design. The components can be reusable Ada modules extracted from the software base, simulations generated from the logical specifications, or hand crafted code. The main functions of the RPL translator are to implement data streams, constraints, and interfaces to components implemented in programming languages such as Ada. An initial version of such a translator has been developed [1, 48].

The specification interpreter translates an executable subset of the specification language into a program that simulates the specified component. Two versions are anticipated, based on equation solving via narrowing [25] for expressive power and on functional programming [60] for better efficiency.

The static scheduler allocates time slots for critical modules with hard real-time constraints. If the allocation succeeds, all modules are guaranteed to meet their deadlines even with worst-case execution times. An initial version of the static scheduler has been developed [24, 49].

The dynamic scheduler invokes modules without hard real-time constraints in the time slots not used by the static scheduler. The dynamic scheduler together with the debugging system allows the designer to control and examine the execution of the prototype [16, 62].

2.4.3. Verification System

For verifying the consistency of a specification and the correctness of a hierarchical decomposition, CAPS will include a theorem prover which provides powerful heuristics for generating first order proofs and supporting formal reasoning [56]. The built-in heuristics guide the designer in searching the proof and defining intermediate proof steps. Efficient evaluation mechanisms such as pattern matching, graph reduction, or term rewriting can be used to simplify the proof of theorems using expressions of the executable sublanguage of RPL. The detailed design and implementation of the theorem prover will be guided by previous work in the verification area [12, 23].

To verify liveness and safety properties, the augmented dataflow model underlying design is mechanically transformed into formulas of a first order temporal logic. The theorem prover can be applied to these temporal logic formulas to construct liveness and safety proofs.

A prototyping database containing verified reusable components will significantly speed up system verification. In such a case only the correctness of the high level interconnections must be proved explicitly.

2.5. Example

An example of a simple design for an aircraft control system is shown in Fig. 4. This example illustrates the use of a dataflow decomposition to describe a system with a

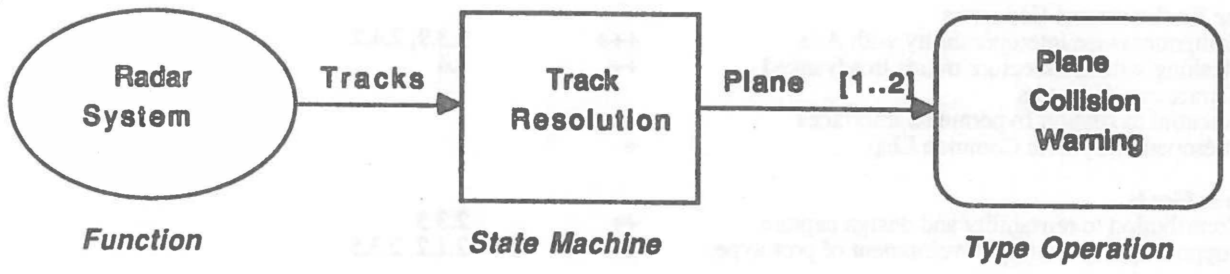


Fig. 4 Simplified Aircraft Control System

variable number of prototype components, and shows that our design representation can handle systems with dynamic configurations, even though the graphical design representations are fixed. There are a variable number of instances of the mutable type "plane", which are kept in the state component of the machine "track_resolution". The "plane" type is a software representation of physical airplanes which keeps track of current courses and positions and handles radio transmissions between the aircraft control system and the physical airplanes. Instances of the type are created and destroyed as the physical airplanes enter and leave the range of the radar system. Two instances of the "plane" type are sent to the node "plane.collision_warning" when the aircraft control system needs to interact with the airplanes to correct a collision course. The node "plane.collision_warning" represents a primitive operation of the type "plane".

2.6. RPL Technical Criteria

ISSUES	PRIORITY	RELEVANT SECTIONS
Executability		
1. Executability of substantial RPL subset	+++	2.1.2
2. Non-executable superset for specification support	++	2.1.2
Major Language Characteristics		
3. Clarity of expression	+++	2.1, 2.3.5, 2.5
4. Conceptual sparseness of design	+++	2.1.3, 2.1.5
5. Amenability to formal analysis	+++	2.1.2, 2.3.6
6. Ability to address programming-in-the-large	+++	2.1.1, 2.1.3
7. Ability to separate concerns of performance/function	++	2.1, 2.3.4, 2.5
8. Conventionality of notation	++	2.1.2, 2.1.3, 2.5
Major Language Features		
9. Support for concurrency	+++	2.3.3
10. Ability to support rule-based programming style	+++	2.3.2
11. Suitability for description of parallel and distributed algorithms and systems	++	2.1.3, 2.1.4, 2.2, 2.3.3
12. Support for time-constrained execution	++	2.1.4
13. Rich type system with user defined types	++	2.1.2

Major Implementation Concerns		
14. Instrumentability, debuggability	+++	2.3.4, 2.4.2
15. Open architecture, modular implementation	++	2.1.3, 2.3.2, 2.3.4, 2.3.5, 2.4
16. Amenability to optimization	++	2.4
17. Separability of concrete and abstract syntax	+	2.1, 2.2
Major Environment Concerns		
18. Componentwise interoperability with Ada	+++	2.3.9, 2.4.2
19. Meshing with architecture trends in advanced software environments	++	2.4
20. Potential to support hypermedia interfaces	++	
21. Interoperability with Common Lisp	+	
Process Goals		
22. Contribution to reusability and design capture	++	2.3.5
23. Support for exploratory development of prototypes	++	2.1.2, 2.3.5

3. RPL Development Issues

3.1. Computer facilities

The Software Engineering Lab at NPS supplies a Sun 3/160 server with a 2.3 GB disk capacity, connected to a network of four more Sun servers for the effort. The project will also have access to the computing facilities of the Computer Science Department. The department has several professional staff members for supporting the hardware and software facilities.

3.2. RPL Development Organization

Faculty at Computer Science Department, Naval Postgraduate School

Prof. Luqi, Principal Investigator
Prof. Valdis Berzins, Associate Professor
Prof. Bernd Kraemer, Research Professor
Prof. Murat Tanik, Research Professor
Prof. M. Nelson, J. Yurchak, R. Griffin, Military Instructors

Students Working on Theses Related to RPL Design

R. Steigerwald, G. Guglielmo, Ph.D. Students
L. White, I. Mostov, F. Palazzo, J. Cervantes, Master's Students

Technical Consultants

Prof. John Guttag, CS Lab, MIT, Specification Languages
Prof. Wayne Richter, Mathematics, University of Minnesota, Logic
Prof. Ben Rosen, CS Dept., University of Minnesota, Optimization
Prof. Amiram Yehudai, CS Dept., University of Maryland
& Tel Aviv University, Israel, Transformations

4. Bibliography

1. C. Altizer, "Implementation of a Language Translator for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.
2. G. Arango, I. Baxter, P. Freeman and C. Pidgeon, "TMM: Software Maintenance by Transformation", *IEEE Software 3*, 3 (May 1986), 27-39.
3. D. Beebe, "Design and Implementation of a Syntax-Directed Editor for the Spec Language", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1989.
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, *The Semantics of Inheritance in Spec*, Computer Science, Naval Postgraduate School, 1987. NPS 52-87-032.
7. V. Berzins and Luqi, "Semantics of a Real-Time Language", in *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society, Huntsville, AL, Dec. 1988, 106-110.
8. V. Berzins, "The Design of Software Interfaces in Spec", in *Proceedings of the International Conference on Computer Languages*, Miami, Oct. 1988, 266-270.
9. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1989.
10. V. Berzins and Luqi, "Specifying Large Software Systems in Spec", *IEEE Software*, to appear 1989. Also NPS 52-87-033, Computer Science Department, Naval Postgraduate School.
11. V. Berzins and R. Kopas, "Specification of a Robust Network", NPS Tech. Rep. 052-89-034, Computer Science Department, Naval Postgraduate School, 1989.
12. R. Boyer and J. Moore, *A Computational Logic Handbook*, Academic Press, 1988.
13. K. Chandy and J. Misra, *Parallel Program Design*, Addison Wesley, Reading, MA, 1988.
14. M. Christ-Neumann, B. Kraemer, H. H. Nieters and H. W. Schmidt, "The GRASPIN Environment on the Lisp Machine - User's Guide", GRASPIN Technical Paper GMD37/1, GMD, Sankt Augustin, Mar 1989.
15. B. Douglas, "A Conceptual Design of a Design Database for the Computer Aided Prototyping System", M.S. Thesis, Computer Science Department, Naval

Postgraduate School, 1989.

16. 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.

17. D. Galik, "A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

18. H. Genrich and K. Lautenbach, "S-Invariance in Predicate-Transition Nets", in *Applications and Theory of Petri Nets*, A. Pagnoni and G. Rozenberg (editor), Springer-Verlag, Berlin-Heidelberg-New York-Tokyo,, 1983,, 98-111 .

19. S. Gerhart, D. Musser, D. Thompson and D. B. etc., "An Overview of Affirm: A Specification and Verification System", in *IFIP 80, Ed. S. Lavington*, North-Holland Publishing Company, 1980.

20. H. Gerlach and W. Sommer, "The Rewrite-Rule-Laboratory of the GRASPIN Environment", GRASPIN Technical Paper KAI30/2, GMD, Sankt Augustin, Feb 1989.

21. R. Gerth and A. Pnueli, "Rooting Unity", in *Proceedings fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, May, 1989, 11-19.

22. H. Guenterberg, "Case Study on Rapid Software Prototyping and Automatic Software Generation: An Inertial Navigation System", M.S. Thesis, Computer Science Department, Naval Postgraduate School, 1989.

23. F. W. Henke, J. S. Crow, R. Lee, J. M. Rushby and R. A. Whitehurst, "EHDM Verification Environment: An Overview", in *Proceedings of the 11th National Computer Security Conference*, (not listed), Baltimore, 1988.

24. 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.

25. B. Jayaraman and G. Gupta, "EqL: The Language and its Implementation", *IEEE Trans. on Software Eng. 15*, 6 (June 1989), 771-779.

26. M. Ketabchi, "On The Management of Computer Aided Design Databases", Ph. D. Thesis, University of Minnesota, 1985.

27. R. Kopas, "The Design and Implementation of a Specification Language Type Checker", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1989.

28. B. Kraemer, "SEGRAS - a Formal Language Combining Petri Nets and Abstract Data Types for Specifying Distributed Systems", in *Proceedings of 9th International Conference on Software Engineering*, March 1987, 116-125.

29. B. Kraemer, "A Sort of Polymorphism in an Algebraic Specification Language", in *Proc. of Joint CWI-GMD-Inria Workshop*, GMD, Karlsruhe, Apr 1989.

30. B. Kraemer and H. W. Schmidt, "Object-Oriented Development of Integrated Programming Environments with ASDL", *IEEE Software*, Jan 1989.

31. B. Kraemer, *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems*, Oldenbourg Verlag,, Muenchen-Wien, 1989.

32. B. Liskov, "Distributed Programming in Argus", *Communications of the ACM 31*, 3 (March 1988), 300-312.

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

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

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

36. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988, 1409-1423.

37. Luqi, "Knowledge Base Support for Rapid Prototyping", *IEEE Expert 3*, 4 (Nov. 1988), 9-18.

38. Luqi and V. Berzins, "Execution of a High Level Real-Time Language", in *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society, Huntsville, AL, Dec. 1988, 69-76.

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

40. Luqi, "Rapid Prototyping Language for Expert Systems", *IEEE Expert*, Summer 1989.

41. Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer*, May 1989.

42. Luqi, "A Software Prototype of the Message Processor in a Navy C3I Station - Modeling and Specification of Hard Real-Time Systems in PSDL", Technical Report NPS 52-89-015, Computer Science Department, Naval Postgraduate School, 1989.

43. Luqi and P. Barnes, "Graphical Support for Reducing Information Overload in Rapid Prototyping", Technical Report NPS 52-89-016, Computer Science Department, Naval Postgraduate School, 1989.

44. Luqi, "Handling Timing Constraints in Rapid Prototyping", in *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, Jan. 1989, 417-424.

45. Luqi, "Automatic Translation of Prototyping Data and Knowledge", *to appear in Journal of Data and Knowledge Engineering*, 1989.

46. Luqi and Y. Lee, "Interactive Control of Prototyping Process", in *Proceedings COMPSAC89*, Orlando, Florida, Sep. 1989.

47. L. Marlowe, "A Scheduler for Critical Timing Constraints", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

48. 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.

49. 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.

50. S. Porter, "Design of a Syntax Directed Editor for PSDL", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

51. K. Ramamritham, J. Stankovic and P. Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems", in *Proc. International Conference on Parallel Processing Systems*, Aug. 1989.

52. H. Raum, "Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

53. H. W. Schmidt, *Specification and Correct Implementation of Non-Sequential Systems Combining Abstract Data Types and Petri Nets*, Oldenbourg Verlag,, Muenchen-Wien, 1989.

54. E. Shibayama, "How to Invent Distributed Implementation Schemes of an Object-Based Concurrent Language - A Transformational Approach", in *Proc. OOPSLA '88*, Sep. 1988, 297-305.

55. J. Sidney and G. Steiner, "Optimal Sequencing by Modular Decomposition: Polynomial Algorithms", *Operations Research 34*, 4 (July 1986), 606-612.

56. D. Smith, "Reasoning by Cases and the Formation of Conditional Programs", Technical Report KES.U.85.4, Kestrel Institute, Palo Alto, CA, 1985.

57. D. Smith, "Top-Down Synthesis of Divide-and-Conquer Algorithms", *Artificial Intelligence 27*, 1 (Feb. 1985), 43-96.

58. W. Swartout and R. Balzer, "On The Inevitable Intertwining of Specification and Implementation", *Comm. of the ACM 25*, 7 (July 1982), 438-440.

59. R. Thorstenson, "A Graphical Editor for the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

60. D. Turner, "Miranda: A Non-Strict Functional Language with Polymorphic Types", in *Functional Programming Languages and Computer Architectures*, vol. 201 , Springer-Verlag, Berlin Heidelberg New York Tokyo, 1985, 1-16. Lecture Notes in Computer Science.

61. J. Weigand, "Design and Implementation of a Pretty Printer for the Functional Specification Language Spec", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1988.

62. M. Wood, "Run-Time Support for Rapid Prototyping", M.S. Thesis, Computer Science Department, Naval Postgraduate School, Dec. 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, Virginia 22217-5000	1
4.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
5.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, D.C. 20301	1
6.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
7.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, D.C. 20301	1
8.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, D.C. 20301	1
9.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, D.C. 20375-5000	1

10.	Navy Ocean System Center Attn. Linwood Sutton, Code 423 San Diego, California 92152-500	1
11.	National Science Foundation Attn. Dr. William Wulf Washington, D.C. 20550	1
12.	National Science Foundation Division of Computer and Computation Research Attn. Tom Keenan Washington, D.C. 20550	1
13.	National Science Foundation Director, PYI Program Attn. Dr. C. Tan Washington, D.C. 20550	1
14.	Office of Naval Research Computer Science Division, Code 1133 Attn. Dr. Van Tilborg 800 N. Quincy Street Arlington, Virginia 22217-5000	1
15.	Office of Naval Research Applied Mathematics and Computer Science, Code 1211 Attn: Dr. James Smith 800 N. Quincy Street Arlington, Virginia 22217-5000	1
16.	New Jersey Institute of Technology Computer Science Department Attn. Dr. Peter Ng Newark, New Jersey 07102	1
17.	Southern Methodist University Computer Science Department Attn. Dr. Murat Tanik Dallas, Texas 75275	1
18.	Editor-in-Chief, IEEE Software Attn. Dr. Ted Lewis Oregon State University Computer Science Department Corvallis, Oregon 97331	1
19.	University of Texas at Austin Computer Science Department Attn. Dr. Al Mok Austin, Texas 78712	1

20.	University of Maryland College of Business Management Tydings Hall, Room 0137 Attn. Dr. Alan Hevner College Park, Maryland 20742	1
21.	University of California at Berkeley Department of Electrical Engineering and Computer Science Computer Science Division Attn. Dr. C.V. Ramamoorthy Berkeley, California 94720	1
22.	University of California at Los Angeles School of Engineering and Applied Science Computer Science Department Attn. Dr. Daniel Berry Los Angeles, California 90024	1
23.	University of Maryland Computer Science Department Attn. Dr. Y. H. Chu College Park, Maryland 20742	1
24.	University of Maryland Computer Science Department Attn. Dr. N. Roussapoulos College Park, Maryland 20742	1
25.	Kestrel Institute Attn. Dr. C. Green 1801 Page Mill Road Palo Alto, California 94304	1
26.	Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 545 Tech Square Attn. Dr. B. Liskov Cambridge, Massachusetts 02139	1
27.	Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 545 Tech Square Attn. Dr. J. Guttag Cambridge, Massachusetts 02139	1
28.	University of Minnesota Computer Science Department 136 Lind Hall 207 Church Street SE Attn. Dr. Fox Minneapolis, Minnesota 55455	1

29.	International Software Systems Inc. 12710 Research Boulevard, Suite 301 Attn. Dr. R. T. Yeh Austin, Texas 78759	1
30.	Software Group, MCC 9430 Research Boulevard Attn. Dr. L. Belady Austin, Texas 78759	1
31.	Carnegie Mellon University Software Engineering Institute Department of Computer Science Attn. Dr. Lui Sha Pittsburgh, Pennsylvania 15260	1
32.	IBM T. J. Watson Research Center Attn. Dr. A. Stoyenko P.O. Box 704 Yorktown Heights, New York 10598	1
33.	The Ohio State University Department of Computer and Information Science Attn. Dr. Ming Liu 2036 Neil Ave Mall Columbus, Ohio 43210-1277	1
34.	University of Illinois Department of Computer Science Attn. Dr. Jane W. S. Liu Urbana Champaign, Illinois 61801	1
35.	University of Massachusetts Department of Computer and Information Science Attn. Dr. John A. Stankovic Amherst, Massachusetts 01003	1
36.	University of Pittsburgh Department of Computer Science Attn. Dr. Alfs Berztiss Pittsburgh, Pennsylvania 15260	1
37.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) Attn. Dr. Jacob Schwartz 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1

38. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

39. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

40. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

41. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

42. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

43. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

44. MCC AI Laboratory 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

45. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

46. University of Maryland 1
Attn. Dr. Basili
Computer Science Department
College Park, MD 20742

47. University of California at San Diego 1
Department of Computer Science
Attn. Dr. William Howden
La Jolla, California 92093

48. University of California at Irvine 1
Department of Computer and Information Science
Attn. Dr. Nancy Levenson
Irvine, California 92717

49. University of California at Irvine 1
Department of Computer and Information Science
Attn. Dr. L. Osterweil
Irvine, California 92717

50. University of Colorado at Boulder 1
Department of Computer Science
Attn. Dr. Lloyd Fosdick
Boulder, Colorado 80309-0430

51. Santa Clara University 1
Department of Electrical Engineering and Computer Science
Attn. Dr. M. Ketabchi
Santa Clara, California 95053

52. Oregon Graduate Center 1
Portland (Beaverton)
Attn. Dr. R. Kieburtz
Portland, Oregon 97005

53. Dr. Wolfgang Halang 1
Bayer AG
Ingenieurbereich Progessleittechnik
D-4047
Dormagen, West Germany

54. Dr. Bernd Kraemer 1
GMD Postfach 1240
Schloss Birlinghaven
D-5205
Sankt Augustin 1, West Germany

55. Dr. Aimram Yuhudai 1.
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

56.	Dr. Robert M. Balzer USC-Information Sciences Institute 4676 Admiralty Way Suite 1001 Marina del Ray, California 90292-6695	1
57.	U.S. Air Force Systems Command Rome Air Development Center RADC/COE Attn. Mr. Samuel A. DiNitto, Jr. Griffis Air Force Base, New York 13441-5700	1
58.	U.S. Air Force Systems Command Rome Air Development Center RADC/COE Attn. Mr. William E. Rzepka Griffis Air Force Base, New York 13441-5700	1
59	**LuQi** **Code 52Lq** **Computer Science Department** **Naval Postgraduate School** **Monterey, CA 93943-5100**	**100**
60	Steve Huseth Honeywell Systems & Research Center 3660 Technology Dr Mels, MN 55418	1
61	Research Administration Code: 012 Naval Postgraduate School Monterey, CA 93940	1