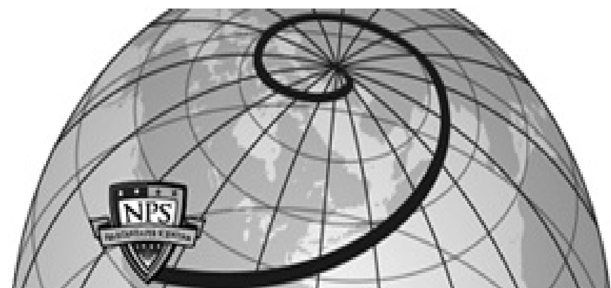
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1989

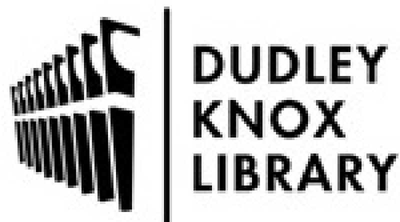
Rapid Prototyping Languages in Computer-Aided Software Engineering

Luqi; Berzins, Valdis

Naval Postgraduate School

Luqi and V. Berzins, "Rapid Prototyping Languages in Computer-Aided Software Engineering'', Technical Report NPS 52-89-021, Computer Science Department, Naval Postgraduate School, 1989.
https://hdl.handle.net/10945/65227

Downloaded from NPS Archive: Calhoun



DUDLEY KNOX LIBRARY

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

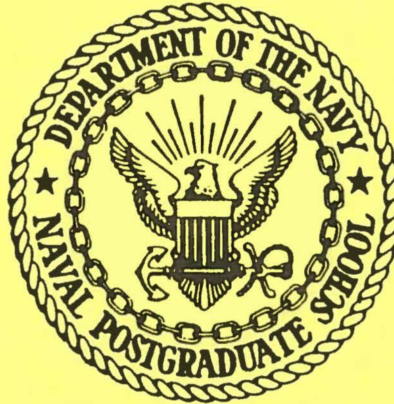
Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

T38

NPS52-89-021

NAVAL POSTGRADUATE SCHOOL

Monterey, California



RAPID PROTOTYPING LANGUAGES IN COMPUTER-AIDED SOFTWARE ENGINEERING

LUQI & VALDIS BERZINS

APRIL1989

Approved for public release; distribution is unlimited.

Prepared for;

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

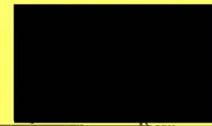
Rear Admiral R. C. Austin
Superintendent

H. Shull
Provost

The work reported herein was supported by the National Science Foundation, the Office of Naval Research and the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.

This report was prepared by:



LUQI
Assistant Professor
of Computer Science

Reviewed by:

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

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) NPS52-89-021	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 NSF CCR-8710737 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)

RAPID PROTOTYPING LANGUAGES IN COMPUTER-AIDED SOFTWARE ENGINEERING (U)

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

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	40

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)

The goal of computer aided rapid prototyping is to automate the design effort at the early phases of software development. The only way to reach this goal is to create mechanically processable and executable documents at the specification level. Rapid prototyping languages form a new category in the computer language family which supports rapid prototyping. Prototyping languages are used to express specification and design of software prototypes in an iterative process of prototype evolution. Prototyping languages combine the functions and benefits of specification, design and programming languages. We describe the requirements, language features, computational models, and general principles for the design of prototyping languages. We compare specification, design and programming languages, conclude that prototyping languages differ from the languages used in traditional software development and support a higher level of automation at the early phases of software development, and indicate the key issues for further progress on prototyping languages.

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

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24.

UNCLASSIFIED

Rapid Prototyping Languages in Computer-Aided Software Engineering

Luqi & Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

The goal of computer aided rapid prototyping is to automate the design effort at the early phases of software development. The only way to reach this goal is to create mechanically processable and executable documents at the specification level. Rapid prototyping languages form a new category in the computer language family which supports rapid prototyping. Prototyping languages are used to express specification and design of software prototypes in an iterative process of prototype evolution. Prototyping languages combine the functions and benefits of specification, design, and programming languages. We describe the requirements, language features, computational models, and general principles for the design of prototyping languages. We compare specification, design and programming languages, conclude that prototyping languages differ from the languages used in traditional software development and support a higher level of automation at the early phases of software development, and indicate the key issues for further progress on prototyping languages.

1. Introduction

The traditional software life cycle consists of a series of phases 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. Traditionally the phases other than implementation have been carried out largely by manual processes, and the resulting documents have been expressed in informal notations. As an alternative, Fig. 1 shows a hierarchy of formal languages which can be used to express these documents in mechanically processable forms. The formal documents from the earlier phases are usually descriptive and cannot be directly translated into efficient implementations. The implementation phase produces programs in a programming language. Testing and evolution are usually done at the programming language level with limited computer aid.

The goal of computer aided software engineering (CASE) is to automate the effort at the early phases [35]. The way to achieve automation is to create mechanically checkable and transformable documents [5]. The computer languages for representing such documents differ from programming languages because of the need to describe things other than algorithms and data structures [3]. These languages should support new life cycles which integrate the functions of all the phases before and after implementation. Prototyping languages are good examples of languages supporting an alternative life cycle, rapid

Traditional Life Cycle		**Rapid Prototyping**	
Phases	**Languages**	**Stages**	**Languages**
Requirements Analysis Functional Specification Architectural Design	Conceptual Modeling Specification Design / Pseudo Code	Rapid Prototyping	Rapid Prototyping
Implementation & Testing	Programming	Code Generation	Programming

Fig. 1 Software Language Hierarchy

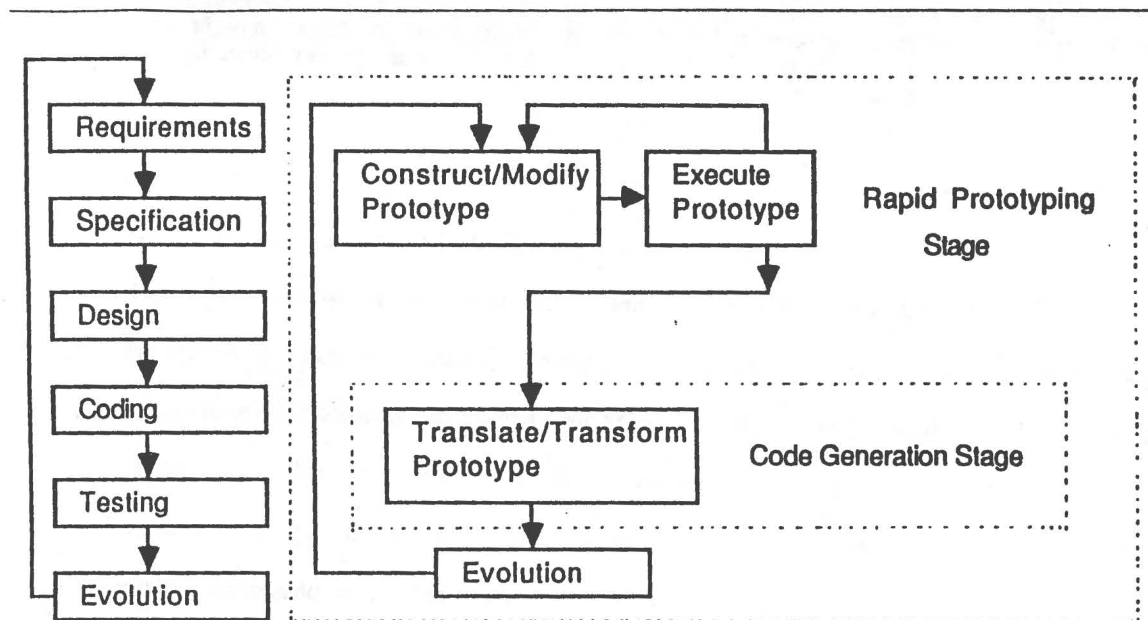


Fig. 2 Traditional Development vs. Rapid Software Prototyping

prototyping [4, 22]. These languages combine specification and design with an execution capability to better support the early phases (Fig. 1).

Rapid software prototyping is for gaining information to guide analysis and design, and supporting the automatic generation of the production code. It consists of two stages: prototyping and code generation. Fig. 2 illustrates the prototyping process and compares it to traditional software development. The prototyping stage firms up software requirements through iterative negotiations between customers and

designers via the examination of executable prototypes [24]. A software prototype is a simplified model of the proposed system. The designer adjusts the requirements and modifies the prototype accordingly based on feedback from the customer until the customer agrees on the requirements. The code generation stage focuses on augmenting the prototype to generate the production code.

Prototyping with computer-aided tools makes a rapid prototyping process possible. CAPS (Computer Aided Prototyping Systems) [21] are the tool sets or environments designed especially for the purpose of rapid prototyping. A special prototyping language with the assistance of a CAPS system should make it easy to specify, construct, demonstrate, understand, explain, and modify a software prototype. As the most critical component, the language provides the basic communication and representation medium for all the tools in a CAPS system. CAPS should be used to prototype large, parallel, distributed, real-time, and knowledge-based systems because the requirements for such systems are difficult to assess, leading to demand for prototyping support in these areas [24]. A prototyping language differs from the languages in the traditional language hierarchy shown in the left half of Fig. 1 since it addresses functions at all of the levels of the hierarchy. Because of its wide range of functions, a prototyping language may not be as application specific at a single level of the hierarchy as a special purpose language designed just for that level.

Conventional compiler technology is not sufficient for the execution of a prototyping language, because one of the goals of a rapid prototyping system is to execute prototype descriptions that do not contain details of algorithms and data structures [36]. Some of the issues that must be faced by an execution support system for a prototyping language are providing missing details and ensuring that real-time constraints are met [13, 23]. This can be done using by combining program transformations 2] and specialized schedulers [11, 18, 20, 29, 31, 33, 34, 39] with a knowledge base containing programming and problem domain knowledge [25].

Prototyping languages were designed based on knowledge and experience from all levels of the hierarchy. The relevant aspects of specification, design, and programming languages are discussed in section 2. Section 3 describes the requirements for a prototyping language, discusses issues in designing a prototyping language, and gives examples of prototyping languages. Section 4 presents conclusions.

2. Formal Languages for Specification, Design, and Programming

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

Formal languages can be used to apply computer-aided analysis and design from the earliest stages of software development. The goals and required behavior of a proposed system are negotiated in the context of a repeated analysis of the customer's problem. Knowledge-based assistants for each phase of development starting with requirements analysis are examples of this approach. This computer-aided process includes completeness and consistency checking, displaying descriptions of the system from various viewpoints, demonstrations of prototypes, concurrency and configuration control for the design data, and information retrieval functions. The tools in such an environment depend on each other, and must be integrated together to effectively support the process. Such integration depends both on formal languages and emerging technologies for managing engineering databases [15].

This section examines the categories of computer languages for specification, design and programming. Many of the existing languages for software development have characteristic properties and restrictions from one or several of these categories. These language categories and the relations among them provide the background and technology for the creation of rapid prototyping languages, which are used to create specifications, express designs, and execute prototypes. We focus on general purpose languages that can span a range of applications, and exclude application specific "fourth generation" languages.

2.1. Specification Languages

A specification is a black-box description of the behavior of a software system, which may interact directly with users or may be part of a larger system. A black-box description defines the interfaces of a software system in terms of the data that crosses its boundary, without reference to the mechanism inside. Such interfaces define the form and meaning of all interactions between the system and its environment.

Specification languages are formal notations for representing specifications. The primary benefits of using a formal specification language are precision and the potential for automation, which lead to better software products [37]. Formal specifications are used for defining, communicating, analyzing, and realizing system interfaces.

2.1.1. What can Specification Languages do?

A specification language provides a set of concepts and notations which help the designer formulate an interface for a system or component. The language influences the designer's thinking and determines which things are easy to express, and which are impossible or impractically difficult. A specification language should help the designer construct simple conceptual models of proposed systems and maintain their conceptual integrity.

The specification language is a medium for communication between the designers, the development teams, and the customers. Written specifications are needed for contract agreements and for internal communication in development organizations. Since people come and go more frequently than large projects are completed, specifications are needed to record the state of a project in a permanent form. The information in the specification is also the basis for customer review, although it is usually necessary to paraphrase the information and to provide summaries and simplified views to effectively communicate with typical customers.

A formal specification language enables analysis of the proposed interface with respect to many different kinds of properties. Examples of such properties include type consistency, freedom from deadlock for multistep protocols, satisfiability of the constraints on the required response for each possible input, coverage of all possible input values, uniqueness of outputs, and consistency with a proposed design. None of these semantic properties can be determined without a precise specification. Software systems can also

be realized with the aid of formal specifications, which can be used for retrieving and adapting existing reusable components or as a starting point for a transformation system.

2.1.2. Should Specifications be Executable?

There has been increasing interest in executable specification languages, motivated by automated *prototyping* for validating requirements and specifications, and automated *implementation of production* quality software. The first of these applications of executable specifications relaxes performance constraints while the second does not. Both of these processes are not computable in the general case, if the specification language is strong enough to be interesting. The practical impact of an "executable" specification language can be judged by considering the expressiveness of the entire language, the expressiveness of its executable subset, and the relative difficulty of transforming simple but non-executable specifications into executable equivalents.

There is a tradeoff between convenience of expression and a uniform guarantee of executability for a specification language. Many specification languages gain their power by including non-effective operators such as unrestricted logical quantifiers, and the presence of such operators is one of the things that distinguishes specification languages from programming languages. Non-effective operators are defined by infinite processes, and cannot in general be evaluated in a finite amount of time. Such operators enable simpler descriptions of practical systems than those possible using effective operators, but they also allow descriptions of processes that are not computable. It is necessary to add restrictions to specification languages to make them uniformly executable, but such restrictions tend to bring the languages down to the programming level.

An example of executable specifications is provided by the algebraic approach to specifying data types, in which a data type is specified by giving axioms for the primitive operations of the type in the form of conditional equations. Algebraic specifications can be made executable by imposing restrictions on the form of the axioms, such as those shown in Fig. 3. These conditions allow the axioms to be treated as rewrite rules, and there exist algorithms for checking that the conditions hold for particular sets of axioms.

An example of an algebraic specification with these properties is shown in Fig. 4. This example defines an abstract data type whose elements are finite sets of values drawn from the type t, where the type

2.2.1. What Should a Design Language Do?

A design document should provide brief and natural descriptions of implementation strategies, justifications, assumptions, conventions, module decompositions, module dependencies, algorithms, and data structures. The design language should support such descriptions with a controlled degree of incompleteness to avoid too much detail. Details that must be supplied by an implementation should be locatable by a mechanical procedure. Designs can be analyzed with respect to correctness, performance, and development cost. Design tools can produce summary information of design for design review, project planning, and management.

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

The top level structure of an algorithm can be described in a design language using conventional control constructs as shown in Fig. 6. Design languages gain expressive power by including non-effective operations in the set of primitives that can be combined with these control constructs. For example, the test of a conditional can include logical quantifiers, the range of a *foreach* loop can be an implicitly specified set, and the actions governed by a conditional or a loop can be described implicitly using a transition predicate, as illustrated in Fig. 7. All of these primitives illustrate ways of defining parts of algorithms without going into coding details. The test in (a) might appear in a system for scheduling meeting, the loop range in (b) might appear in a system for evaluating employee performance, and the transition predicate in (c) might appear in the design of a search tree. The transition predicate describes a balancing operation on two

---

(a) if [ ] then [ ] else [ ]
(b) foreach [ ] in [ ] do [ ]

Fig. 6 Algorithmic Constructs for Design

(a) Non-effective test
SOME(t: time :: unscheduled(t, t + meeting_length))

(b) Implicit loop range
{e: employee :: overtime(e) > required_overtime}

(c) Transition predicate
-- balance the sequences x and y
TRANSITION append(x, y) = append(*x, *y) & length(x) <= length(y) <= length(x) + 1

Fig. 7 Examples of Design Language Primitives

sequences which preserves the order of the elements and makes the two sequences approximately equal in length. The notation shown in the figure interprets variables prefixed with a "*" in the state before the transition and variables without a prefix in the state after the transition. Formal transition predicates such as this one can be used for computer-aided testing or proving of implementations and for automatically constructing prototypes to determine the properties of a proposed design.

2.2.2. Design Justifications

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

Data invariants are restrictions on data structures that must be respected by all programs creating or modifying its instances. Data invariants usually apply to the implementation structures for abstract data types or abstract state machines, serving as hidden internal properties specified in the design of a module. Many of the well known data structures for efficiently implementing common data types gain their efficiency from elaborate data invariants that have been crafted to avoid recomputation of various properties of the data structure. The data invariants constitute the assumptions shared by the implementations of all operations of a type. Since they are not local to a single procedure they can be a vehicle for unwanted interactions, especially for types so large that it is not practical for the same person to implement all the operations. Bugs caused by procedures damaging invariants are common and are difficult to diagnose

* The axioms must be orientable so that the right hand side of each equation is strictly less than the left hand side with respect to some well founded ordering on symbolic terms.
* The oriented axioms must be confluent [12].
* The set of axioms must be sufficiently complete [10].
* Every variable on the right hand side of an axiom must also appear on the left hand side.

Fig. 3 Restrictions Defining Executable Algebraic Specifications

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

Fig. 4 An Executable Algebraic Specification

parameter t can be replaced by any data type. The free variables in each equation are implicitly universally quantified. Equations in this form are equivalent to recursive definitions. Consequently, writing specifications in the restricted form is much like programming. Sometimes it is necessary to introduce auxiliary operations to define the operations we really want. In the restricted form shown in the example, it is difficult to define the **equal** operation on sets in terms of the **in** operation without introducing an auxiliary operation such as **subset**. If the problem does not require a **subset** operation, then introducing one complicates the specification by adding unnecessary details.

To illustrate the simplifications gained by allowing non-effective operators, an example of a Spec 87 [6] fragment defining the equal operation on sets is shown in Fig. 5 (a). This specification says two sets are equal if they have the same elements. This specification is simpler than the corresponding algebraic specification, since three axioms have been replaced by one and the auxiliary concept **subset** has been eliminated. The specification (a) is not executable in its original form because the bound variable x ranges

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

Fig. 5 Making a Non-effective Specification Executable

over a potentially infinite type t, but it is subject to the meaning-preserving transformations shown in Fig. 5 (b) and (c). The transformed specification (c) is executable by enumeration because the bound variables x and y have been restricted to finite sets. Informally, the transfomed specification says two sets are equal if all of the elements of the first are contained in the second and vice versa. The assertion (c) can be evaluated with a relatively small number of operations, given a facility for generating all of the elements of a finite set, so that it is executable in a practical sense as well as a theoretical one. Transformations such as this one bridge the gap between non-effective operators and executable specifications, thus providing the best of both worlds. More work is needed to characterize the cases where such transformations are possible and to develop general methods for constructing them.

2.2. Design Languages

A design is a clear-box description of a software system. A clear-box description gives the decomposition of a component into lower level components and defines their interconnections in terms of both data and control.

Design languages are used to record a design, which decomposes a system into a hierarchically structured set of components. A specification language is used for expressing black-box descriptions and a design language is used for expressing clear-box descriptions of each component. Design languages can be used for formulation, communication, analysis, and planning in the same way as specification languages. They are a medium of communication between the designers, the managers of the project, the programmers, and the design tools. Concise design notations are important for inventing, recording, and communicating designs and connecting specifications.

based on fault symptoms because they involve interactions between pieces of code that are separated both in the text and in execution time. This justifies expending a fair amount of effort on documentation and checking.

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

Bounding functions are justifications for believing that the loops and recursions in the program will terminate. A bounding function gives an upper bound on the number of loop iterations still left for given values of the state variables of the loop, or an upper bound on the depth of any remaining recursive calls for given values of the formal parameters of a recursive subprogram. A terminating program will strictly reduce the bounding function after each execution of the loop body or upon each recursive call. Checking the termination of a program becomes easy if the bounding functions are given. The bounding functions are also useful for performance analysis, because they give worst case estimates of the running times.

The kinds of justifications described above can be used in the process of formally or informally verifying the correctness of a design with respect to a given specification. Other kinds of justifications include priorities for different design goals, such as optimize space. Such justifications are useful when a system must be changed to meet evolving requirements.

2.2.3. Should Design Languages be Extensible?

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

2.3. Programming Languages

Since programming languages are familiar to most readers they will be discussed briefly. Programming languages are formal notations used to record programs. These notations can be processed by a variety of automated tools, such as compilers, static analyzers, debuggers, execution profilers, etc.

In the traditional waterfall model of software development, the implementation phase produces a document expressed in a programming language. Most of the computer-aided design in traditional software development environments is applied in the implementation and later phases, and most of the automated tools currently in use are based on programming languages rather than specification or design languages. The programming languages used in most major software development projects have been designed to emphasize execution efficiency, possibly at the expenses of clarity, flexibility, and expressiveness.

2.4. Relations Among Language Categories and Implication for Prototyping Languages

Prototyping languages are used in requirements analysis for the purpose of requirements validation via early demonstrations to the customer. They are also useful for evaluating competing design alternatives, validation of system structures, and feasibility studies. 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.

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. 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. Computer aid is desirable for determining the properties of a specification and certifying 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 *difference* between a design and a program is the difference between a plan and a finished product: a design records the early decisions that determine an implementation strategy, while a program contains all the details necessary to get an efficiently executable system. The primary goal of a design is documentation rather than execution, while the primary goal of a program is usually efficient execution.

Common *strengths* of specification languages are simplicity, abstraction, clarity of expression, and means for rigorous logical reasoning. Common *strengths* of design languages are expressiveness and support for recording goals and justifications. A common *weakness* of specification and design languages is lack of efficient facilities for execution or lack of any effective means for execution. The *strength* of most programming languages is supporting efficient execution, while common *weaknesses* are the need for specifying many details and lack of facilities for recording goals and justifications in a formal way. The contribution of a prototyping language is to integrate the functions of specification and design languages with the capability for execution. However, because of the wide range of goals for prototyping languages,

they may not be as effective for any of the purposes mentioned above as a language optimized just for that purpose.

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

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. In the near future these four kinds of languages will remain distinct to more effectively support different classes of powerful CASE tools. Programming languages will support optimizing compilers whose main objective is to produce efficient implementations. Specification and design languages will support CASE tools for requirements analysis and for proving the correctness of designs and implementations. Prototyping languages will support tools for prototype demonstrations and implementation planning.

3. Prototyping Languages and CAPS Systems

The purpose of a prototyping language is to define an executable model of a system, using both black-box and clear-box descriptions. A prototyping language has no obligation to give detailed algorithms for all components of the system as long as it is descriptive and executable.

To be useful, prototypes must be constructed quickly and at low cost. To achieve this, a rapid prototyping language should be integrated with a systematic prototyping method and a comprehensive set of tools for computer-aided software design and prototyping. The goals for such an integrated prototyping system are:

(1) Rapid construction and adaptation of software,

(2) Enabling the development of more powerful systems,

(3) Checking if specified systems are acceptable to users,

(4) Checking internal consistency of proposed designs, and

(5) Generating production code and ensuring it conforms to specifications.

Such a system should automatically supply programming level details needed for execution, help the designer construct, analyze, explain, demonstrate, and modify the prototype, and help the development team transform the prototype into a production version of the system. Prototyping systems thus span the entire range of computer-aided software engineering technology. A prototyping system is designed together with a prototyping language. The design of the prototyping language is constrained by the need to support the software tools in the prototyping system.

3.1. Requirements for a Prototyping Language

A prototyping language has two interfaces: one to human users and the other to the software tools in the prototyping system. To support the human users, a prototyping language should be easy to write, understand, and modify. To support the tools, the language should be easy to analyze and transform mechanically.

A prototyping language should have a *simple structure* and *clear semantics* to make it easy to learn, understand, and process mechanically and rapidly. This implies uniform structure, a small number of orthogonal constructs, and general interpretations without special cases or restrictions. To support automated tools, the language should have a simple abstract syntax and an unambiguous and precisely defined meaning. The underlying model should have a mathematical basis to support execution, analysis, verification, and trusted transformations. In particular, the semantics of the language should support rigorous reasoning about the properties of prototypes described in the language and transformations on

expressions of the language. The language should also support a user interface for communication with untrained people, with graphical summary views, English paraphrasing, and explanation facilities.

A prototyping language should be *expressive, clear,* and *concise* to make the language easy to use for prototyping a wide variety of systems. This implies language support for abstractions, uniform communication, logical inference, incomplete descriptions, and automated design completion. In addition to providing traditional facilities for functional, data, and control abstraction, the language should also support abstractions for concurrency, synchronization, and timing constraints. The constructs of the language should correspond directly to decisions made by the designer, rather than to operations performed by the processor, to make prototype descriptions self-documenting and easy to change. The language should allow the designer to specify only the essential attributes of a proposed system. This requires automatically supplying default values for all attributes needed for execution of a software prototype. The language should be capable of constructing the software tools in its own prototyping environment.

To support large scale prototypes, system evolution, and parallel execution, a prototyping language should have mechanisms for *localizing design decisions* in the description and localizing interactions between system components or pieces of knowledge in the knowledge base. These features allow independently designed subsystems of complex systems to cooperate without unexpected interference.

A prototyping language should have facilities for recording *black-box specifications* to support prototype component documentation, verification via proofs and automated testing, and queries for reusable component retrieval. Such descriptions also form the basis for automated synthesis capabilities, inheritance of common properties and constraints, and consistency checking. For expressiveness, this part of the language may contain non-computable constructs such as quantifiers ranging over unbounded sets. The language should also have facilities for describing *clear-box characteristics* of designs such as interconnections of available components, dependencies between components, design goals such as invariant constraints or bounding functions, and design justifications such as criteria for choosing between alternative designs.

The language should have a *distinguished executable subset* that is easily recognizable, both by human users and automated tools. Every expression in this distinguished subset should be executable for all possible initial conditions, although some expressions may denote non-terminating computations. The

distinguished subset should be as large as possible given the above constraints. In particular, some expressions outside the distinguished subset may be executable or partially executable, in the sense that execution may fail in some cases. It should be possible to either augment or transform expressions of the language outside the distinguished executable subset to make them recognizably executable.

To avoid complicating the language, particular high level abstractions should be expressed as *standard pre-defined components* in a software base whenever it is possible to do so without extending the language. Such components should have black-box descriptions in the language, for both documentation and retrieval. The language should have facilities for adapting components to new uses and making small perturbations on their behavior without examining the details of the internal implementation of the components, to make it easier to reuse components.

To save designer time, the language should support the construction of efficient implementations by *augmenting* the prototype description with annotations describing additional constraints or lower level design decisions. This enables the designer to view optimization as a refinement step where additional information is added to the original descriptions, rather than a complete reformulation of the system description. Such an approach saves designer time by avoiding repeated treatment of the same issues in different ways, and by reducing the opportunities for making transcription or translation errors.

Efficiency is mostly of concern for the production version of the system, but it cannot be ignored entirely for the prototype version because it must be possible to run test cases and gather data in a practical amount of time. This implies that execution mechanisms based on exhaustive enumeration are insufficient to meet the requirements of a prototyping language, although they may be supplied as a default to allow running small test cases in the absence of information about more efficient execution strategies. The language should therefore support a set of fairly efficient execution mechanisms, tools for locating performance bottlenecks in larger systems, and incremental optimization transformations to improve prototypes that are impractically slow.

Real-time constraints impose a slightly different set of subgoals: execution times must be predictable, although not necessarily very fast. Prototypes of real-time systems may operate in simulated time or linearly scaled real time, but the actual execution times for the production version must be predictable within accurate bounds [26].

3.2. Foundations of a Prototyping Language

The rapid construction of software prototypes depends on simplifying the view of the system through which the specifiers and designers do their work, and providing automated means for bridging the gap between this simplified view and the detailed programming level description currently needed to make a software system efficiently executable. This automated support should include mechanisms for execution, static analysis of the properties of the proposed system, preparation of test cases, reporting and analyzing results, and diagnosing ill-formed descriptions and departures from desired behavior to allow the specifiers and designers to work entirely within the simplified view, at least during the construction of the initial prototype.

3.2.1. Static and Dynamic Properties of a Prototyping Language

Static properties of a computer language are those that must be fixed before a system can be executed, while dynamic properties can be changed as the system runs. To maintain flexibility in demonstrations and to allow the description of highly adaptive systems, prototyping languages should support dynamic treatment of as many properties as possible. This introduces some difficulties, because implementations can be more efficient and analysis tools can give more information about a system if its properties are fixed.

Some areas where this distinction is relevant are data types, code construction, and scheduling. Programs that can manipulate data types, programs, and schedules at run-time can adapt to unanticipated circumstances more readily than those that cannot. However, introducing these facilities into a prototyping language requires run-time type checking, run-time interpreter calls or dynamic compilation, loading, and linking, and run-time scheduling. All of these features are difficult to implement efficiently, and are not supported by the class of languages usually used for production versions. Also, blanket guarantees of type correctness, clean termination, or meeting hard real-time constraints may not be possible without static restrictions on these properties. Thus a prototyping language should allow selected properties to have static restrictions, and should support transformations that add static restrictions for the purpose of improving efficiency or predictability.

3.2.2. Computational Model for a Prototyping Language

The models underlying the language provide the common ground for the associated set of tools. The semantic model for the language provides the basis for automated analysis, while the computational model provides the basis for execution. One of the main challenges in developing a prototyping language is finding a model that can coherently span the range of applications required. This will require a significant advance in the state of the art.

There is no single common model of expert systems available for rapid prototyping. First order logic is one of the most familiar models for reasoning, but it has been criticized for its weaknesses, such as lack of facilities for handling uncertain information, representing heuristic methods for speeding up conclusions, and non-monotonic reasoning. Many other kinds of logic have been proposed, but these logics are still being explored and there has been no consensus on whether there is a single logic suitable for constructing all types of expert systems, or which variety of logic is the most promising. There are also approaches to expert systems that are based on models other than logic, such as semantic networks, Bayesian statistics, and production systems. Since it is not clear which approach will yield the best results in the long run, a prototyping language should find a unified way of treating most of the issues raised by this diverse set of models.

There is also no single commonly accepted model for representing real-time constraints. Some approaches that have been explored include temporal logic, state machines, mode charts, augmented data flow diagrams, Petri nets, and I/O automata. The model for a prototyping language should be chosen to enhance the application of recent results in logic, graph theory, and combinatorics to link the semantic model to an effective execution mechanism. Other unexplored areas include effective models for real-time databases and real-time communications networks. In both of these areas, the problems of providing service within guaranteed worst-case time bounds are largely unexplored.

3.2.3. Execution Support for a Prototyping Language

To provide adequate execution support for a prototyping language without requiring many programming-level details, it is necessary to take a knowledge-based approach. The supporting environment for the language should provide knowledge base support for the following functions:

Managing reusable components - The environment should contain a large software base with reusable components. This software base should be coupled with a set of rules for tailoring and combining available components to fulfill queries that do not exactly match any of the components explicitly stored in the software base. This insulates the designer from programming details because it allows the system to find algorithms and data structures for realizing some classes of black-box specifications.

High level debugging - Errors and failures during prototype execution should be mapped from the programming language level to level of the prototyping language, to allow the designer to work entirely in terms of the semantic model associated with the prototyping language. This function is necessary to keep programming details from intruding when the designer tests and demonstrates the prototype.

Optimization - The transformations for optimizing a prototype version of a system to produce a production version should be performed with minimum interaction with the designer. This implies keeping track of the decisions made by the designer in optimizing previous versions of the system, determining which of those decisions are still valid for later versions, and automatically applying the ones that are found to be still valid. While it is not feasible at the current state of the art to produce highly optimized implementations without human help for the major decisions, it should be possible to filter out the routine decisions and rely on the human designer for only the most difficult decisions involving estimates of load characteristics and execution frequencies from informal characterizations of the problem domain.

Explanations - Justifications for decisions made automatically should be available to provide feedback to the designer in cases where automated design completion procedures fail. Such a facility is needed to support systematic computer-aided design in situations where complete automation is not possible, which includes many aspects of computer-aided prototyping at the current state of the art. This requires an expert system with a substantial knowledge base.

3.3. Semantics of a Prototyping Language

The development of an integrated set of prototyping tools requires a consistent and simple semantic model rich enough to express and support the entire range of expected applications. Finding suitable models is the key to computer-aided prototyping.

3.3.1. Language Support for Real-Time Systems

The language should include a means of declaring timing constraints and overload resolution policies. Scheduling is a difficult issue for real-time systems. High level representations of timing constraints and overload resolution policies are essential to allow the prototype to express the necessary constraints on the scheduling of different tasks at a level matching the problem rather than at the level of the underlying run-time support system. Timing constraints on data transfers are needed to express timing constraints in distributed systems.

3.3.2. Language Support for Parallel Systems

To simplify and speed up the construction of parallel systems, it is essential to provide means for defining independent activities that are guaranteed not to interfere with each other and to provide high-level means for coordinating independent activities [16]. Localized modules with limited data access are essential for this purpose. Message passing, dataflow, and object-oriented ideas are relevant to this area. Another consideration is avoidance of deadlock. It is useful to have a syntactically recognizable subset of the language that is capable of describing concurrent computations and carries a uniform guarantee of freedom from deadlock. Such a guarantee is possible if a suitable computational model is chosen. While it may not be possible to design all concurrent systems using just the deadlock-free subset, this kind of restricted subset is sufficient for many applications, and it can be augmented with facilities for adding additional constraints on global orderings of events, which are known to be potentially unsafe and which are designed together with tools for checking safety of particular designs. For example, atomic transactions are essential for simplifying the design of distributed systems, although they introduce the potential for deadlocks.

3.3.3. Language Support for Distributed Systems

The important aspects of distributed systems are communications delays and atomic transactions. A prototyping language should provide a high level means for describing

(1) constraints on the relation between software tasks and physical processors,

(2) constraints on communication time,

(3) standard protocols for achieving reliability despite processing and communications failures,

(4) the granularity of atomic transactions.

In all of these cases, the information should be optional, and the default values should provide the safest option, rather than the most efficient one.

3.3.4. Language Support for Knowledge-Based Systems

Many of the standard building blocks for knowledge-based systems or expert systems can be provided as standardized generic predefined components. These include facts, rules, patterns, frames, contexts, constraints, demons, instance generators, pattern matchers, unification mechanisms, and forward and backward chaining inference engines. Standardization requires careful analysis of these components and specification of their required properties. An open issue is whether current mechanisms for defining generic components are flexible enough to adequately capture the range of behavior required for these kinds of components, and if not, what extensions are required.

Some features that a prototyping language should provide to support rapid construction of expert systems include:

(1) a means for conveniently defining external representations and input facilities for the knowledge in the knowledge base,

(2) support for the first class treatment of higher order objects such as types, functions, tasks, and generators, and

(3) support for control mechanisms such as state-triggered demons, backtracking, run-time control over task priorities, and scheduling of temporal events.

Several of these features are needed to support flexible prototypes for other kinds of systems as well.

The presence of real-time constraints severely restricts the kinds of computations a system may perform, and in the case of expert systems, limits the amount of logical inference that can be performed. The design of expert systems that operate within real-time constraints is a largely unexplored area, and significant research progress is needed in this area to fully realize the goals of a rapid prototyping language.

3.4. Prototyping Methodology and Tool Support

The language and its supporting tools should be designed to support a systematic method for constructing prototypes. One approach to systematically constructing prototypes and to support software evolution via rapid prototyping is described in [24,27]. The basis for the approach is the combination of a set of guidelines for decomposing software modules, a set of reusable components, and an automated software base management system with inference capabilities [25,28].

Ordinary compiler technology is insufficient for execution of a prototyping language. Conventional translation techniques must be coupled with facilities for scheduling to meet real-time constraints [30] and with transformations to allow the execution of incompletely specified processes.

Many programming languages do not support parallel processing, and those that do generally do not provide sufficient control over the scheduler to guarantee that real-time constraints can be met. For example, Ada provides support for tasks and allows pragmas (compiler directives) for specifying static priorities, but does not have any direct means for guaranteeing an upper bound on scheduling delays. Since this is somewhat removed from the level of support needed for implementing hard real-time systems, the execution support system for a prototyping language will have to provide higher level facilities for scheduling real-time operations. Such facilities can be classified as on-line (done at run-time) and off-line (done prior to execution). There is no universally accepted approach to real-time scheduling. Optimal scheduling algorithms are very time consuming, and generally cannot be carried out on-line, while off-line approaches are inflexible and do not handle overload situations very well. There are many different scheduling algorithms, and choosing the best one for a given application is a difficult problem. Thus the execution support system for a prototyping language should provide the designer with several choices with respect to scheduling, and the prototyping language should provide a means for specifying those choices, with reasonable defaults.

Transformations are needed to execute incompletely specified components. Such transformations should supply reasonable default values for attributes necessary for execution if the designer does not explicitly specify them. Different choices for these attributes can be explicitly specified to produce a more accurate model of the system or to improve its performance. In particular, default algorithms for unspecified or partially specified components should be supplied. It is essential to automatically generate

stubs for components that are unavailable in the software base and have not yet been addressed by the designer to allow testing and demonstrating partially completed systems. Such stubs can be created by simple or increasingly sophisticated techniques, such as asking the user to supply values, using random selections from a fixed set of responses, using logic programming to simulate black-box specifications, or using transformation techniques to generate efficient implementations from the black-box descriptions. Other examples include assignment of tasks to physical processors and choosing display formats for outputs and error messages.

The tool set should provide facilities for analyzing the consistency of a prototype design. Some of the checks that should be performed include:

* Type consistency,
* Feasibility of timing constraints,
* Consistency between the levels of a hierarchical description,
* Preconditions on input parameters and generic parameters,
* Constraints on relative rates of producer and consumer processes,
* Absence of deadlocks in distributed and parallel systems,
* Absence of unhandled exceptions.

In addition to providing facilities for constructing and checking the internal consistency of a prototype, the tool set should provide facilities for generating input data, debugging, displaying output, and evaluating the results of prototype execution at the in terms of the same semantic model used for the design of the prototype.

To support user validation and system evolution, a prototyping language should support a facility for maintaining the correspondence between requirements and design decisions. Tools will be needed for determining which parts of a description must be removed or modified when a requirements change removes the support for previously made design decisions, and for determining which requirements are affected by a proposed change to the behavior of a prototype.

The tool set must also provide a design database for maintaining the design history in terms of a set of versions of the system and the alternative designs that were considered. This database should also be capable of recording and maintaining constraints on the system. A related issue is the relation between the

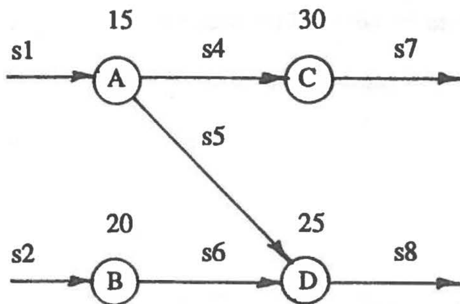
language, which is used for describing the design objects in the database, and the notations for describing the attributes, relationships, and constraints among those objects that are used by the tools in the associated environment. The information in the designer's view of the language is likely to be subset of the information in the tool views, since tools are likely to add additional attributes to the entities defined by the language, recording the results of analysis and synthesis procedures.

3.5. Example of a Prototyping Language

The rapid prototyping language PSDL [22] and its CAPS were designed to support prototyping of large, parallel, and real-time systems. The language has a simple and expressive computational model based on modified dataflow augmented with non-procedural constraints. PSDL encourages localized descriptions and software structures, and provides execution facilities that realize timing constraints with respect to either actual or linearly scaled real-time [1].

PSDL prototypes have a *specification part* for black-box descriptions and an *implementation part* for clear-box descriptions. The black-box descriptions are used both for documentation of the prototype and for retrieval of reusable software components. The black-box specifications are executable without further information from the designer if the CAPS can automatically retrieve, adapt, and combine the components in its software base to match the specification. In cases where this is not possible, the designer must provide an implementation part giving a clear-box description decomposing the specified system into more primitive subsystems along with black-box specifications of the subsystems. The decomposition is done in terms of the PSDL computational model, using augmented data flow diagrams. The nodes in an augmented data flow diagram are operators representing functions or abstract state machines, while the edges are data streams carrying instances of abstract data types or exceptions. An example of an augmented data flow diagram is shown in Fig. 8, along with a two processor schedule that takes advantage of parallel execution to meet the timing constraints.

The data flow diagram is augmented with *non-procedural control constraints* and *hard real-time constraints*. The control constraints support conditional execution, conditional output, control of exceptions, and control of timers. The real-time constraints support both periodic execution and sporadic, data-driven execution with bounded response times. The control constraints and the timing constraints deter-



CONTROL CONSTRAINTS
OPERATOR A PERIOD 100 FINISH_WITHIN 50
OPERATOR B PERIOD 100 FINISH_WITHIN 50
OPERATOR C PERIOD 100 FINISH_WITHIN 50 OUTPUT s7 IF s7 > s4
OPERATOR D PERIOD 100 FINISH_WITHIN 50 TRIGGERED IF s5 + s6 < 10

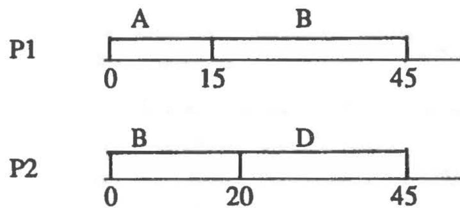


Fig. 8 Example of an Augmented Data Flow Diagram in PSDL

mine the conditions under which the operators are triggered and the buffering disciplines for the data streams. Dataflow streams act as first-in-first-out buffers, and are used for synchronizing data-driven computations. Sampled streams act as continuously available sources of data that can be read or updated on demand, and are used for connecting unsynchronized operators which can fire at different or unpredictable rates.

Locality is realized in PSDL by the absence of a mechanism for transmitting objects with internal states along data streams and scoping rules that do not allow direct non-local data references. These mechanisms ensure that operators can interact only via the documented interfaces, and can be executed in parallel without danger of interference. This simplifies the design of parallel systems and makes it easier to modify prototype behavior without damaging the design.

The mechanisms in PSDL can be used to prototype *knowledge-based systems*. For example, the control constraints of PSDL can be used to represent rules and demons. To effectively support the construction of knowledge-based systems in PSDL, the software base in the CAPS should be extended to include a suitable set of reusable software components, such as several types of inference engines and data types for representing rules, patterns, frames, contexts, etc. The language would be more effective for these purposes if it were augmented with a facility for creating concise external representations for the knowledge in a knowledge-based system, and tools providing graphical support for displaying and analyzing that knowledge.

PSDL requires *static definitions* of types, operators, and timing properties to support static type checking and scheduling. Knowledge-based systems are often currently designed using dynamic constructs which create new types and operators at run-time, or to control the schedulers based on information determined at run-time. The freedom such constructs provide appears to make design easier, but it is not clear whether they are inherently necessary for constructing knowledge-based systems. While PSDL cannot directly express such dynamic constructs, it can simulate them. An inference engine is similar to an interpreter, and can provide alternative meanings for PSDL objects within its scope. This approach can simulate capabilities such as higher order types, types whose instances are treated as operators, and expressions representing priorities for different activities of the inference engine. Thus the addition of new components to the software base can extend the semantics of the language in radical ways. While such an interpretive approach may impose a performance penalty, it may be the only viable approach to prototyping knowledge-based systems if dynamic treatment of types, operators, and priorities turns out to be inherently necessary for designing such systems and the production code must be written in a statically oriented programming language such as Ada.

The addition of hard real-time constraints to knowledge-based systems is a difficult problem, but the essential difficulty belongs to the *problem domain* rather than to the *prototyping domain*. It is possible to design an inference engine that performs a single logical inference in a bounded amount of time if the size of the assertions and the number of assertions in the knowledge base are bounded. A prototype for such a design can be described at a reasonable level by a language such as PSDL. The difficult problems in creating knowledge-based systems that respond within hard real-time constraints are determining the bounds on

the number of logical inferences needed to solve a given problem, the size of the required knowledge base and the size of the assertions in the knowledge base.

4. Conclusions

The state of the art in computer languages and computer aided software engineering is sufficient to support exploratory development of a general purpose rapid prototyping language. The first special purpose rapid prototyping language PSDL and its tool set were published in 1985 [38] for the general use of rapid prototyping in the development of large and real-time systems. Many languages previously designed for programming, design or specification were used or intended to be used for prototyping purposes in different degrees. In the 1970's, the SETL programming language was designed based on set theory and already had the primitive concepts for integrating high level mathematics with an execution capability [17]. Other languages that have been used for prototyping at the programming level include APL, SNOBOL, LISP, and PROLOG. The Gypsy language from UT Austin provides a simple basis for representing, executing, and proving correctness of communicating processes. Research on real-time modeling and scheduling provides fundamental support for the design of hard real-time systems. Work on attribute grammars paved the road to automated approaches for prototyping special purpose languages [32]. The GIST system at ISI has explored computer-aided requirements modeling with the aid of symbolic evaluation and English paraphrasing [14]. The Argus project at MIT has explored implementation of atomic transactions in distributed systems [19]. The wide spectrum language Refine from Reasoning systems, the DRACO system from UC Irvine [8], and the CIP project from Dr. Bauer's group in Munich [2] have explored the feasibility of using transformations to realize specifications.

DARPA has decided to develop designs for a rapid prototyping language, which is intended to apply to a variety of large software systems, including knowledge-based systems, parallel systems, distributed systems, and real-time systems [7]. The DARPA Common Prototyping Language project has an ambitious set of goals that raises many interesting research problems [9]. Solutions to these problems are essential for achieving significant improvements in the quality and productivity of the software development process. Goals for this language include computer-aided transformations of prototypes into Ada implementations of the production version of the software, and eventually implementing the tools of the prototyping system in Ada to provide portability. Ordinary compiler technology is insufficient for execution of the

prototyping language. The need for flexibility and run-time handling of newly created types and procedures to support expert systems provides challenges for efficient implementation techniques in terms of Ada. Conventional translation techniques must be coupled with facilities for scheduling to meet hard real-time constraints, transformations to allow the execution of incompletely specified processes, and access to an interpreter or an incremental compiler at run-time.

Ada provides a completely static type system, treats types and functions as second class objects, and requires task priorities to be known at compilation time. The flexibility required for supporting expert systems development can be provided by adding a run-time interpreter on top of the Ada language. The problem will be to provide these features efficiently, and without introducing excessive run-time overheads for prototyping applications that do not require such flexibility. Ada provides relatively weak guarantees about the scheduling of tasks, and limits programmer control over scheduling to statically specified priorities. Since this is somewhat removed from the level of support needed for implementing hard real-time systems, the execution support system for the prototyping language will have to provide higher level facilities for scheduling real-time operations. There is no universally accepted approach to real-time scheduling. Optimal scheduling algorithms are very time consuming, and generally cannot be carried out on-line, while off-line approaches are inflexible and do not handle overload situations very well. Choosing the best algorithm for a given application is currently difficult. Better practical algorithms and better criteria for choosing among them are needed. Maximum impact on software development practice depends on transformations from prototyping languages to Ada. Key problems are finding systematic ways of developing such transformations and determining reasonable default values based on models of the application domain.

Since the completely automatic and totally correct implementation of powerful specification languages is an algorithmically unsolvable problem, research on rapid prototyping should emphasize human interaction for effectively guiding computer-aided implementation tools. A promising approach is augmenting abstract specifications with annotations or pragmas giving advice about implementation strategies. An important problem is finding concepts and notations that can naturally express such advice in an abstract and orthogonal way. It is desirable to keep the abstract specification separate or easily mechanically separable from the annotations to provide simplified views of large system models.

Progress on automatically generating prototypes or efficient implementations from abstract specifications depends on a knowledge-based approach. The size of the required knowledge bases depends on the range of problems the language attempts to address. 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 relevant knowledge include methods for adapting and combining the components in the software base, properties of application domains and properties of the CAPS environment.

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. F. Bauer, B. Moller, H. Partsch and P. Pepper, "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming", *IEEE Trans. on Software Eng. 15*, 2 (Feb. 1989), 165-180.
3. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Comm. of the ACM 29*, 5 (May 1986), 402-415.
4. 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.
5. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1989.
6. 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.
7. DARPA Information Science and Technology Office, "Broad Agency Announcement for New Language in Rapid Construction of Software Prototypes", *Commerce Business Daily*, Arlington, VA, Feb. 1989.
8. P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems", *IEEE Trans. on Software Eng. SE-13*, 7 (July 1987), 830-844.
9. R. Gabriel, ed., *Draft Report on Requirements for a Common Prototyping System*, DARPA Information Science and Technology Office, Arlington, VA, Nov. 1988.
10. J. V. Guttag, "The Specification and Application to Programming of Abstract Data Types", CSRG-59, Ph. D. Thesis, University of Toronto, 1975.
11. R. Holte, A. Mok and L. R. etc, "The Pinwheel: a Real-Time Scheduling Problem", in *Proceedings of 22nd Hawaii International Conference on System Science*, Kona, Hawaii, Jan. 1989.
12. G. Huet, "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *J. ACM 27*, 4 (Oct. 1980), 797-821.
13. F. Jahanian and A. Mok, "A Graph-Theoretic Approach for Timing Analysis and Its Implementation", *IEEE Transactions on Computers C-36*, 8 (August 1987), 961-975.
14. L. Johnson, "Overview of the Knowledge-Based Specification Assistant", in *Proc. Second Annual RADC Knowledge-based Assistant Conference*, RADC(COES), Grifiss AFB, NY, 1987.
15. M. Ketabchi and V. Berzins, "Modeling and Managing CAD Databases", *IEEE Computer 20*, 2 (Feb. 1987), 93-102.

16. 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.

17. P. Kruchten, E. Schonberg and J. Schwartz, "Software Prototyping Using the SETL Programming Language", *IEEE Software 1*, 4 (Oct. 1984), 66-75.

18. K. Lin, S. Natarajan and J. Liu, "Imprecise results: Utilizing partial computations in real-time systems", in *Proceedings of the 8th Real-Time Systems Symposium*, San Jose, Dec. 1987, 210-217.

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

20. J. Liu, K. Lin and X. Song, "Scheduling hard real-time transactions", in *Proceedings of the 1988 Workshop on Real-Time Operating Systems and Software*, Washington, D. C., May 12-13, 1988.

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

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

23. 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.

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

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

26. 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.

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

28. R. Mittermeier and W. Rossak, "Software Bases and Software Archives Alternatives to Support Software Reuse", in *Proceedings 1987 Fall Joint Conputer Comference*, IEEE, Oct. 1987, 21-28.

29. A. Moitra, *Analysis of Hard Real-Time Systems*, Computer Science Department, Cornell University, 1985.

30. A. Mok and S. Sutanthavibul, "Modeling and Scheduling of Dataflow Real-Time Systems", in *Proceedings of the Real-Time Systems Symposium*, IEEE, San Diego, CA, Dec. 1985, 178-187.

31. C. Monma and J. Sidney, "Optimal Sequencing via Modular Decomposition: Characterization of Sequencing Functions", *Mathematics of Operations Research 12*, 1 (Feb. 1987), 22-31.

32. T. Reps, "Generating Language Based Environments", Ph. D. Thesis, University of Massachusetts, Amherst, 1983.

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

34. J. Stankovic, "Decentralized Decision Making for Task Reallocation in a Hard Real-Time System", *IEEE Transactions on Computers 38*, 3 (March 1989), 341-355.

35. M. Tanik and R. Yeh, "The Role of Rapid Prototyping in Software Development", in *Proceedings of the 22nd Hawaii International Conference on System Science*, Kona, Hawaii, Jan. 1989, 337-338.

36. J. Tsai, M. Aoyama and Y. Chang, "Rapid Prototyping Using FRORL Language", in *Proc. COMPSAC 88*, Oct. 1988, 410-417.

37. S. Tyszberowicz and A. Yehudai, "OBSERV Object-oriented Specification, Execution and Rapid Verification System", in *3nd Israeli Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, June 1988.

38. R. Yeh, N. Roussopoulos, Luqi and etc., "Research in Software Reusability", Final Report, Tech. Rep.-p106/83/0004-3, Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama, July 1985.

39. W. Zhao, K. Ramamritham and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, May 1987.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 — 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Office of Naval Research — 1
   Office of the Chief of Naval Research
   Attn. CDR Michael Gehl, Code 1224
   800 N. Quincy Street
   Arlington, Virginia 22217-5000

4. Space and Naval Warfare Systems Command — 1
   Attn. Dr. Knudsen, Code PD 50
   Washington, D.C. 20363-5100

5. Ada Joint Program Office — 1
   OUSDRE(R&AT)
   Pentagon
   Washington, D.C. 20301

6. Naval Sea Systems Command — 1
   Attn. CAPT Joel Crandall
   National Center #2, Suite 7N06
   Washington, D.C. 22202

7. Office of the Secretary of Defense — 1
   Attn. CDR Barber
   STARS Program Office
   Washington, D.C. 20301

8. Office of the Secretary of Defense — 1
   Attn. Mr. Joel Trimble
   STARS Program Office
   Washington, D.C. 20301

9. Commanding Officer — 1
   Naval Research Laboratory
   Code 5150
   Attn. Dr. Elizabeth Wald
   Washington, D.C. 20375-5000

10. Navy Ocean System Center 1
Attn. Linwood Sutton, Code 423
San Diego, California 92152-500

11. National Science Foundation 1
Attn. Dr. William Wulf
Washington, D.C. 20550

12. National Science Foundation 1
Division of Computer and Computation Research
Attn. Dr. Peter Freeman
Washington, D.C. 20550

13. National Science Foundation 1
Director, PYI Program
Attn. Dr. C. Tan
Washington, D.C. 20550

14. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

15. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn: Dr. James Smith
800 N. Quincy Street
Arlington, Virginia 22217-5000

16. New Jersey Institute of Technology 1
Computer Science Department
Attn. Dr. Peter Ng
Newark, New Jersey 07102

17. Southern Methodist University 1
Computer Science Department
Attn. Dr. Murat Tanik
Dallas, Texas 75275

18. Editor-in-Chief, IEEE Software 1
Attn. Dr. Ted Lewis
Oregon State University
Computer Science Department
Corvallis, Oregon 97331

19. University of Texas at Austin 1
Computer Science Department
Attn. Dr. Al Mok
Austin, Texas 78712

20. University of Maryland 1
College of Business Management
Tydings Hall, Room 0137
Attn. Dr. Alan Hevner
College Park, Maryland 20742

21. University of California at Berkeley 1
Department of Electrical Engineering and Computer Science
Computer Science Division
Attn. Dr. C.V. Ramamoorthy
Berkeley, California 94720

22. University of California at Los Angeles 1
School of Engineering and Applied Science
Computer Science Department
Attn. Dr. Daniel Berry
Los Angeles, California 90024

23. University of Maryland 1
Computer Science Department
Attn. Dr. Y. H. Chu
College Park, Maryland 20742

24. University of Maryland 1
Computer Science Department
Attn. Dr. N. Roussapoulos
College Park, Maryland 20742

25. Kestrel Institute 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

26. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. B. Liskov
Cambridge, Massachusetts 02139

27. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. J. Guttag
Cambridge, Massachusetts 02139

28. University of Minnesota 1
Computer Science Department
136 Lind Hall
207 Church Street SE
Attn. Dr. J. Ben Rosen
Minneapolis, Minnesota 55455

29. International Software Systems Inc. 1
12710 Research Boulevard, Suite 301
Attn. Dr. R. T. Yeh
Austin, Texas 78759

30. Software Group, MCC 1
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759

31. Carnegie Mellon University 1
Software Engineering Institute
Department of Computer Science
Attn. Dr. Lui Sha
Pittsburgh, Pennsylvania 15260

32. IBM T. J. Watson Research Center 1
Attn. Dr. A. Stoyenko
P.O. Box 704
Yorktown Heights, New York 10598

33. The Ohio State University 1
Department of Computer and Information Science
Attn. Dr. Ming Liu
2036 Neil Ave Mall
Columbus, Ohio 43210-1277

34. University of Illinois 1
Department of Computer Science
Attn. Dr. Jane W. S. Liu
Urbana Champaign, Illinois 61801

35. University of Massachusetts 1
Department of Computer and Information Science
Attn. Dr. John A. Stankovic
Amherst, Massachusetts 01003

36. University of Pittsburgh 1
Department of Computer Science
Attn. Dr. Alfs Berztiss
Pittsburgh, Pennsylvania 15260

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

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. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

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 — 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, California 90292-6695

57. U.S. Air Force Systems Command — 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

58. U.S. Air Force Systems Command — 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, New York 13441-5700

59 LuQi — 50
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

60. Research Administration — 1
Code: 012
Naval Postgraduate School
Monterey, CA. 93943