LIBRARY
RESEARCH REPORTS DIVISION
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93

NPS52-87-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

RESEARCH ASPECTS
OF RAPID PROTOTYPING

LUQI

MARCH 1987

Approved for public release; distribution unlimited

...ed for:

...of Naval Research
Arlington, VA 22217

FEDDOCS
D 208.14/2:NPS-52-87-006

NAVAL POSTGRADUATE SCHOOL
Monterey, California

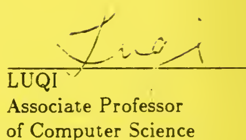
Rear Admiral R. C. Austin
Superintendent

D. A. Schrady
Provost

This report was prepared for the Naval Postgraduate School.

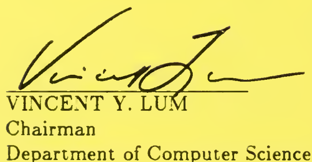
Reproduction of all or part of this report is authorized.

This report was prepared by:



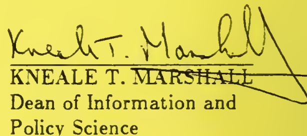
LUQI
Associate Professor
of Computer Science

Reviewed by:



VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:



KNEALE T. MARSHALL
Dean of Information and
Policy Science

Note:

This project was supported by the NPS Foundation Research Program, which was funded by the Chief of Naval Research, Arlington, VA 22217.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-87-006	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RESEARCH ASPECTS OF RAPID PROTOTYPING		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) LUQI		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93941		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N : RR014-01 · W0001487WR4E011
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE March 1987
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Finding effective and efficient methods for determining and validating the requirements for a software system is an important unsolved problem in software engineering. Prototyping is a promising approach for requirements validation. Current prototyping methods require an impractical amount of time and effort. The objective of the proposed research is to make automated rapid prototyping possible.

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

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Research Aspects of Rapid Prototyping

1. Research Summary

The objectives of the proposed research are to develop specification methods for identifying and retrieving reusable software components, to develop implementation techniques for the prototyping language PSDL, and to extend the language and techniques to a multiprocessor model for the prototype.

Our approach to component specifications will be based on term rewriting systems and the concept of generalization per category. We will seek component specifications that admit effective reductions to a canonical or normal form to aid component retrieval. The attributes of reusable software components will be abstracted to support the design of a software base schema, which will be structured using generalization per category to aid component retrieval. The relation between the specification language and the software base operators will be established. Experimental conditions for testing the results will be established through the experimental generation of a series of translators for the prototyping language PSDL with the aid of an attribute grammar based translation tool. We will also seek efficient interpretive techniques for implementing PSDL, extend the language to include scheduling constraints due to distributed external systems, and develop suitable multiprocessor scheduling algorithms.

The proposed research will solve some key problems in automated prototyping based on reusable software. Validating software requirements by rapid prototyping depends on three major components: a prototyping language, a software base, and a prototyping method. The objectives of the proposed research contribute to the software base and the prototyping language.

2. Research Description

Finding effective and efficient methods for determining and validating the requirements for a software system is an important unsolved problem in software engineering. Prototyping is a promising approach for requirements validation. Current prototyping methods require an impractical amount of time and effort. The objective of the proposed research is to make automated rapid prototyping possible.

2.1. Objectives and Significance

The rapidly growing demand for software has shifted towards larger systems and higher quality software, to the point where current software development methods are inadequate. A jump in software technology is needed to improve programming productivity and the reliability of the software product. Rapid prototyping is one of the most promising methods proposed to reach this goal.

A prototype is an executable model or a pilot version of the intended system. A prototype is usually a partial representation of the intended system,

used as an aid in analysis and design rather than as production software. The construction activity leading to such a prototype is called rapid prototyping. Rapid prototyping has been found to be an effective technique for clarifying requirements and eliminating the large amount of wasted effort currently spent on developing software to meet incorrect or inappropriate requirements in traditional software life cycles[1]. A key issue in the design of large software systems is how to agree on the requirements. Lack of agreement on the requirements as specified by the customer and as analyzed by the designer causes inconsistencies between the delivered system and customer expectations, leading to expensive rebuilding[1]. This problem is especially acute for large systems and systems with real-time constraints because the requirements for such systems are complicated to describe and difficult to understand. Because the user can usually recognize whether or not a working software system does what is needed, but usually can't describe the requirements accurately, prototypes are an effective means for achieving stable and accurate requirements early in the development process.

A prototype can also be used to specify a well modularized skeleton design for the intended system and to validate the important attributes of the intended system, e.g. timing constraints, input and output formats, or interfaces between modules. Rapid prototyping is a useful tool in feasibility studies. Prototypes of critical subsystems or difficult parts of a complicated system can significantly increase the confidence that the system can be built before large amounts of effort and expense are committed to the project. Rapid prototyping helps in estimating costs, since the cost of the intended system is usually proportional to the cost of the prototype. The experiences gained in applying rapid prototyping to special applications, e.g. database design, metaprogramming methods and translator design, have substantiated this cost relationship between the prototype and the completed system[2-4].

Software tools are needed to make rapid prototyping practical. An initial description of a framework for a rapid prototyping environment based on reusability can be found in[5]. Since automatic program generation from very high level specifications is not yet practical, reusing existing system components appears to be the most economical approach for constructing prototypes. An important problem in reusable software is finding the relevant software components, because the retrieval must take less effort than constructing the components for this approach to be practical. We propose to develop methods for organizing a software base to aid interactive retrieval of reusable components, and to seek better automated methods for component retrieval. Another important problem is tailoring and connecting the reusable components into a higher level assembly. We believe the best way to do this is by using a language expressly designed for the purpose, and we have designed such a language (PSDL). We propose to investigate efficient ways to implement this prototyping language.

2.2. Relation to Long Term Work

Our long term goal is to enable the construction of a highly automated software engineering environment supporting a development method for large real-time systems based on rapid prototyping. The proposed research addresses

some key steps in our approach towards this goal.

We use an integrated approach to prototyping that combines a computational model tailored for describing real-time systems with a high level prototyping language PSDL [6], a systematic design method for rapid construction of prototypes[7,8], and an automated prototyping environment with a software base[9] containing a large set of reusable software components. The computational model has been designed to prevent hidden interactions between system components, to encourage designs with good module independence. The language supports the model and combines it with a powerful set of data and control abstractions to make it easy to describe system at a high level. The automated environment relies on a software base management system for retrieving and adapting reusable software components, a syntax directed editor for speeding up design entry and preventing syntax errors, and an execution support system for demonstrating and measuring prototype behavior and for performing static analyses of the prototype design.

Rapid construction of a prototype in PSDL is made possible by the associated prototyping method and support environment. The prototyping method relies on an improved modularization technique and reusable software components. The support environment reduces the efforts of the analyst and designer by automating some of the tasks involved in prototype construction. The most important aspects of the support environment are the software base, the prototype execution facilities, and the design entry facilities.

2.2.1. The Prototyping Language PSDL

A good language for expressing design thoughts in terms of a precise model is important for rapid prototyping. It is impossible to do a good design without a language especially designed for this purpose. A powerful, easy to use, and portable prototype description language is also a critical part of an automated rapid prototyping environment. Such a language is needed before the tools in the environment can be built. PSDL (Prototype System Description Language) was designed to serve as an executable prototyping language working at a specification or a design level[6], together with a prototyping method and an automated support environment. The language has special features particularly appropriate for real-time system design. PSDL prototypes are especially well suited for requirements analysis and validation because they are executable and are specified at a high level with requirements tracing.

PSDL[10] and its prototyping method are concerned primarily with hard real-time systems. A hard real-time constraint is a bound on the response time of a process or the period between invocations that must be satisfied under all operating conditions. A hard real-time system[11] has hard real-time constraints as part of its requirements. Such systems are modeled in PSDL as networks of operators communicating via data streams, which uses enhanced data flow diagrams for that purpose. The data streams can carry data values of an abstract data type[12] as well as tokens representing exception conditions. Each type or operator is either composite or atomic. Composite operators are implemented by decomposing them into networks of more primitive operators using PSDL. The decomposition of a composite operator is described in PSDL by an enhanced data flow diagram that includes non-procedural control constraints and timing constraints. Atomic operators are realized by retrieving an

implementation from a software base[9,13] containing reusable software components.

PSDL provides sufficient structures and descriptive ability to describe the internal and external situation for the modules comprising the system. Good modularity is one of the key factors for increasing productivity, since it significantly reduces the debugging effort for producing a correct executable system, and also influences the understandability, reliability, and maintainability of the developed system, which are especially important in rapid prototyping. A clear and powerful modularization model is introduced in PSDL for building and describing the prototype. .The model is based on data flow under real-time constraints. This model and the associated prototyping method[7,8], lead to PSDL prototypes with a highly cohesive structure and few coupling problems. This structure is suitable for multiple modifications at a specification level during the prototyping iterations of the new life cycle.

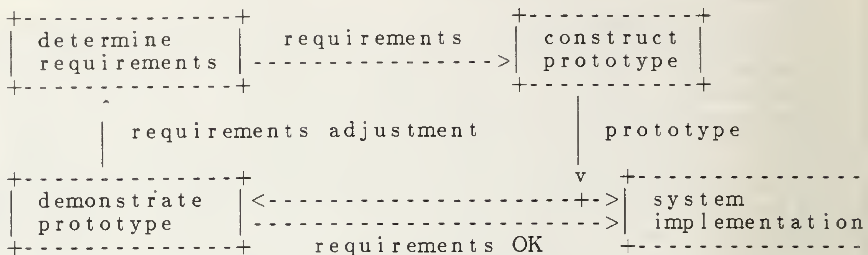
A PSDL prototype is useful for checking real-time requirements because the critical timing constraints and the most important concerns, e.g. maximum execution time, minimum response time, and synchronization, are very hard to validate without actually constructing a valid schedule and observing the execution of the prototype. Most real-time systems are used to monitor and control physical processes external to the computer in an embedded system. The precision and accuracy requirements in the design of a real-time control system complicate the demands on the execution of the designed software system. For these reasons, the design of real-time systems imposes particularly stringent demands on a prototyping language. The formal structure in PSDL specifying the real-time constraints provides a basis for automating the production of code from the formal requirements specifications to the underlying programming language. The execution of PSDL prototypes helps to verify that the design of an embedded system with given timing constraints for the components in the prototype will interact with its environment in a way that meets the timing constraints of the system as a whole. This is important because making a production quality implementation is very expensive, so that it is desirable to check that a design is feasible by using an inexpensive prototype before committing significant resources to an implementation.

2.2.2. The PSDL Prototyping Method

In the rapid prototyping paradigm, the traditional software life cycle used in software design is replaced by a recently proposed alternative life cycle which consists of two phases: rapid prototyping and automatic program generation [5]. Completely automatic generation of programs from very high level specifications is not currently practical. In our approach, program construction is sped up by taking advantage of reusable software components drawn from a software base. The aspects of program construction that benefit from mechanical assistance are retrievals from the software base, generation of code for interconnecting available modules, and static task scheduling.

In rapid prototyping the prototype is used in an iterative process of negotiation. The user describes the requirements, and the analyst interprets them and builds a prototype. The analyst then demonstrates the execution of the prototype to the customer. The requirements are adjusted based on feedback from the customer, and the prototype is modified accordingly until both the customer

and the analyst agree on the requirements. This process is illustrated below.



Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the real needs of the user, increasing reliability and reducing costly requirements changes. PSDL was developed together with a method for rapidly constructing prototypes for large systems with real-time constraints. The purpose of the PSDL prototyping method and its support environment is the rapid construction of executable prototypes for large real-time systems with the following properties.

(1) The prototype must satisfy and be traceable to its requirements. Iterated prototype construction is used to analyze and firm up the requirements for the intended system.

(2) The prototype must be easy to modify. The prototype will be subject to many revisions before the user is satisfied with the requirements as reflected by the behavior of the prototype.

(3) The prototype must be easy to read and analyze. The prototype serves to document an initial design, and to support analysis of the intended system. Clarity and simple high level structures allow designers to easily answer questions about the properties and the feasibility of the intended system based on the prototype.

The goal in constructing a rapid prototype is different than in constructing a production quality software system. Efficient use of designer time and rapid feedback for the user are more important than robust operation, efficient use of machine resources, or completeness.

A problem oriented top-down strategy is used to focus the prototyping effort on critical problems or selected attributes of the entire system. The major system attributes that must be demonstrated to the user usually appear in a critical subsystem. It is necessary to create a quick sketch of the skeleton of the intended system, because the environment of the critical subsystem must be at least partially simulated to demonstrate the behavior of the prototype. This quick sketch can be built rapidly and understood easily by means of a highly interactive graphics editor for PSDL. The essential advantage of rapidly building the sketch of the prototype is that it provides an initial description of the intended system, which can serve as the basis for analysis and negotiation. The prototype system gradually fulfills the requirements during the iterations of the prototyping effort[6]. Our prototyping method enables each update to the prototype to be made quickly and easily.

The PSDL prototyping method results in a hierarchically structured prototype. The method provides a hierarchical decomposition strategy for filling in more details at any level of the prototype design. It also helps the designer to concentrate on the critical subsystems that must be refined to resolve the problems that motivated the rapid prototyping effort. The prototyping method uses stepwise refinement to selectively refine and decompose critical components. Each higher level component is described in terms of lower level ones and the relations between them. The decomposition of each composite component is a realization of the system at a lower level of detail.

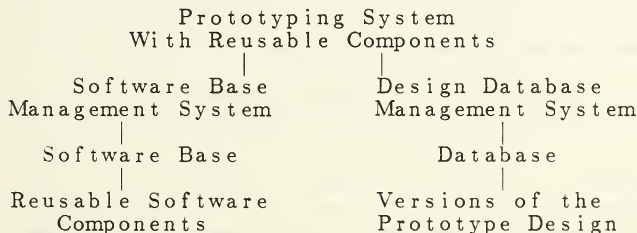
The prototype is designed based on abstract functions, abstract data, and abstract control. This high level view emphasizes the overall configuration at each level without getting bogged down in programming level details. The design is refined by decomposing abstract functions and data types into lower level ones. Functional, data, and control abstractions are used to hide lower level details, effectively carrying out the recommendations in[14]. Control constraints are combined with the data flow model to achieve the best modularity with sufficient control information. Data flow is used to simplify the interactions between modules, eliminating direct external references and communication by means of side effects.

2.2.3. The Rapid Prototyping Environment

An automated support environment is essential for the rapid construction of prototypes. PSDL and its prototyping method have been designed for use in an environment containing a software base management system, an execution support system, a syntax directed editor with graphics capabilities, and a design database.

2.2.3.1. Software Base

The software base management system[13] is responsible for organizing, retrieving, and instantiating reusable software components from the software base, while the design database is responsible for managing the versions and alternatives of the prototype design, as illustrated in the following diagram.



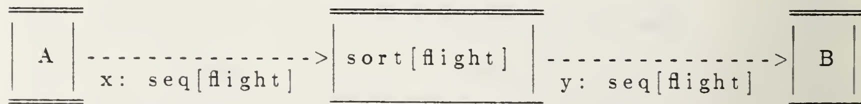
Prototyping Approach

A software base management system should support the retrieval of the set of software components whose specifications match a given template. The reusable

components in the software base are used to realize subsystems of the prototype, and the available reusable components are used to guide the decomposition process by which the behavior of the prototype is refined. PSDL is used to describe the connections between the components of a prototype, and to specify the behavior of the reusable components in the prototype as well as those in the software base. In addition to implementation information, each component in the software base must have a PSDL specification. The PSDL specification is organized as a set of orthogonal attributes. Component retrieval based on partial matches of specified subsets of these attributes must be provided by the software base management system. A browsing capability[15] and a set of operators for tailoring and instantiating generic components[9] should also be provided. The browsing capability is important because it can give the designer guidance on how to decompose a prototype to best take advantage of the available reusable components. The operators are important because it is impossible to explicitly store all possibly relevant variations on each reusable component. We believe it is possible to find a manageable set of generic components and adapting operators sufficient to cover most of the variations needed in rapid prototyping. Establishing this is one of our research goals.

A sufficiently large practical software base containing high quality reusable components is needed. It is important to have a relatively complete set of general purpose components for performing the functions that are common to many systems, such as managing displays, sorting and searching, parsing input strings, and managing lookup tables. Many of these functions can be effectively encapsulated in a relatively small set of abstract data types, such as those described in[15]. It is very important to provide generic versions of the reusable components, since it would otherwise be impossible to design with abstract data types while relying on standard reusable components for performing common utility functions.

The advantages of a set of general purpose utility modules that can easily be used together in various combinations has been well established by the early work on the Unix Programmer's Workbench[16]. This work relied on a single interface format, namely an ascii text file, to ensure compatibility between the various tool interfaces. Such an approach works for applications dominated by text processing, but restricting interfaces to any single data type is unacceptably restrictive for prototyping large software systems. We solve the compatibility problem by requiring the utilities to be polymorphic operators, i.e. parameterized families of closely related operators[17]. An example of a generic reusable component used in this way is shown below.



In this example, A and B are special purpose modules realized by the designer using PSDL, and sort[flight] is an instance of a generic reusable component from the software base. The sort operator can be applied to sequences containing elements of any fixed type that provides a less operation for comparing two instances of the type. The type of the elements in the sequence is specified by a generic parameter, and is bound to the abstract type flight on retrieval from the

software base.

Polymorphic operators can be realized using the generic packages of Ada, where the types of the inputs and outputs are supplied as generic parameters. These operators can be connected to any other components, because the input and output types of the utility operator can be adjusted to fit the interfaces of the modules it will be connected to. Note that generic parameters are required because no fixed set of interface types is sufficient to accommodate all of the abstract data types that might be defined by a designer in future applications, and even a large software base must be finite. This makes Ada a good choice for the underlying programming language.

2.2.3.2. Execution Support System

In order to construct and update a prototype rapidly, the execution support system for PSDL must be efficient. Since prototype modifications are at least as frequent as prototype runs in the expected usage pattern for the execution support system, both preprocessing time and execution time must be given roughly equal weight, making an interpretive implementation strategy preferable to compilation.

The execution support system should be able to save the state of a computation, and to run several alternative versions of a prototype from a given state without repeating the initial part of the computation. This is important because the designer will be engaged in an interactive dialogue with the user, where a given aspect of a prototype's behavior is demonstrated, criticized, and alternatives are explored interactively. Since it may have taken a long user interaction to arrive at the particular state to be examined, it is not acceptable to require the designer and the user to go through many repetitions of that dialogue, or even to incur the delay due to re-running the initial part of the dialogue from a saved script. The need for modifying the prototype in the middle of a run implies the need for a dynamic loader that can be used in the middle of a given execution of the prototype, and for some means for rapidly responding to changed specifications for a component of the prototype. This motivates the need for a high quality software base of sufficient size to accommodate most common variations on system behavior and a powerful software base management system capable of retrieving reusable components efficiently.

The execution support system consists of a static scheduler, a dynamic scheduler, and a debugger. An initial design for these components is described in[18]. The purpose of the static scheduler is to schedule time for the computations with hard real-time constraints in such a way that all of the timing constraints will be guaranteed to be met. We use the standard approach of statically allocating time slots sufficient for the worst case execution times of the operators. The abstract treatment of timing information is an important property of the data flow model since only the essential time orderings among the events in the computation are given. These time orderings act as constraints on the static scheduler, and allow the flexible exploration of schedulers for multiprocessor configurations. The purpose of the dynamic scheduler is to utilize time slots not needed the time critical computations to schedule the computations that do not have hard real-time constraints. The purpose of the debugger is to exercise the prototype, to collect statistics, and to enable the designer to readily modify the prototype to conform to new or modified requirements.

2.2.3.3. Designer Interface

The proposed designer interface consists of a syntax directed editor for PSDL and a graphics tool for constructing and displaying data flow diagrams. The syntax directed editor helps to speed up the process by eliminating syntax errors, automatically supplying keywords, and prompting the designer with a choice of legal syntactic alternatives at each point. The graphics tool is a part of the syntax directed editor, whose purpose is to provide a graphical view of the dataflow diagram part of the PSDL implementation of a composite module. The graphics tool helps the designer visualize the relationships between the components of a decomposition by means of a two dimensional data flow diagram, and provides a convenient way to enter and update the decomposition information in the enhanced data flow diagram, which is part of a PSDL implementation of a component. This capability is important because the text form of a data flow diagram is harder to understand than the graphics form.

2.2.3.4. Design Database

The design database[19] in the prototyping environment contains a PSDL design and a set of requirements. The most important function of the design database is to manage and record the refinements and alternatives that were considered in the prototyping effort. This is especially important in prototyping because it is an exploratory activity, in which insights gained in later refinements often shed new light on the problem and make previously rejected design alternatives look attractive again. The design database should provide facilities for backtracking to previous stages, and for combining different decisions that were made along different alternatives in the development[20].

Using a database rather than a text file also simplifies job of writing programs that analyze PSDL prototypes, and helps to provide a continuous cross referencing capability, by maintaining binary relations between pairs of syntactic objects. Examples of syntactic objects include individual requirements and individual software components. The cross referencing capability is most important for requirements tracing, and is used mostly in updating the requirements and adjusting the prototype to match. The binary relationship relevant to cross referencing is satisfies-requirement. The design database must support retrievals of the forms

(1) given a requirement, find all the PSDL components that realize it, and

(2) given a PSDL component, find all of the requirements it realizes

to effectively support prototype modification.

2.3. Relation to Previous Work

The proposed work is related to earlier work in five main areas: rapid prototyping, design methods, specification and design languages, modeling of real-time systems, and reusable software.

2.3.1. Rapid Prototyping

Prototyping has become increasingly popular in system development[21]. There is a popular branch of rapid prototyping work aimed at database applications[2]. Several approaches are based on programming languages[16,22]. These systems do not link very well to user requirements, and do not address real-time constraints. SREM[23] is a pioneering piece of work on the use of prototypes for validating requirements. This work addresses real-time constraints, machine assisted generation of simulations, and tracing aspects of a simulation to user requirements. However, SREM does not support abstractions and hierarchical decompositions very well. PAISLey[24] addresses the operational specification of real-time systems. Other work on executable specifications has concentrated on transforming specifications into run-time checks for detecting specification violations[25]. This work is still far from the automatic transformation of specifications into running systems.

There has been a fair amount of work on machine aided rapid prototyping for systems without hard real-time constraints.[26] describes a system for prototyping user interfaces for interactive systems.[27] uses Petri nets to prototype the synchronization and interprocess communication aspects of process control systems. While the notation is not very easy to read, it does support automated deadlock detection and performance evaluation in terms of steady state probabilities for graph markings.[28] describes a system for prototyping data processing applications. While they have a tool for determining delay times of modules, they do not describe a method for designing systems with real-time constraints, nor do they support data abstractions, making their approach cumbersome for complex systems.

2.3.2. Design Methods

The important problems facing the computer software industry are achieving cost effective production of software systems and increasing the quality of software products with respect to meeting user requirements. Many software development methodologies have been proposed to approach this goal. Most of the well-known ones, such as Object Oriented Design (OOD)[29,30] and the Jackson System Development Method (JSD)[31,32] more or less depend on the skill of individual designers at the level of manual work.[32] describes a technique for modeling real world systems which is appropriate for typical data processing applications. This method does not address real-time constraints and is weak on data abstractions. These methods are labor intensive, and are too informal to guarantee any quality standards for the resulting design. They are unlikely to lead to any significant improvements in the reliability of software products.

Newly proposed software tools for software design, such as GENESIS[33], SREM[23], the PAISLey operational approach[24], and DIANA[34] go one step further. Most of them are really software development environments consisting of many software tools for computer-aided software development. Some of them are designed to fit specific needs, e.g. DIANA is a tool for the design of Ada systems. Even if we consider only the prototyping aspects of these tools, it is clear that complete automation of software development is still a distant goal. Most of these types of tools cannot be extended to the point of complete automation

because of the lack of mathematical formalization in the related theoretical fields of software engineering.

Two kinds of software system decompositions have been identified[35,36], one based on data flow and the other based on control flow.[35] suggests circumstances in which each of the two kinds of decomposition is preferable and give some restrictions sufficient to guarantee that the computed results are independent of scheduling decisions, but does not address real-time constraints. The method amounts to choosing either a data flow decomposition or a control flow decomposition at each level, depending on the circumstances. It is difficult to follow all the confusing restrictions given in these papers and to understand the overall idea since they do not provide a good computational model and their system is intended for lower level applications.

2.3.3. Specification and Design Languages

A prototyping language must have the characteristics of a good design language, because the structure of a prototype must be understandable and easy to modify. Early design languages[37,38[were not executable, although more recent work has promise in this direction[39]. These languages do not support real-time constraints or requirements tracing. Some design languages address the design and specification levels, but are not executable[40]. Languages for specifying real-time systems have also been investigated[41-43].

A number of non-procedural programming languages have been proposed[43,44]. These languages have the advantage of being easy to analyze. and of exposing the natural parallelism in an algorithm. The design of the non-procedural control constraints of PSDL owes much to these ideas. One difference between our work and previous approaches to rapid prototyping using applicative languages[24,45] is that we provide a black box specification for each component in addition to a non-procedural implementation. Black box specifications are important because they separate required properties of the prototype from the incidental ones, and because they can be used for retrieving reusable components from a software base.

Many informal versions of data flow diagrams[46,47] have been used extensively to model the data transformation aspects of software systems. Data flow diagrams are easy to read, revealing the internal structure of a process and the potential parallelism inherent in a design. The automatic drawing of data flow diagrams is a practical step towards design automation using data flow diagrams[48]. The use of a convenient graphical form makes data flow attractive to many designers because it reveals the structure of the design in an easily understandable form. We believe an automated prototyping environment should provide graphical capabilities for displaying and updating the system structure of the prototype.

However, these informal notations do not provide a unified mechanism to represent all of the relevant attributes of software systems (e.g. timing and control[49]) and are not sufficiently formal to be executable. A more precise model of a data flow computation has been developed in the context of hardware design[50]. We have extended the model and the notation to include control aspects and critical timing constraints in a two dimensional data flow diagram without losing its natural benefits. These extensions are needed for the design of systems with hard real-time constraints.

2.3.4. Real-Time System Modeling

Attempts to use data flow diagrams for modeling real-time systems[49] have resulted in complicated low level models that reflect only qualitative rather than quantitative information. Some of the work on modeling real-time systems has focused on the scheduling problems associated with real-time constraints[11,51]. These results are important for execution of PSDL prototypes. The application of these ideas to PSDL is described in[18]. In[52], an analysis of hard real-time systems as well as scheduling policies for a single processor are provided.

2.3.5. Reusable Software and the Software Base

Methods for enhancing the reusability of software[5,53] are important for managing the software base[9,13], which is one of the building blocks used in our work.

2.4. General Work Plan

The proposed directions for research include better techniques for retrieving reusable software components from a software base and more efficient approaches for implementing the execution support system.

2.4.1. Software Base Retrievals

Better methods for organizing and retrieving reusable components from the software base are important because the software base can effectively speed up the prototyping effort only if the fraction of successful retrievals is relatively high. We propose to investigate software base organizations based on adaptive generalization hierarchies, reusable component retrieval based on specifications with a semantic canonical or normal form, and techniques for combining program synthesis with software base retrievals based on partial matches.

Generalization hierarchies have been found to be effective for supporting browsing tools [15], and have also been effective as knowledge representation tools in artificial intelligence applications. We believe that generalization hierarchies are useful for two different purposes in the context of software base retrievals. First, a properly designed generalization structure should be a significant aid for interactive retrieval of components, especially for the process of looking for relevant components before deciding how to decompose a composite PSDL operator. Second, the hierarchy can be used to provide a basis for approximate retrievals when there is no exact match in the database.

Generalization per category is a database organization principle that was recently developed for organizing parts libraries in VLSI design[54]. Generalization per category differs from conventional generalization because the specializations of a general concept are disjoint and indexed by values of a categorical property. We believe that a disjoint categorization for reusable software components would be a very valuable aid to understanding and organizing a software base, and that such an organization is an important first step towards computer aided retrievals of the set of components relevant to decomposing a

composite operator. Working out such a categorization is therefore one of the goals of our proposed research. It has been conjectured[19] that a categorization with overlapping categories is a sign of a missing categorical property, and that it is always possible to find a natural categorical property that will rearrange an overlapping categorization into a disjoint one. No counter-examples to this conjecture have been found, leading us to believe that it should be possible to find a disjoint categorization for software components. We also believe that working out such a categorization should provide some insights for constructing an algorithm for automatically retrieving a useful approximation to the set of relevant components, which is another goal of the proposed research. Generalization per category induces a lattice structure which can be exploited for efficient database organization. We propose to investigate the application of this structure to the design of an architecture for the software base.

A special purpose prototyping language such as PSDL must be simultaneously suitable for component specification and for use as a query language in the software base. These are conflicting requirements, because a specification language that is easy for people to use will have a rich set of primitives, which allow many syntactically different ways to describe the same function, while software base (or database) retrievals work best in situations where each individual has a unique identifier or key. We propose to investigate specification formalisms with effectively computable canonical forms for specifications. Advances in this area would simplify the organization and accessing methods of a software base as follows. The specifications of each component would be transformed to canonical form before being entered into the software base. This can be visualized as a process of mechanically reducing the specification to a unique simplest form. The template for a query would also be transformed into canonical form before the physical retrieval is performed. This scheme would guarantee that a perfect match retrieval would always succeed if an appropriate reusable component was available, because it would eliminate the possibility that a retrieval would fail to find a module because it was specified in a way that is syntactically different from the retrieval template but semantically equivalent to it. Normal forms will also be investigated. Since a normal form is not necessarily unique, retrieval failures would not be completely eliminated by reducing specification and queries to normal form, but the hit ratio should be significantly improved. We propose to determine if canonical form specifications are practical, and if not, to seek normal forms with good hit ratios.

Another area for investigation is the automated synthesis of a component in cases where the component is not available in the software base, but a small set of primitives that can be combined to form the required module is available. Limited logical inference techniques combined with heuristics for limiting the size of the space of building blocks will be investigated in the later stages of the proposed research, with attention to the use of the generalization hierarchy for limiting the size of the search space.

2.4.2. Implementation Techniques for PSDL

We also propose to investigate efficient methods for implementing flexible interpreters with restarting checkpoints. The efficiency criteria for a prototyping language are different than those for a programming language because modifications are frequent, and it is desirable to be able to run several different

modifications from the same point midway through a demonstration session. These considerations favor an interpretation strategy over a compilation strategy. A strategy utilizing partially compiled and partially interpreted components, together with guidelines for triggering the automated compilation of components that are heavily used is promising. We propose to develop an experimental translator for PSDL, and to develop and evaluate several different implementation methods. This process will be aided by an attribute grammar based translator generator[4].

One of the techniques currently used for meeting tight real-time constraints is multiprocessing. We propose to extend PSDL and the initially developed implementation techniques to multiprocessor implementations. The most important extension needed in the language is related to faithfully modeling scheduling constraints imposed by the implementation structure for the external systems interacting with the real-time software. For example, it may be that two of the external systems are allocated to the same hardware and must have non-overlapping executions, or that the scheduling of some of the external events is already fixed, so that it must be treated as a constraint on the design of the real-time software rather than as a quantity to be derived in the prototyping process. Facilities for expressing this kind of information must be developed and integrated into the execution support system. This will be done by investigating real-time scheduling algorithms for multiple processor systems that respect the above mentioned constraints.

References

1. R. T. Yeh, *Software Engineering,* IEEE Spectrum (NOV 1983).
2. J. Connell and L. Brice, "Rapid Prototyping," pp. 93-100 in *Datamation*, (AUG 1984).
3. L. Levy, "A Metaprogramming Method and Its Economic Justification," *IEEE TSE* SE-**12**(2) pp. 272-277 (FEB 1986).
4. R. Herndon and V. Berzins, *The Realizable Benefits of a Language Prototyping Language,* to appear in IEEE TSE (1987).
5. R. T. Yeh, R. Mittermeir, N. Roussopoulos, and J. Reed, "A Programming Environment Framework Based on Reusability," *Proc. Int. Conf. on Data Engineering,* (APR 1984).
6. Luqi, "Rapid Prototyping for Large Software System Design," Ph.D. Thesis, University of Minnesota (1986).
7. Luqi and V. Berzins, "Rapid Construction of PSDL Prototypes," TR-86-17, Computer Science, University of Minnesota (1986).
8. Luqi and Valdis Berzins, "Rapid Prototyping of Real-Time Systems," *Revised for IEEE SOFTWARE,* (1987).
9. N. Roussopoulos, "Architectural Design of the SBMS," Quarterly Report for the STARS SB/SBMS Project, DCS, UNIV of Maryland (APR 1985).
10. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *to appear in IEEE TSE,* (1987).
11. A. K. Mok, *The Design of Real-Time Programming Systems Based on Process Models,* IEEE (1984).
12. J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation," *CACM* **21**(12)(1978).

13. R. T. Yeh, N. Roussopoulos, and B. Chu, "Management of Reusable Software," *Proc. COMPCON,* pp. 311-320 (SEP 1984).

14. David Parnas, "On the Criteria to be Used in Decomposing a System into Modules," *CACM* **15**(12) pp. 1053-1058 (DEC 1972).

15. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation,* ADDISON (1983).

16. E. L. Ivie, "The Programmer's Workbench - A Machine for Software Development," *CACM* **20**(10) pp. 746-757 (OCT 1977).

17. J. A. Goguen, "Parameterized Programming," *IEEETSE* **SE-10**(5) pp. 528-543 (SEP 1984).

18. Luqi and V. Berzins, "Execution Aspects of Prototypes in PSDL," TR 86-2, University of Minnesota (1986).

19. Mohammad Ketabchi, "On The Management of Computer Aided Design Databases," Ph.D. Thesis, University of Minnesota (1985).

20. Valdis Berzins, "On Merging Software Extensions," *Acta Informatica* **23** pp. 607-619 (1986).

21. , "Special Issue on Rapid Prototyping," *Software Engineering Notes* **7**(5) pp. 3-184 ACM SIGSOFT, (December, 1982).

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

23. M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEETSE* **SE-3**(1) pp. 60-68 (JAN 1977).

24. P. Zave, "An Operational Approach to Requirements Specifications for Embedded Systems," *IEEETSE* SE-8(3) pp. 250-269 (1982).

25. D. Luckham and F. W. von Henke, "An Overview of Anna, a Specification Language for Ada," *IEEE Software* **2**(2) pp. 9-22 (MAR 1985).

26. A. Wasserman, P. Pircher, D. Shewmake, and M. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology," *IEEETSE* **SE-12**(2) pp. 326-345 (FEB 1986).

27. G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEETSE* **SE-12**(2) pp. 346-357 (FEB 1986).

28. J. Tseng, B Szymanski, Y. Shi, and N. Prywes, "Real-Time Software Life Cycle with the Model System," *IEEETSE* **SE-12**(2) pp. 358-373 (FEB 1986).

29. Grady Booch, *Software Engineering with Ada,* Benjamin/Cummings, Menlo Park (1983).

30. Grady Booch, "Object-Oriented Development," *IEEETSE* **SE-12**(2) pp. 211-221 (FEB 1986).

31. M. A. Jackson, *Principles of Program Design,* ACADEMIC, New York (1975).

32. J. R. Cameron, "An Overview of JSD," *IEEETSE* **SE-12**(2) pp. 222-240 (FEB 1986).

33. C. V. Ramamoorthy, Y. Usuda, W. Tsai, and A. Prakash, "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software," *Proc. COMPSAC 85,* pp. 472-479 (1985).

34. A. Evans, K. J. Butler, G. Goos, and W. A. Wulf, *DIANA Reference Manual,* Tartan Laboratories Inc., Pittsburgh, PA (1983).

35. O. Shigo, K. Iwamoto, and S. Fujibayashi, "A Software Design System Based on a Unified Design Methodology," *Journal of Information Processing* **3**(3) pp. 186-196 (SEP 1980).

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

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

38. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, ADDISON (1979).

39. T. Cheatham, J. Townley, and G. Holloway, "A System for Program Refinement," pp. 198-214 in *Interactive Programming Environments*, McGraw-Hill (1984).

40. F. W. Beichter, O. Herzog, and H. Petzsch, "SLAN-4 A Software Specification and Design Language," *IEEETSE* **SE-10**(2) pp. 155-162 (MAR 1984).

41. V. H. Haase, "Real-Time Behavior of Programs," *IEEETSE* **SE-7**(5)(SEP 1981).

42. G. Luckenbaugh, "The Activity List: A Design Construct for Real-Time Systems," Master's Thesis, DCS, UNIV of Maryland (1984).

43. A. A. Faustini and C. B. Lewis, "Toward a Real-Time Dataflow Language," *IEEE Software* **3**(1) pp. 29-35 (JAN 1986).

44. W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*, ACADEMIC (1985).

45. P. Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping," *IEEETSE* **SE-12**(2) pp. 241-250 (FEB 1986).

46. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press (1978).

47. E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall (1979).

48. C. Batini, E. Nardelli, and R. Tamassia, "A Layout Algorithm for Data Flow diagrams," *IEEETSE* **SE-12**(4) pp. 538-546 (APR 1986).

49. P. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEETSE* **SE-12**(2) pp. 198-210 (FEB 1986).

50. J. B. Dennis, G. A. Boughton, and C. K. C. Leung, "Building Blocks for Dataflow Prototypes," in *Proc. Seventh Symposium on Computer Architecture*, La Baule, France (MAY 1980).

51. A. K. Mok, "The Decomposition of Real-Time System Requiremennts into Process Models," *IEEE Proc. of the 1984 Real Time Systems Symposium*, pp. 125-133 IEEE, (DEC 1984).

52. Abha Moitra, *Analysis of Hard Real-Time Systems*, Computer Science Department, Cornell University (1985).

53. B. Leavenworth, "ADAPT: A Tool for the Design of Reusable Software," RC 9728, IBM Watson Research Center, Yorktown Heights, NY (1982).

54. Mohammad Ketabchi and Valdis Berzins, "Generalization Per Category: Theory and Application," *Proc. Int. Conf. on Information Systems*, (1986). also TR 85-29, Computer Science Dept., University of Minnesota

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	50
Chief of Naval Research Arlington, VA 22217	2

1228649

5 6853 01057732 3
0228649