1988

Software Evolution Via Prototyping

Luqi

Naval Postgraduate School

T.31

NPS52-88-039

NAVAL POSTGRADUATE SCHOOL

Monterey, California

SOFTWARE EVOLUTION VIA PROTOTYPING

Luqi

September 1988

Prepared for :

Navel 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 in part by the National Science Foundation and the Naval Postgraduate School Research Foundation.

This report was prepared by:

[redacted]
LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]
ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]
KNEALE T. MARSHALL
Dean of Information
and Policy Science

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) NPS52-88-039	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA. 93943	7b. ADDRESS (*City, State, and ZIP Code*) Washington DC 20550
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct funding
8c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA. 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

SOFTWARE EVOLUTION VIA PROTOTYPING (U)

12. PERSONAL AUTHOR(S)
LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Progress	FROM 87/10 TO 88/09	1988 Sept	30

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	specification, rapid prototyping, computer aided software engineering, real-time embedded system, ada, software design

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

Rapid prototyping is widely accepted as an alternative methodology for software deveolpment. The problems of software maintenance are magnified in rapid prototyping because prototypes are subject to frequent and repeated changes. The concepts and mechanisms presented in this paper support such changes in rapid prototyping based on component specifications. We discuss the following important issues for software evolution via prototyping: (1) explicit interactions between prototype components for easily determining the impact of a proposed change, (2) requirements tracing facilities for identifying the parts of a prototype affected by a proposed requirements change, (3) structured system construction by maximizing reusability of software components, and (4) the use of specifications in retrieving, composing, and adapting reusable components in minimizing effort for code analysis and modification in software maintenance.

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

DD FORM 1473, 84 MAR

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

SECURITY CLASSIFICATION OF THIS PAGE

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

Software Evolution via Prototyping

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Rapid prototyping is widely accepted as an alternative methodology for software development. The problems of software maintenance are magnified in rapid prototyping because prototypes are subject to frequent and repeated changes. The concepts and mechanisms presented in this paper support such changes in rapid prototyping based on component specifications. We discuss the following important issues for software evolution via prototyping: (1) explicit interactions between prototype components for easily determining the impact of a proposed change, (2) requirements tracing facilities for identifying the parts of a prototype affected by a proposed requirements change, (3) structured system construction by maximizing reusability of software components, and (4) the use of specifications in retrieving, composing, and adapting reusable components in minimizing effort for code analysis and modification in software maintenance.

1. Introduction

Software evolution is a most important issue in software development because changes to existing systems account for more than half of the total software cost. Evolution corresponds to the maintenance phase in traditional software life cycle, and consists of the process of firming up the requirements and adapting the software system to maintain the correspondence between the two. It is difficult to eliminate this expensive process because later phases of development often bring more knowledge and more insight into the problem domain and the properties of the intended system to the designers and customers, but the need for some types of changes can be reduced [12]. Software systems are changed for the following reasons:

Requirements errors

The developers have incorrectly understood the requirements and have produced a system that does not meet user needs.

Implementation errors

A faulty design or implementation does not correspond to the specification.

Phased delivery

A partial implementation has been delivered because the customer cannot wait until a complete implementation is available.

User education

Customer requirements have changed because experience with the current version of the system has changed their perception of how computers can be used to solve their problems.

New situations

Changes in the environment of the system have introduced new requirements. Examples of such changes are new external systems, new policies, new technologies, and new competitive pressures.

Prototyping can help reduce the need for unplanned changes by stabilizing the requirements before a significant amount of effort has been invested in implementation. A key problem in large scale software development is the need for communication between people with different areas of expertise. Typically customers know much more about the problem domain than they do about programming, while the programmers know much more about programming than they do about the customer's problems. Both the problem domain and the programming domain have many specialized concepts and terms, many of which are unfamiliar to people who are not experts on the domain. The requirements for the proposed software system are influenced by constraints from both the problem domain and the programming domain that cannot be completely understood without knowledge of specialized concepts from both domains. Requirements are difficult to construct and validate because usually there is no single person who understands all of the constraints on the proposed system. This is especially evident in large systems with hard real-time constraints, since the requirements for such systems are generally very difficult to understand or describe.

A prototype is a concrete executable model of selected aspects of the proposed system. Prototypes are valuable aids in requirements analysis because they can be used to demonstrate the behavior of the proposed system in a form that can be readily understood by all concerned parties. Prototypes can help customers visualize and test consequences

of their requirements and provide an effective basis for communication between the customers and the requirements analysts. The process of constructing a prototype also helps the analysts to determine what questions they need to ask to construct a conceptual model of the problem domain that is sufficiently complete to be used in designing the proposed system.

Prototyping can help reduce maintenance costs primarily by reducing requirements errors, so that fewer changes to the software are needed. This applies both to the original formulation of the system and for evolutionary changes sparked by user education or new situations. Phased delivery has traditional been the main mechanism for inducing requirements changes due to user education. Extensive exercising of a prototype by a group of users can trigger some of the requirements changes due to this effect before the production version of the system has been produced. This can alleviate wasted design effort, which is one of the main problems with phased delivery. Phased delivery is usually accomplished by developing a design for the whole system, and then choosing a subset for implementation and delivery in the first release. If the requirements changes due to user education are severe, the design for the rest of the system can be invalidated before it is ever implemented.

Prototyping can also help reduce implementation errors. The prototype can allow more extensive testing of a system by providing a means for evaluating test results. The output of the production code can be mechanically compared to the corresponding output of the prototype, thus allowing more test cases to be examined without increasing the amount of human effort involved. The prototype can also be used by implementors to resolve questions about the intended behavior of the system in particular cases. This can

reduce the incidence of errors caused by programmers making plausible but unfounded assumptions about unspecified cases. Such assumptions are often needed in traditional software development because specifications are incomplete and schedule pressures do not allow queries to the customer about every minor detail of system behavior. Implementation errors can also be reduced by systematic development techniques supported by formal methods and automated tools [1].

For large systems, every adjustment to the requirements has to be recorded and incorporated into the system specifications, as well as the architecture and the implementation. A systematic way to reduce the extraordinary effort required for software maintenance is to manage the changes in the system needed to reflect adjustments to the requirements at the specification level rather than the implementation level. We can view these two aspects of software maintenance in a unified way if we can mechanically transform specifications into code. While mechanical transformations from black-box specifications into production quality code is not practical at the current time, computer-aided generation of prototype implementations is both feasible and useful [3].

1.1. The Prototyping Life Cycle

The traditional software life cycle consists of a series of phases which yield runnable software only late in the process. One view of the traditional life cycle is illustrated in Fig. 1. A major problem with the traditional approach is that there is no guarantee the resulting product will reliably solve the customer's problem. Often users will be able to indicate the true requirements only by observing the operation of the system, and the traditional life cycle yields executable programs late in the process, when too much money has already been spent and there is no time left to recover from requirements

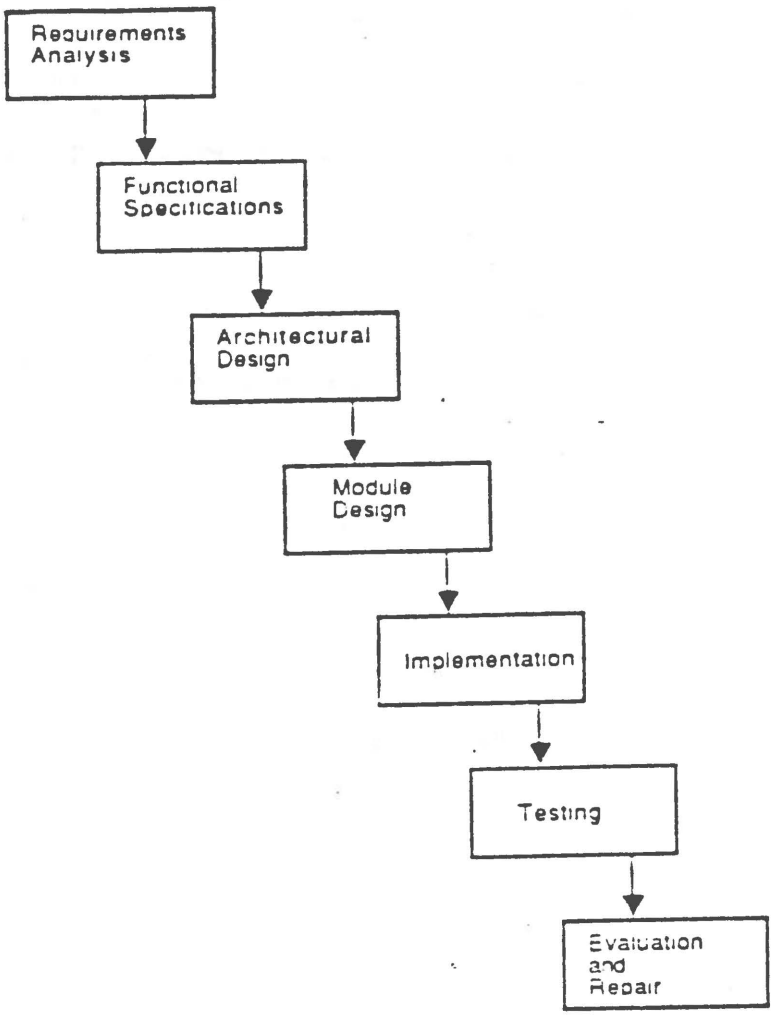

Fig. 1 Traditional Software Life Cycle

errors.

When this traditional life cycle approach is applied to hard real-time or embedded systems, the potential for inconsistencies increases. One of the major differences between a real-time system and a conventional computer system is the required precision

and accuracy of the application software. The response time of each individual operation may be a significant aspect of the associated requirements, especially for operations whose purpose is to maintain the state of some external system within a specified region, as is common in embedded software systems. In hard real-time systems response times are a critical determining factor in the accuracy of the software. These response times, or deadlines, must be met or the system will fail to function correctly, with potentially catastrophic consequences. For example, as part of a larger computer system, the requirements for an embedded system can incorporate stringent real-time constraints, parallel processing on multiple computers, and a high degree of reliability. These requirements will often exceed the intellectual capacity of a single software engineer, requiring several individuals working independently on different segments of the system. In such cases the requirements can be very difficult to understand.

Current research suggests a revised software development life cycle, which consists of two phases, rapid prototyping and automatic program generation [8]. This prototyping life cycle is an alternative to the traditional life cycle which has been proposed to alleviate problems stemming from incorrect requirements, especially when designing hard real-time systems. Although current capabilities preclude completely automatic program generation, the required software tools and capabilities do exist for computer-aided rapid prototyping. As a software methodology, rapid prototyping provides the user and designer with a fast, efficient and easy-to-use stepwise process. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specifications and initial design before valuable time and effort are expended on implementation software. Fig. 2 graphically describes this methodology as a feedback loop [8]. Rapid prototyping initially establishes an iterative process between

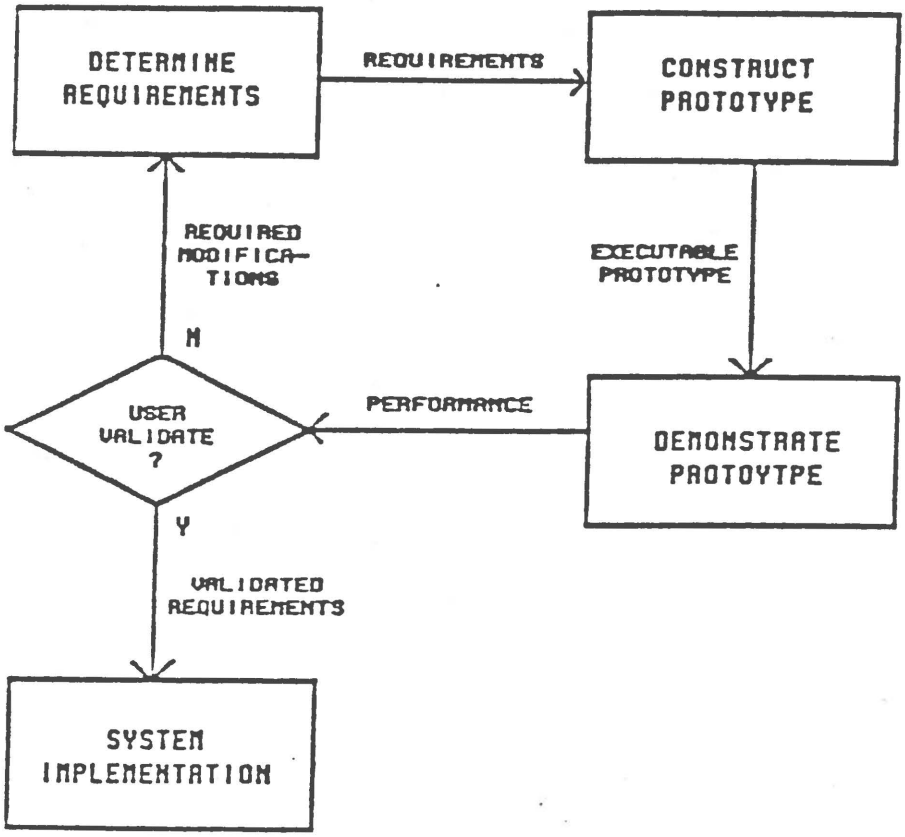

Fig. 2 Prototyping Life Cycle

the user and the designer to concurrently define specifications and requirements for the time critical aspects of the envisioned system. The designer then constructs a model or prototype of the system in a high-level, prototype description language. This prototype is a partial representation of the system, including only those critical attributes necessary

for meeting user requirements, and is used as an aid in analysis and design rather than as production software. During demonstrations of the prototype, the user validates the prototype's actual behavior against its expected behavior [9]. If the prototype fails to execute properly or to meet any critical timing constraints, the user identifies required modifications and redefines the critical specifications and requirements. This process continues until the user determines that the prototype successfully meets the time critical aspects of the envisioned system. Following this validation, the designer uses the validated requirements as a basis for the design of the production software.

Computer-aided rapid prototyping further refines the efficiency and accuracy of this new methodology. While utilizing the same iterative approach, computer-aided rapid prototyping relies on software tools which assist the designer in constructing and executing the prototype. We have designed a computer-aided prototyping system (CAPS) to provide an integrated set of tools to support prototyping of complex software systems which may include hard real-time constraints [7]. These tools operate on the prototyping language PSDL (Prototype System Description Language) [10]. This language has been designed to support the needs of rapid prototyping by providing a high level description of the system which can be used to demonstrate the behavior of the prototype by means of the above software tools. Since requirements are especially difficult to define for large systems with hard real-time constraints, PSDL has been designed to apply to such systems.

Prototyping is an iterative process which depends on the ability to rapidly adjust the behavior of the prototype based on feedback from the customer or user. The problems of software maintenance are magnified in rapid prototyping because prototypes are subject

to frequent and repeated changes. The goal of the prototyping life cycle is to shift a sizable part of the initial maintenance activity from the production software to the prototype. The potential benefits to be gained from prototyping depend critically on the ability to modify the behavior of the prototype with substantially less effort than that required to modify the production software. The purpose of PSDL and the associated software tools is to provide this ability. The mechanisms by which this is achieved are described below.

1.2. Mechanisms to Support Modifications

The prototyping language PSDL approaches the requirement to support frequent design modifications by means of the following subgoals.

Modularity

The language must make it easy for the system designer to create a prototype with a high degree of module independence and to preserve its good modularity properties across many modifications.

Simplicity

The language should be simple and easy to use.

Reuse

The language should be suitable for specifying the retrieval of reusable modules from a software base.

Adaptability

The language should support small modifications to the behavior of a module without the need to examine its implementation.

Abstraction

The language should support a set of abstractions suitable for describing complex software systems with real-time constraints.

Traceability

The language should support requirements tracing.

Good modularity is essential for achieving ease of modification. An experimental study shows that many of the problems with correctly performing software modifications are due to interactions between widely separated pieces of code [4]. Locality of information was an important design goal of PSDL. The underlying computational model was chosen to make all interactions between components explicit. This model supports a system decomposition criterion that combines data flow and control flow considerations [8].

PSDL is simple and easy to use because it contains a small number of powerful constructs. Designs are described in PSDL as networks of operators connected by data streams. Such networks can be represented as dataflow diagrams augmented with timing and control constraints. The user interface uses the diagrams to provide a convenient means for presenting the system structure to the designer. The operators in the network can be either functions or state machines. The data streams can carry exception conditions in addition to values of arbitrary abstract data types.

PSDL supports reusable components by means of black-box specifications suitable for retrieving modules from a software base. The specification part of a PSDL component contains several attributes which describe the interface and behavior of the component. These attributes can be used to automatically generate a uniform specification for the reusable component [6]. These uniform specifications are used both for retrieval

of reusable components and for organizing the software base.

PSDL supports small modifications to modules by means of control constraints. Control constraints can be used to impose preconditions on the execution of a module, to add filters to the output of a module, to suppress or raise exceptions in specified conditions, and to control timers. These facilities allow small modifications to the behavior of a module which do not involve the internal implementation of the module.

PSDL provides abstractions suitable for describing large systems and real-time constraints. These include the non-procedural control constraints mentioned above, timing constraints, timers, functional abstractions, and data abstractions. Timing constraints can be used to associate hard real-time constraints with operators. Examples of timing constraints include the maximum execution time, the maximum response time, and the minimum calling period. Timing constraints implicitly determine when operators with hard real-time constraints will be executed. This simplifies the designer's view of the prototype by removing explicit scheduling considerations from the design of the prototype system. Timers are used to control aspects of behavior that depend on the duration of particular system states or classes of system states. Timers allow static descriptions of durational timing constraints, allowing the designer to ignore the operational details involved in their implementation. A rich set of functional and data abstractions are provided by the pre-defined part of the software base. The designer can define additional functional and data abstractions, either by adding them to the software base or by defining them in terms of more primitive abstractions using PSDL.

PSDL supports requirements tracing by means of a construct for declaring the requirements associated with each part of the prototype. Requirements tracing is impor-

tant because the prototype must be adapted to the changing perceptions of the requirements resulting from demonstrations of prototype behavior. The links between each requirement and the parts of the prototype realizing the requirement are used to determine which parts of the prototype must be modified when a requirement is changed or dropped. In order to prevent the structure of the design from being corrupted by multiple modifications, it is important to remove parts of the code that are no longer supported by an updated set of requirements. This cannot be done safely unless the correspondence between the requirements and the code is recorded and kept up to date. In situations where this correspondence is not maintained, each change to the system results in the addition of new code, without the removal of any old code. Such a process leads to increasingly complex systems that eventually escape from human control, making removal of old code essential for systems that will be changed many times. The facilities for recording requirements trace information in PSDL are used by software tools in CAPS to provide automated aid in maintaining and using this information.

1.3. Benefits of Prototyping

Prototyping allows an appreciable part of the maintenance activity to be carried out in terms of the prototype rather than in terms of the production code for the intended system. This is useful because the prototype description is significantly simpler than the production code, is expressed in a notation tailored to support modifications, and the software tools in the computer-aided prototyping environment can be used to help carry out the required modifications.

Rapid prototyping is also a useful tool in feasibility studies, for reducing project risks and estimating costs. Prototypes of critical subsystems or difficult parts of a com-

plicated 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, eg. database design, the metaprogramming method and others, have substantiated the expected cost relationships between the prototype and the completed system [5]. 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, eg. timing constraints, input and output formats, or interfaces between modules.

2. Computer-Aided Prototyping System

The main software tools in the computer-aided prototyping system are shown in Fig. 3. These tools communicate by means of the PSDL language, which serves to integrate the tools and provide a uniform conceptual framework for the prototype designer.

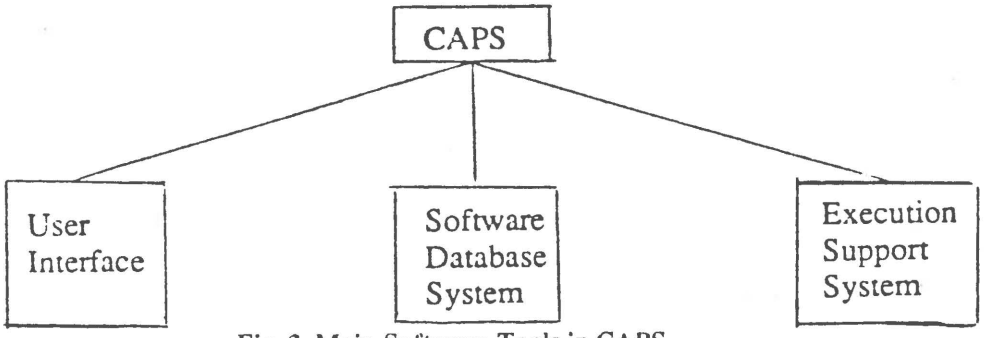

Fig. 3 Main Software Tools in CAPS

The user interface consists of a syntax-directed editor with graphics capabilities, an expert system for communicating with end-users, a browser, and a debugger. The editor enables convenient entry of PSDL descriptions into the system while preventing syntax errors. It also provides support for displaying graphical summary views of the prototype, maintaining the requirements trace, and locating parts of the prototype design related to particular requirements or data streams. The expert system provides a paraphrasing capability which generates English text from PSDL descriptions, to allow end-users to directly examine the prototype without the need for familiarity with PSDL. The browser allows the designer to interact with the software database. In particular the browser provides facilities for retrieving and examining reusable components stored in the software database system. The debugger allows the designer to interact with the execution support system. In particular, the debugger provides facilities for initiating execution of the prototype, displaying results or trace information, and gathering statistics about prototype behavior and performance.

The software database system consists of a design database, a software base, a software design-management system, and a rewrite subsystem. The design database contains the PSDL prototype descriptions for each software development project using CAPS. The software base contains PSDL descriptions and code for all available reusable software components. The software design-management system is responsible for managing and retrieving the versions, refinements, and alternatives of the prototypes in the design database and the reusable components in the software base. The rewrite subsystem translates PSDL specifications into a normalized form that is used by the design-management system for retrieving reusable components from the software base [6].

The PSDL execution support system contains a translator, static scheduler, and a dynamic scheduler [9]. The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams, control constraints, and timers. The static scheduler allocates time slots for operators with real-time constraints before execution begins. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints as execution proceeds.

3. Using CAPS for Maintenance

If the prototyping process is carried out manually, the associated benefits are limited because it takes too much effort. CAPS can increase the leverage of the prototyping strategy by reducing the effort that must be spent by the designer in producing and adapting a prototype to perceived user needs. This section describes how the facilities provided by CAPS can be used to assist in the maintenance activities involved in the prototyping process.

In the prototyping life cycle shown in Fig. 2, the maintenance activity for the prototype starts after the cycle has been carried out once: the analysts have determined the initial requirements by talking to the customers, constructed an initial prototype, and demonstrated it to the customer, who finds some aspects of the prototype's behavior unacceptable and requests some modifications. The process of demonstrating the prototype is aided by the user interface, which has facilities for presenting the results of prototype execution to the customer and for guiding the choice of which aspects of the prototype to demonstrate. The latter function is accomplished by an embedded expert system

containing heuristics for exercising prototypes and a facility for recording test case coverage information. The analysts use customer feedback about the behavior of the prototype to modify or refine the requirements, which are maintained as a tree of subgoals. The incremental change leading to the new version of the requirements is entered into the system. At this point the facilities provided by CAPS are used to adapt the prototype to the new requirements.

The user interface helps the prototype design team identify the tasks that have to be carried out to update the prototype. The user interface maintains a list of unresolved new requirements and a list of unresolved modified requirements. Whenever a member of the design team is ready for a new task, the system presents the lists and lets the designer pick an item to resolve. If a modified requirement is chosen, the interface returns a list of modules previously supporting the requirement, and lets the designer check them off as they are adapted or determined to be still valid. The effort required to do this task coordination is minimized by presenting the lists as menus, and allowing the designer to pick items by means of a pointing device. Choosing an item results in a summary view of the module, which can be browsed and updated as required.

The user interface speeds up the process of adapting the prototype by

(1) Helping to coordinate tasks performed by a team of designers,

(2) Helping to focus the designer's attention on the information relevant to a task,

(3) Providing summary views of the system or selected components, and

(4) Locating all potentially relevant parts of the prototype.

The components of the software database actively contribute to the process of adapting the prototype to new requirements. The software design-management system

helps to maintain the design history and to locate relevant reusable software components. The design history consists of the relationship between each version of the requirements and the corresponding versions of parts of the prototype. This information is useful because sometimes the customer will retreat to previous versions of the requirements. Situations in which this may happen include cases where the customer gives up on an ambitious requirement in response to cost or performance estimates resulting from examination of the prototype. In such cases parts of the requirements are returned to previous configurations, and the system can help to restore the corresponding parts of the prototype to their previous configurations.

The design database also provides concurrency control functions which allow multiple designers to update the parts of the prototype without risk of unintentional interference. In the interests of minimizing delay, the design database will not lock out access to any part of the design, even while the design is being updated. Instead, the system will allow the previous version of the component to be examined, with a warning that a new version is currently in preparation. The system will provide information about the reason the component is being modified (i.e. some particular new or modified requirement) on request.

The software base provides reusable software components matching given PSDL specifications. In the PSDL prototyping method [8] modules are realized by three main mechanisms:

(1) Retrieval of a suitable component from the software base. The software base should contain flexible generic modules, whose parameters are determined as part of the retrieval process. It also should contain rules for matching a

specification by means of a composite operator, which is realized by a network of operators, at least one of which must be an available reusable component [11]. The retrieval mechanism is therefore capable of performing some routine aspects of bottom-up design, freeing the designer from the need to be familiar with all of the reusable components in the software base.

(2) Decomposition of the component into a network of simpler components. This is done by the designer if the component cannot be retrieved directly from the software base, and the component is sufficiently complex to have a useful decomposition into simpler parts. The designer is responsible for top-down design activities such as inventing new abstractions. Each of the identified parts is specified in PSDL and realized by the same set of mechanisms, applied recursively.

(3) Direct implementation in a programming language. This is done by the designer if the component cannot be retrieved directly from the software base, and the component does not have a useful decomposition into simpler parts. This should be infrequent if the software base is mature, containing the results of prototyping many other systems in the same problem domain.

The execution support system helps to speed up design changes by providing a localized view of the processes in the prototype and by analyzing its timing properties. These features are especially important for prototyping real-time systems. At the programming language level, implementations of real-time systems are difficult to understand because the instructions of several logically independent processes must often be interleaved to meet the timing constraints [2]. PSDL presents a view to the designer in

which logically distinct processes are represented as separate independent components. The PSDL execution support system contains a translator which mechanically transforms this independent representation into the corresponding programming language representation, adding the necessary interleaving in a fashion transparent to the designer.

The timing properties of a real-time system are analyzed by the static scheduler. If the static scheduler succeeds in constructing a schedule, then the operators in the schedule are guaranteed to meet their timing constraints even under worst case operating conditions. In case the static scheduler fails to find a valid schedule, it provides diagnostic information useful for determining the cause of the difficulty and whether or not the difficulty can be resolved by adding more processors [9]. These functions are important because the timing constraints in complex systems can have complicated interactions that can be very difficult to analyze manually.

The prototyping language PSDL is the vehicle for carrying a powerful set of concepts useful for modeling complex systems and for providing a uniform framework for representing prototypes and software components which is common to all of the tools. PSDL also helps the designer achieve good modularity and allows the descriptions of small modifications to component behavior to be separated from the potentially complex implementations of those components.

Good modularity means the prototype should be realized by a set of independent modules with narrow and explicitly specified interfaces. PSDL supports this concept via operators and data streams. An important property of the language is that two distinct operators can communicate or affect each other's behavior only by means of the data streams explicitly connecting them, either directly or indirectly. This locality property is

important for maintenance because it allows the set of modules that can potentially interact with a given module to be determined via a simple mechanical analysis of the data flow network, allowing the software tools to guarantee that all aspects of a proposed change have been covered. It also encourages designs containing an independent component for each major design decision. Such designs are easier to modify because the information required to change a design decision is localized in one region of the code.

The locality property is embodied by the PSDL scoping rules and mechanically enforced. The implementation of an operator can only refer to the explicitly declared input and output streams of the operator and to data streams local to the implementation of the operator. Implementations of operators representing state machines may contain closed loops consisting of local data streams.

PSDL supports small modifications to the behavior of a component by means of control constraints. This mechanism can be used to adapt the behavior of a prototype to make a small change without examining the internal implementation of the affected component. For example, a common kind of problem discovered in the demonstration of a prototype is that a given operator has the intended behavior most of the time but not always. The PSDL control constraints governing conditional execution of operators are useful in such a case. A control constraint in the form of an input guard predicate can be added in such a case, where the guard predicate describes the circumstances in which the execution of the operator will produce the intended result, and serves to disable the execution of the operator in cases where it would not produce the correct result. This allows the addition of another operator for producing the correct output in the remaining cases, controlled by a complementary guard predicate. Another example is a case where an

operator is discovered to have an inappropriate response to ill-formed inputs. The PSDL control constraints governing conditional outputs or exceptions are useful in such a case. Such control constraints introduces an output guard predicate, which serves to disable the output of the operator if it does not satisfy the predicate, or an exception triggering predicate, which produces an exception instead of the value computed by the operator. Output guard predicates can be used to filter out inappropriate responses in cases where no response is needed. In particular, they can be used to disable exceptions raised by implementations of components if the conditions reported by the exception do not require any action on the part of the prototype. Exception triggering predicates can be used to trigger exceptions when incorrect outputs have been computed, or to rename exceptions produced by the implementation of a module to other conditions meaningful at a higher level. Triggering an exception is useful because it allows a new module to be added for handling the exception, again without affecting the implementation of the original component. Renaming exceptions is useful for repairing inappropriate error messages. For example, in the context of an operating system, a process_table_overflow condition might be translated into a "machine_busy" condition to convert an internal view of a failure in terms of the implementation to an external view meaningful to the users of the prototyped system. An exception triggering predicate is used instead of an input guard predicate in cases where the condition to be checked depends on the output values of the operator in addition to its inputs values.

4. Conclusions

The effort required for supporting the evolution of a software system can be reduced via prototyping. Prototyping can be used to stabilize the requirements for either an initial

software development project or a proposed enhancement to an existing system. Especially for systems with complex requirements, such as large or embedded real-time systems, human communication is ineffective without the feedback provided by demonstrations of proposed system behavior. An iterative process involving modifications to perceived requirements and proposed system behavior is needed to arrive at a common understanding of the proposed system by the customer and the developer. It is more cost effective to use a prototype rather than production quality code to provide the demonstrations of proposed system behavior because prototypes are simpler and easier to modify than production quality implementations.

The effectiveness of prototyping is limited if it must be carried out manually. A high level language, a systematic prototyping method, and an integrated set of computer-aided prototyping tools are important for realizing the potential benefits of prototyping. The prototyping effort is also aided by a powerful set of abstractions appropriate for a problem domain, especially if these abstractions are embodied in a set of reusable software components. Effort can be saved in the long run by building up a comprehensive library of such components for an application area, especially if more than one software system must be developed for the same problem domain, which is often the case.

A typical software system evolves in a long series of repairs and enhancements, in response to the discovery of faults and to changes in user requirements. Most useful systems are too large to be maintained by just one person, leading to the need for coordinating the concurrent efforts of a group of people. Enhancements to software systems must often be developed concurrently even though they are not independent. Sometimes the

designers talk to each other as they proceed, and adjust their designs to make sure they are compatible. In other cases, the designs are developed independently, and then are merged at the end, with both designers examining each other's code and making adjustments as needed. Manual methods for merging enhancements are inadequate because they are slow and error prone. Automatic methods for merging enhancements are needed. To be useful, such methods should provide some assurances that the results are correct, or else locate potential inconsistencies.

A better understanding of the software merging problem and better computer aided design tools enabled by that understanding are important because the bulk of the cost of a software system is due to enhancements and repairs. Coordinating the efforts of many people working on the same software system is difficult and expensive, and a design style that allows people to work more independently and uses automatic merging of independent enhancements should reduce communication and coordination problems.

The ability to easily swap in different alternative choices for an aspect of the behavior of a system would also be useful in using a prototype to aid in determining user requirements. Automated merging would make it practical to customize software products for each user's needs by picking options from a multiple choice menu, as is commonly done for automotive products now. Such an approach would make software systems more flexible and make it less critical to get the requirements right the first time. It would also make it easier to design and maintain a family of software products intended to exist in a variety of configurations.

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

2. S. Faulk and D. Parnas, ''On Synchronization in Hard-Real-Time Systems'', *Comm. of the ACM 31*, 3 (Mar. 1988), 274-187.

3. 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.

4. S. Letovsky and E. Soloway, ''Delocalized Plans and Program Comprehension'', *IEEE Software 3*, 3 (May 1986), 41-49.

5. L. Levy, ''A Metaprogramming Method and Its Economic Justification'', *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 272-277.

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

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

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

9. Luqi and V. Berzins, ''Execution of a High Level Real-Time Language'', in *Proc. of the Real-Time Systems Symposium*, Dec. 1988.

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

11. Luqi, ''Knowledge Base Support for Rapid Prototyping'', *IEEE Expert*, Nov. 1988.

12. R. Yeh and T. Welch, ''Software Evolution: Forging a Paradigm'', in *Proc. Fall Joint Computer Conference*, ACM and IEEE Computer Society, Oct. 1987, 10-12.

Initial Distribution List

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

Ada Joint Program Office
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 230301 — 1

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

Office of the Secretary of Defense
Attn: CDR Barber
The Star Program
Washington, D.C. 20301 — 1

Naval Ocean Systems Center
Attn: Linwood Sutton, Code 423
San Diego, CA 92152-5000 — 1

National Science Foundation
Division of Computer and Computation Research
Washington, D.C. 20550 — 1

Director of Research Administration
Code 012
Naval Postgraduate School
Monterey, CA 93943 — 1

Chairman, Code 52
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100 — 1

LuQi
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100 — 150