1990

Models for Evolutionary Software Development

Luqi

Naval Postgraduate School

NPS52-90-012

NAVAL POSTGRADUATE SCHOOL

Monterey, California

MODELS FOR EVOLUTIONARY
SOFTWARE DEVELOPMENT

Luqi

August 1989

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research funded by the Naval Ocean Systems Center.

[redacted]

LUQI
Associate Professor
of Computer Science

Reviewed by:

[redacted]

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]

GORDON E. SCHACHER
Dean of Faculty
and Graduate Studies

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): NPS52-90-012	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: NOSC
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5100	7b. ADDRESS (City, State, and ZIP Code): SAN DIEGO, CA 92152-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: NOSC; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N66 00189WRB0355
8c. ADDRESS (City, State, and ZIP Code): SAN DIEGO, CA 92152-5000	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): MODELS FOR EVOLUTIONARY SOFTWARE DEVELOPMENT (U)

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
FINAL	FROM Mar TO Aug 89	August 1989	27

16. SUPPLEMENTARY NOTATION

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

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

This report explores the benefits to be derived from elaborating and automating models for evolutionary software development. The structure of the report is summarized in Figure 1.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] 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

Models for Evolutionary Software Development

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This report explores the benefits to be derived from elaborating and automating models for evolutionary software development. The structure of the report is summarized in Fig. 1.

1. Introduction

In recent years, the software engineering research community has been focusing significant attention on the process of software development and enhancement as a complement to the more traditional emphasis on the products of those processes. This interest recently has led to several new approaches to modeling and analyzing software processes [1-6] which provide ways of coping with some of the difficulties encountered in previous models.

In view of the rapidly expanding number of competing approaches, it is important to re-assess the key issues of software development processes, re-examine some fundamental assumptions, identify common characteristics of life-cycle models, and determine

---

*** The Problem**

*** Objectives**

*** Key Issues**

*** Advantages**

*** Approach**

*** Example**

Fig. 1 Outline

---

some principles guiding the steps of software processes.

Examples of models for evolutionary software development are provided in the attached additional technical reports, as summarized below.

(1) NPS 52-88-39 "Software Evolution Via Prototyping", describes rapid prototyping models used in evolutionary software development.

(2) NPS 52-89-16 "Petri-Net Based Models of Software Engineering Process" presents an extension of the classical Petri net model to formally define the functional, structural, and dynamic aspects of software engineering processes.

(3) NPS 52-89-44 "Software Analysis and Testing Through Prototyping" focuses on the validation and verification aspect of the software development process, and indicates the role of prototyping methods in this context.

(4) NPS 52-89-56 "Multi-level Software Analysis and Testing in Evolutionary Software Development" discusses research directions on aspects of software analysis and testing in the software process, and identifies needed software analysis processes at all levels of the development process, from requirements analysis to software evolution.

1.1. Problems With Models

Some of the difficulties encountered in previous models are shown in Fig. 2. The variety of incomparable models makes it difficult to compare and evaluate different software development processes, and to combine different models which focus on different aspects of software development. Alternative paradigms for development make comparisons difficult because concepts important in one model may not have any counterparts in another model based on a completely different paradigm. Accurate comparisons are impossible when the basic concepts and notations defining a model have not been clearly defined, and experimental evaluation becomes impossible when actual development practices do not correspond to the models used to describe and analyze those

---

*** the variety of incomparable models,**

*** alternative underlying software development paradigms,**

*** lack of precise meaning of concepts and notation used, and**

*** mismatch between models and practical development practices.**

Fig. 2 Difficulties With Previous Models

---

processes.

Some of these issues are illustrated in Fig. 3, which shows a plausible but informal description of the software development process. Some questions raised by this example are shown in Fig. 4. An essential problem with previous models has been that they are too informal. Early models were described by simple block diagrams without any underlying mathematical structure or precise definitions. Such models may be useful for orientation purposes, but they do not provide enough structure to form the basis for computer-aided software development.

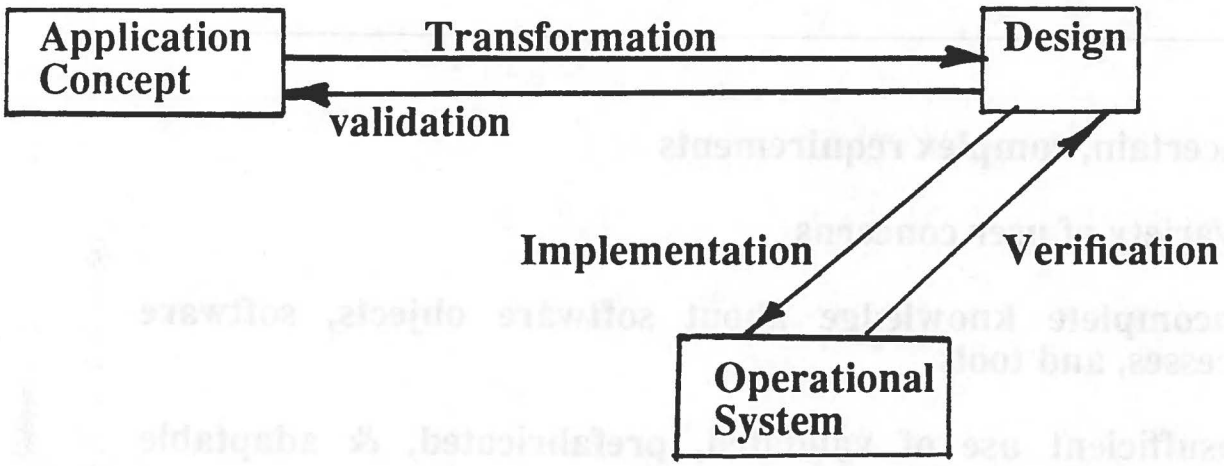

Fig. 3 Example of an Imperfect Process Model

*** What does the notation mean (box, arrow, text)?**

*** How are these elements composed?**

*** How can complete observation of such a model be guaranteed?**

Fig. 4 Questions Raised by the Imperfect Process Model

Practical software development processes must cope with the difficulties shown in Fig. 5.

Most software products are developed for a group of users with differing needs and concerns. These concerns may conflict with each other, and are imprecisely known because most of the potential users of the system do not interact directly with the developer. In addition to being uncertain, the goals for a large system can be very complex, to the point where a single person may not be able to understand them all in detail. Currently software objects, processes, and tools are not completely understood, partially because of unresolved technical and scientific questions, and partially because all three are constantly being changed and extended. In the face of so much uncertain information, feedback is essential for avoiding wasted effort and reducing risk. Such feedback is needed early in the process to provide the opportunity to solve potential problems before

---

*** uncertain, complex requirements**

*** a variety of user concerns**

*** incomplete knowledge about software objects, software processes, and tools**

*** insufficient use of validated, prefabricated, & adaptable software components**

*** risks of misdevelopment due to late or insufficient feedback information**

*** individuality of application domains, organizations, methods and tools implies need to adapt processes**

*** long lifetime of software requires enhancements due to changing requirements and environmental conditions**

*** need to integrate development and maintenance processes performed by different organizations**

Fig. 5 Difficulties of Software Development

---

the allotted time for the project has run out. Software development processes must be adaptable to different applications and methods because the conditions for different development projects can vary widely. This implies that software development is not a single process, but rather a family of related processes with a rich structure and many conceptual dependencies. Finally, the long lifetime of software products and the large fraction of the costs associated with software evolution is a major concern. Practical software development must accommodate many modifications or enhancements to the product as the process proceeds, and it must address the issue of handing the product over to a maintenance organization once the initial development is complete.

2. The Problem

The need to model and analyze software engineering processes is more important today than ever, because the advent of new methods and technologies aimed at various aspects of these processes is forcing managers and developers to decide how to best utilize them. The variety of life-cycle and process models presents a problem. Different approaches are hard to compare and judge for the reasons listed in Fig. 6.

Most of the life-cycle models do not have sufficient empirical data to prove their effectiveness and show their impact on software quality. In the mass of competing approaches much of the common sense and many of the basic principles of software engineering processes hidden in technical details of specific process models need to be

---

*** emphasize different aspects of software development processes and thereby are likely to sacrifice others,**

*** use different concepts and notations, and**

*** support different software development paradigms such as**

- automatic programming and formal program transformation,

- evolutionary development via rapid prototyping and fourth generation languages, and

- 'knowledge-based software assistant' approaches.

Fig. 6 Difficulties in Comparing Different Approaches

---

re-assessed. Theoretical foundations and analytical comparisons are needed in this area because it is very expensive to develop a life cycle model, create the necessary tool support and carry out experiments to determine the relative effectiveness of competing approaches in practice. The conceptual foundations of the field must be clarified to the point where meaningful questions can be asked, and appropriate experiments can be designed to provide useful answers to those questions with a relatively small number of case studies.

3. Objectives

The objectives for future work on software process models should include the ones listed in Fig. 7.

A meta-model is needed to provide a formal framework and precise notations for formalizing and comparing alternative approaches. Such formalization is needed to support meaningful comparisons and automated tools for effectively realizing the process models in practice. Prototyping is a promising new approach for supporting evolutionary software development, which provides a useful and challenging test case for exercising the meta-model. A meat-model should be capable of describing development methods and supporting tools with sufficient detail to enable automated monitoring or control functions which are capable of preventing or detecting errors and recording derivations or justifications for design decisions based on the process model. The model should provide the basis for automating the aspects of development concerning coordination of component activities and interactions between the different people or processes involved. The process model should also aid in the analysis and description of the product, and provide guidance for the design and construction of automated engineering database support for analyzing and describing the resulting software products.

4. Key Issues

Some of the key issues in the proposed approach are identifying and separating the goals of the meta-model and the goals of the software development process models it will define. The meta-model should provide the capabilities shown in Fig. 8.

A comprehensive and standardized vocabulary is needed to allow meaningful comparisons of different process models. In addition to standardized terminology, standardized structures for describing the elements of software development processes are needed. Such elements include engineering data, states of a development project, actions transforming states and data, constraints on the order of the actions, and the mechanisms for carrying out the actions or verifying that they have been correctly carried out.

Software process models will be defined using the facilities provided by the meta-model. Software process models derived should have the properties listed in Fig. 9.

A process model can be subjected to useful scientific analysis and automated procedures for supporting the model can be objectively designed only if the process model has a precise formal representation. A flexible representation scheme is needed, because the process often has to be changed as it is carried out in response to new information and new circumstances. Management considerations indicate that the model should provide measurable properties of the process which can support estimation, planning, and quality control activities in an objective and computer-aided way. The process models should be

*** Defining a meta-model of software processes which**

- supports alternative development paradigms,

- reflects common understanding and fundamental characteristics of software processes,

- accommodates enhancements to an ongoing process,

- facilitates effective management of development processes,

- enables automation

*** Designing a concrete process model supporting evolutionary software development through prototyping.**

*** Describing suitable methods and support tools.**

*** Preventing/detecting errors.**

*** Recording derivation/justification**

*** Providing foundations for automating part of the process, reducing coordination problems and increasing speed.**

*** Analyzing & documenting the product model with the assistance of design/project/engineering databases.**

Fig. 7 Objectives for Research

*** A comprehensive vocabulary.**

*** A small set of structuring principles to describe the elements and structure of**

- data domains,

- development states,

- actions transforming software objects from one state into another and caus

- mechanisms that help affect such transformations on different levels of abstraction and from different perspectives.

Fig. 8 Properties of a Meta-Model

A software process model should

*** be formal and explicit to enable automated consistency and completeness analysis, reasoning, and replay**

*** be executable to support symbolic testing and a priori demonstration of designed development processes**

*** be changeable as they are executed, to respond to new information**

*** be measurable for estimation, planning, quality control**

*** be parameterized to reflect dependencies on properties of the application**

*** organize tools to guide coordination of development tasks and define units of configuration control**

*** provide a coherent framework for communication, management, and tool development**

*** provide reusable standard process components with alternatives**

Fig. 9 Properties of a Software Process Model

parameterized to capture the dependencies of the process model on the properties of the application domain, so that the necessary adaptation can be carried out in a disciplined and predictable way. The process model provides the context for organizing tools and coordinating development tasks in terms of meaningful transactions which can serve as the basis for configuration control. By making the process predictable and providing a systematic structure for representing alternative choices, formalized process models provide a basis for communication, project management, tool development, and the construction of reusable libraries of standardized process components. This should provide better control over the development process and should enable effective tailoring of the process to current needs of particular projects.

5. Advantages

The advantages of the proposed approach are summarized in Fig. 10.

It is very difficult to make accurate cost and schedule estimates if the tasks to be carried out are unknown. The process model provides a structure for predicting and identifying the tasks involved in particular projects, thus aiding estimation. A clear picture of

---

*** Schedules and costs of development steps can be better calculated based on precise knowledge of actions involved.**

*** Resources can be better allocated based on knowledge about causal dependencies among actions.**

*** Development history of individual components can be traced.**

*** Alternatives courses of development are made explicit.**

*** Process model execution can be automated.**

*** Alternative development steps can be evaluated prior to process execution.**

*** Different process models can be compared and put into contrast.**

*** Fundamental characteristics of software processes are reflected by the basic building blocks and structuring mechanisms of the meta-model.**

*** Families of process models are constructed via parameterization and (de-)composition mechanisms.**

*** Process histories are formally deduced from process models.**

Fig. 10 Advantages of the Proposed Approach

---

causal dependencies between actions can make planning more systematic and can enable automated procedures for aiding decision makers in evaluating their options. The development history of a product can be presented in meaningful terms if it can be linked to a coherent process model. The development history of a software product contains a huge amount of information, which is useless unless it can be structured and simplified to make it comprehensible and to enable people to find the particular pieces of information about the past which will help them make meaningful decisions about what to do in the next step. The formal structure can enable automated support for identifying and evaluating the alternative choices a designer has at each point in the development. The structure also allows evaluation of different process models, and can provide decision support tools for determining which process model is most appropriate for a particular project. The structures provided by generic process components with parameters, explicit composition mechanisms, and standard sets of building blocks enable concise descriptions of different alternatives, which make it easier to represent alternatives and to identify choices. Similar structures induced on process histories can be used for re-evaluating decisions when the product must be modified, by locating and organizing the relevant design choices in terms of the structure of the development processes involved. These structures can also be useful in diagnosing and locating errors.

6. Approach

Our approach to meeting these objectives consists of the tasks shown in Fig. 11.

*** survey of life-cycle models and software development methods used for evolutionary software development,**

*** design of a meta-model by exploiting experience from software specification and design techniques,**

*** specification and restructuring of a selected prototyping approach to software development in terms of the meta-model,**

*** adaptation and integration of prototyping tools to support the designed process, and**

*** evaluate analysis and reasoning capabilities.**

Fig. 11 Summary of Tasks

A survey of existing process models provides an initial version of the requirements for a meta-model, by providing a set of test cases. The proposed meta-model should be capable of representing all of the concerns addressed by current informal process models. These concerns must be formalized and their essential features abstracted to provide a clean and independent set of building blocks for the meta-model. The proposed meta-model should be exercised, evaluated, and extended by applying it to the formalization and improvement of a selected prototyping approach to evolutionary software development. This will involve restructuring and integrating the prototyping tools to correspond to the developed process model. The application will provide a basis for evaluating the analysis and reasoning capabilities provided by the tools.

7. Example

A simplified example of a generalized meta-model is shown in Fig. 12. According to this model, the software development process consumes resources and produces a software product. Different versions of the model differ in the types of resources that are considered and the detailed composition of the resulting products, as well as the steps that are carried out to create the products.

7.1. Versions

The generalized model has multiple specializations, all of which fit the same general framework. Each of the specializations represents an alternative approach to software development. Reasons for using multiple alternatives are listed in Fig. 13.

7.2. Prototyping

The prototyping cycle is one possible variation of the meta-model. Prototyping is an attractive approach for situations described in Fig. 14. We focus on this variation in our case study because it covers an important class of applications.

7.3. Evaluation

Some of the important characteristics of prototyping are shown in Fig. 15.

Prototyping seeks to reduce costs and errors by providing inexpensive feedback earlier in the development process. This is accomplished by separating concerns and reusing software components at different levels of the process. Separation of concerns enhances the capabilities for incremental analysis, development, and validation.

7.4. Functions of Prototyping

The roles of prototyping in software development are shown in Fig. 16.

It is difficult to communicate with users about a proposed new system because software is abstract and difficult to visualize. Since most users are not specialists in computer science, they cannot be expected to understand formal notations for describing system behavior. Demonstrations of the behavior of a prototype provide a representation of the proposed system's behavior that users can readily understand and evaluate. Since a prototype is constructed quickly and inexpensively, it can provide user feedback early in the development process, when adjustments and modifications have a much lower cost than near the end of the cycle. A prototype can provide a demonstration of the feasibility of implementing key system concepts, and can provide the basis for evaluating the merits

Product = { Version }

Version = requirements + spec + design
+ code + manuals + ...

Resource = budget + time + people
+ hardware + tools
+ software components + ...

Fig. 12 Representative/Meta/Generalized Software Development Model

Different

* types of applications,

* tool sets, and

* development organizations.

Fig. 13 Reasons for Adapting Process Models

* unfamiliar applications,

* systems that change user organizations, and

* complex/hard real-time/embedded systems.

Fig. 14 Applicability of Prototyping

* reduced costs and errors,

* quick small feedback loops,

* separation of concerns,

* different levels of reuse, and

* incremental development.

Fig. 15 Characteristics of Prototyping

* communication with users,

* quick & inexpensive feedback,

* demonstration of feasibility,

* evaluation of alternative designs,

* aid in synthesis: decomposition, and

* platform for evolution: flexibility.

Fig. 16 Roles of Prototyping

of alternative designs for critical subsystems. A prototype can help in the construction of the production-quality system by helping to arrive at a modular decomposition of the problem. The prototype also provides a basis for evolution, because prototypes are typically described in simple, high level notations and provide a simplified view of the system before optimization transformations have introduces additional details and logical dependencies that reduce flexibility.

7.5. Creating a CAPS

Prototyping is most effective if supported by a computer-aided prototyping system (CAPS). Such a system is a mechanism for speeding up the process. Some of the important characteristics of a CAPS are shown in Fig. 17.

A formal prototyping language is needed to serve as a basis for the automated tools comprising the CAPS. Such a language should provide simplicity and expressive power to make it easy to analyze and process prototype descriptions and allow a designer to rapidly construct a prototype with minimal mental effort. Such a language is used as a medium to formulate proposed system behaviors in a form that can be demonstrated and measured by automated tools. The language is used to specify and document the intended behavior of the system, both for guiding later development steps and as a basis

---

*** a formal prototyping language,**

*** basis for automated tools,**

*** provides simplicity & expressive power,**

*** formulate, demonstrate, measure,**

*** specify and document,**

*** link to reusable components,**

*** computer-aided transformation to implementation,**

*** execution Support: simulate, translate, schedule, monitor,**

*** user interface: graphics, syntax-directed edit, browser,**

*** database: configuration + reusable components, and**

*** optimization: refinement + transformation.**

Fig. 17 Capabilities of a Prototyping System

---

for automatically retrieving reusable software components that can help to realize the prototype. The specification part of the prototyping language should also serve as the basis for computer-aided transformations into production-quality implementations.

The tools for execution support should provide capabilities to simulate components that are not yet available and to translate descriptions of system decompositions into reasonably efficient implementations. While performance of a prototype is not an overriding issue, prototypes must run with sufficient speed to provide demonstrations of system behaviors within practical time periods. Scheduling is needed to evaluate the feasibility of meeting real-time constraints in a proposed system, relative to given estimates of design characteristics for the system. Facilities for monitoring the execution of the prototype are critical for evaluation and debugging purposes. User interface considerations are important for speeding up the process of constructing and modifying a prototype. Graphical displays, syntax directed editing facilities which provide support for computer-assisted design completion, and browsers for quickly locating relevant pieces of information are some of the kinds of tools that can provide support for user interface issues. Database support is also critical for effective rapid prototyping, because of the large volumes of information involved. Some of the critical functions of the database portion of a CAPS are configuration control and management of reusable components. Tools for refinement and optimization are essential for helping to cope with performance issues. Such issues arise when transforming a prototype design into a production-quality implementation, and when the performance of the prototype must be improved to enable effective demonstrations.

7.6. Levels of Analysis and Testing

A prototype serves as a vehicle for analyzing and testing proposed systems at the earliest stages of requirements analysis and functional specification. Most approaches to testing require the generation of system outputs or responses by some means. Three approaches to providing this capability are summarized in Fig. 18, along with indications of the advantages and disadvantages of each approach. Simulation involves some form of direct execution of a component specification. The advantage of this approach is low designer effort, because the specification is a simplified view of the component, which leaves out many details. This approach has the disadvantage of inefficiency because the details that are left out are needed for efficient execution. General methods for evaluating specifications, such as equation solving or logic programming, usually involve inherently slow processes such as exhaustive, unbounded searching.

An interpreter for a module interconnection language provides better efficiency than direct execution of specifications, and provides generally good flexibility and control because the execution support system has access to detailed information about the execution of the prototype and its relationships to the formal prototype description. Some disadvantages of this approach are that it requires some additional designer effort and that the timing of the prototype execution does not reflect the timing characteristics of the production version of the system very accurately. This second disadvantage is shared with the simulation approach.

The approach of translating the prototype description directly into Ada has the advantages of providing relatively accurate timing estimates, better efficiency, and access to many reusable software components. Disadvantages of this approach are difficulties in

Simulation: Executable spec

+ low designer effort

- inefficient

Interpreter: Module interconnection language

+ Flexible: dynamic binding & modification

+ Controllable: powerful debug, reverse execution

- Timing does not reflect production version

Ada: Augment and transform specifications

+ Efficient

+ Accurate timing

+ Reusable components

- Hard to construct

- Inflexible

Fig. 18 Evaluation of Approaches to Prototype Execution

modifying the properties of the prototype as it executes, and difficulties in constructing the prototype, because of the extra details the designer must specify.

Since none of these approaches is clearly superior to the others, a CAPS should support all three possibilities, and allow them to be used together in the same prototype.

7.7. Verification and Validation

Verification and validation are essential parts of the prototyping process, as illustrated in Fig. 19. The aspect of a prototype design that is most important to verify is the correctness of a proposed system decomposition.

Testing means simulating the design using a finite set of test cases to see if the proposed structure operates as inteded. Testing is an effective means of detecting errors, but it is usually not capable of certifying correctness of a design. In cases where reliability is critical, mechanically checked proofs of correctness may be needed. Such proofs involve showing that a given interconnection of specified components realize the specified behavior of the subsystem for all possible inputs. Such proofs are simpler than traditional proofs of correctness because they operate entirely at the specification and design level, without any consideration of coding details.

The other major function of prototyping is to validate proposed system behavior. Demonstrations allow the user to inspect actual system behaviors and determine whether they meet the real needs of the user, rather than some approximation to those needs captured by a written specification. To effectively carry out such validations, it is necessary to identify a set of typical operational transactions or scenarios which illustrate actual problems that are supposed to be solved by the proposed system. Such transactions act as test cases to be used in the user demonstration. These test cases must be mapped into the interactions supported by the prototype. In cases where this mapping cannot be carried out, faults in the system are detected without the need for users to be involved in a demonstration.

Verification

* testing, or

* proof at component spec level.

Validation

* user demonstrations, and

* typical operational transactions.

Fig. 19 Verification and Validation via Prototyping

7.8. Configuration Control

The need for speed in developing prototypes implies that more than one designer will usually be involved in the process. The purpose of the configuration management facilities of a CAPS is to help coordinate group activities as detailed in Fig. 20. It is essential that modifications made by one designer do not get lost or invalidate work done by other designers. Configuration control mechanisms meet these goals by enforcing serializability of updates via some type of locking protocol. Configuration control systems are subject to additional goals of minimizing the amount of lost time due to waiting for locks.

Another function of configuration control is to avoid wasting everyone's time by making a damaged version of a subsystem visible to the entire group. This goal can be partially met by automatically running mechanical error checking procedures before making a version public, and requiring repairs before allowing a version into the baseline in case the mechanical checks fail.

Configuration control for prototypes provides software objects with frozen versions. The motivation for this capability is shown in Fig. 21.

*** Non-interference: locking, serializability, and**

*** Correctness: enforce error checking.**

Fig. 20 Configuration Control in Prototyping

*** stability,**

*** no read locks,**

*** creating new alternatives: no write locks, and**

*** computer-aided merging of alternatives.**

Fig. 21 Advantages of Frozen Versions

Since versions cannot change, a prototype constructed from a given set of versions of its subsystems provides a stable and reproducible snapshot of a design alternative. Such a snapshot can be recreated and evaluated as necessary, without concern for interference by ongoing experimentation with alternative versions of the prototype. Since versions cannot change, there is no need for locks preventing the reading of a version, thus producing maximum access to all existing versions. If the configuration management system supports creating new alternative lines of development, then there is no need for write locks either, and all designers are free to create new versions without the need to worry about interference. This is made possible by creating a new alternative line of development whenever a lock would have blocked access in an ordinary database system. Such a facility must be combined with automated support for recombining or merging the features of alternative lines of development. This process must be reliable, in the sense that all potential conflicts need to be detected by the merging process.

7.9. Derivations: Evolution and Variations

The database for supporting rapid prototyping should provide facilities for capturing and utilizing derivation information for the decisions embodied in a prototype design. Desirable properties of such derivations are shown in Fig. 22.

The logical structure of a design history indicates what decisions were made and the logical dependencies between them. This information is different than the historical order in which decisions were made, because independent decisions should not be artificially ordered by historical accident. Also, logical dependencies between decisions should not be hidden just because they were made out of order by mistake. The logical structure of a design history should indicate the refinements in each line of development, where each refinement corresponds to the information added by a compatible design decision. Alternatives represent incompatible ways of resolving the same aspect of a design, and represent choice points for the designer. Separating logical dependencies from historical orderings and explicitly representing alternative choices enables computer support for reordering decisions and factoring out common parts of different lines of development. This allows the system to simplify and clarify the choices faces by the

---

*** logical structure, not actual history,**

*** refinements and alternatives, and**

*** computer-aided reordering and factoring.**

Fig. 22 Properties of Derivation Histories

---

designer, and makes it easier to navigate through the design space when seeking to modify an evolving system.

7.10. Versions of Composite Objects

Prototypes of practical software systems are too large to be built as monolithic structures, and hence are usually realized by some type of hierarchical decomposition. This introduces the problem of managing the versions of composite objects, which is described in Fig. 23.

A change to the specification of a composite subsystem introduces a natural unit of atomic transaction: the change should be made either completely or not at all. This induces some dependencies between the versions of the components of the modified subsystem. A configuration control system should support such atomic transactions by keeping track of which versions of the components at the next lower level correspond to each version of the composite subsystem. This facility is important for exploring design alternatives quickly, because it allows the designer to switch between alternative versions of a high level subsystem without concern for the dependencies between the subcomponents: these are managed automatically by the configuration control system.

8. Conclusion

More than 25 theses at NPS show the feasibility of different aspects of computer aided prototyping. Improved solutions for many problems are desirable. Many of these solutions involve interactions between different aspects of the software development process. We have found the need for better formulations of the structure of the development process to provide more effective tools support and to guide the further development of computer-aided prototyping systems.

1. B. Boehm, ''A Spiral Model of Software Development and Enhancement'', *Computer 21*, 5 (May 1988), 61-72.
2. B. Curtis, H. Krasner and N. Iscoe, ''A field study of the software design process for large systems'', *Comm. of the ACM 31*, 11 (Nov. 1988), 1268-1287.
3. A. Finkelstein, " ''Not waving but drowning": Representation Schemes for modelling software development'', in *Proceedings of the 11th Annual*

*** atomic transactions**

*** coordinating versions of components**

Fig. 23 Managing Versions of Composite Objects

International Conference on Software Engineering, Pittsburgh, PA, May 1989, 402-404.

4. W. Humphrey and M. Kellner, ''Software process modelling: Principles of entity process models'', in *Proceedings of the 11th Annual International Conference on Software Engineering*, Pittsburgh, PA, May 1989, 331-342.

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

6. Luqi, ''Software Evolution via Rapid Prototyping'', *IEEE Computer*, May 1989.

DISTRIBUTION LIST

(1)	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
(2)	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
(3)	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
(4)	Director of Research Administration Attn: Prof. Howard Code 012 Naval Postgraduate School Monterey, CA 93943	1
(5)	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
(6)	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1
(7)	National Science Foundation Division of Computer and Computation Research Attn. Tom Keenan Washington, D.C. 20550	1
(8)	Naval Postgraduate School Code 52Lq Computer Science Department Monterey, CA 93943	50
(9)	Attn: Linwood Sutton Naval Ocean Systems Center Code 411 San Diego, CA 92152-5000	1