NPS CS-92-014

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A MODEL FOR MERGING SOFTWARE
PROTOTYPES

David A. Dampier
Luqi

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

REAR ADMIRAL R. W. WEST JR.   HARRISON SHULL
Superintendent   Provost

REPORT DOCUMENTATION PAGE

ORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS

URITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
LASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited

ORMING ORGANIZATION REPORT NUMBER(S) CS-92-014	5. MONITORING ORGANIZATION REPORT NUMBER(S)

ME OF PERFORMING ORGANIZATION uter Science Dept. Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS(52)	7a. NAME OF MONITORING ORGANIZATION

RESS (City, State, and ZIP Code) erey, CA 93943	7b. ADDRESS (City, State, and ZIP Code)

ME OF FUNDING/SPONSORING GANIZATION l Postgraduate School	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER OM&N Direct Funding

RESS (City, State, and ZIP Code) terey, CA 93943	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

E (Include Security Classification)
DDEL FOR MERGING SOFTWARE PROTOTYPES

RSONAL AUTHOR(S)
ID A. DAMPIER, LUQI

PE OF REPORT nical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 92/09/23	15. PAGE COUNT 16

PPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
D	GROUP	SUB-GROUP	SOFTWARE, AUTOMATION, COMPUTER AIDED PROTOTYPING, MAINTENANCE, FORMAL MODELS, SOFTWARE ENGINEERING, SOFTWARE MERGING, CHANGE INTEGRATION, CASE TOOLS

TRACT (Continue on reverse if necessary and identify by block number)
    As software becomes more complex, more sophisticated development and maintenance methods are needed sure software quality. Computer Aided Prototyping achieves this via quickly built and iteratively updated ypes of the intended system. This process requires automated support for keeping track of many independent es and for exploring different combinations of alternative changes and refinements. This paper formalizes the /change merging process and extends the idea to multiple changes to the same base prototype. Applications of chnology include: automatic updating of different versions of existing software with changes made to the ne version of the system; integrating changes made by different design teams during development; and checking tency after integration of seemingly disjoint changes to the same software system.

RIBUTION/AVAILABILITY OF ABSTRACT LASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
ME OF RESPONSIBLE INDIVIDUAL OI	22b. TELEPHONE (Include Area Code) (408) 646-2912	22c. OFFICE SYMBOL CSLq

A MODEL FOR MERGING SOFTWARE PROTOTYPES

David A. Dampier
Luqi
Computer Science Department
Naval Postgraduate School
Monterey, California 93943
e-mail: dampier@cs.nps.navy.mil
or luqi@cs.nps.navy.mil

ABSTRACT

As software becomes more complex, more sophisticated development and maintenance methods are needed to ensure software quality. Computer Aided Prototyping achieves this via quickly built and iteratively updated prototypes of the intended system. This process requires automated support for keeping track of many independent changes and for exploring different combinations of alternative changes and refinements. This paper formalizes the update/change merging process and extends the idea to multiple changes to the same base prototype. Applications of this technology include: automatic updating of different versions of existing software with changes made to the baseline version of the system; integrating changes made by different design teams during development; and checking consistency after integration of seemingly disjoint changes to the same software system.

KEYWORDS

SOFTWARE, AUTOMATION, COMPUTER AIDED PROTOTYPING, MAINTENANCE, FORMAL MODELS, SOFTWARE ENGINEERING, SOFTWARE MERGING, CHANGE INTEGRATION, CASE TOOLS

I. INTRODUCTION

Software development is an ever-increasing and complex industry. As software systems gain sophistication and maintaining them becomes more difficult, automated software development methods and the supporting formal models must be devised to increase reliability and decrease post-development maintenance effort.

Computer Aided Prototyping is one such method to reduce maintenance costs by making the original requirements conform more closely to the real needs of the users. Systems correctly

implementing an accurate set of requirements have lower maintenance costs because there are fewer surprises when the system is put into actual use. An appreciable part of the maintenance activity can be carried out by changing/updating the prototype rather than repeatedly updating the production version of the intended system. This is useful because the prototype description can be significantly simpler than the production code if the prototype is expressed in a notation tailored to support modifications, and the software tools in the computer aided prototyping environment can help carry out the required modifications rapidly [Lu 89]. Prototyping a software system using tools decreases development time and increases maintainability, because it reduces customer dissatisfaction with the delivered system [Lu 92].

The designers construct/change prototypes of the intended systems quickly to meet the customer's desires during the requirements analysis phase. The designers need automated tools which will allow several changes to a base version of a software prototype to be automatically combined as well as automatically propagated through multiple alternative versions of the prototype. Formal models are the keys and foundations for building such automated tools.

Change merging is the process of automatically combining the effects of several changes to a software system. Change merging has been studied in the context of software maintenance and conventional methods for software development. Early version control systems such as SCCS [Si92] and RCS [Ti 82] provide primitive change merging facilities based on string editing operations on the source text, without considering the effects on program behavior. However automated tools must provide guarantees regarding program behavior to be trusted by designers. Semantically-based change merging seeks to construct a program whose behavior agrees with the changed version in all situations where the behavior of a changed version differs from the behavior of the base version. The behavior of the constructed program should agree with the base version

for all situations where the behaviors of all the changed versions agree with the behavior of the base. The problem for functional programs was considered in [Be 86]. Semantically-based change merging based on program slicing [We 84] and data flow analysis has been studied for imperative while-programs [Re 88, Ho 90]. A general theory of change merging that can apply to any kind of programming language is described in [Be 91a], along with a high resolution approach to change merging for while-programs based on specifications and meaning functions [Li 79]. An initial exploration of change merging models for the prototyping language PSDL can be found in [Da 90].

Change merging is an important aspect of computer-aided prototyping because the prototyping process is characterized by rapid and extensive changes. The Computer Aided Prototyping System (CAPS) [Lu 89] is a computer aided prototyping environment comprised of a software database system, an execution support system, and a user interface that helps designers to develop prototypes. The software database system manages changes to multiple versions of prototype designs and provides an expert system to select and retrieve reusable components from the software base. The design database 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 and updated. Such a parallel update will split off a new branch or *variation* in the version history [Lu 90]. The system will provide a warning that a new version is currently in preparation and information about the reason the component is being modified (i.e. some particular new or modified requirement) on request. The methods proposed in this paper provide automated support for combining both branches of a split resulting from parallel updates to produce a version that incorporates the effects of both of the updates.

Our goal is to develop a tool for the CAPS system which will support automatic merging of different versions of a prototype. We have developed a model which shows that it is possible to correctly perform a merge operation in most cases [Da 90]. This paper formalizes the change process for the Prototyping System Design Language (PSDL), a design based language written specifically for CAPS, and uses this formalization to strengthen our merging model.

II. PROTOTYPING IN CAPS

Computer aided prototyping allows the user to get a better handle on his/her requirements early in the conceptual design phase of development and use automated tools to rapidly create "a concrete executable model of selected aspects of a proposed system" [Lu 89], to allow the user to view the model, and to make comments early. The prototype is then rapidly reworked and re-demonstrated to the user over several iterations until the designer and the user have a precise view of what the system should do. This process produces a validated set of requirements which become the basis for implementing the final product [Lu 89]. The prototype can also become part of the final product. In some prototyping methodologies, the prototype is an executable shell of the final system, containing only a subset of the system's ultimate functionality. After the prototype is approved by the customer, the holes are filled in and the system is delivered. In this approach to computer aided prototyping, software systems can be delivered incrementally as parts of the system become fully operational [Lu 89].

CAPS, a computer-aided software development environment, supports prototyping of embedded hard real-time systems [Lu 89]. CAPS reduces the effort of the prototype designer by providing an integrated set of tools that help design, translate and execute the prototypes, along with a language in which to design and program the prototypes.

The Prototype System Description Language (PSDL) is the prototyping language associated with CAPS [Lu 88]. It was created to provide the designer with a simple way to abstractly specify software systems. A PSDL program is a set of PSDL operators and data types, containing zero or more of each. PSDL operators and types consist of a specification and an implementation. The specification defines the external interfaces of each operator through a series of interface declarations, provides timing constraints, and describes the functionality of the operator through the use of formal and informal descriptions. The implementation can either be in PSDL or Ada. PSDL implementations are data flow diagrams augmented with a set of data stream definitions and a set of control and timing constraints.

III. CHANGING PROTOTYPES

A current focus of CAPS is formalization of the change process. In order to discuss the merging of changes made to a prototype, we must first provide a mathematical model of the change process.

PSDL prototypes can be considered iterative versions of a software system. If S is the intended final version of the software system, then each successive iteration of the prototype can be viewed as an element of a sequence S_i where $\lim_{i \to \infty} S_i = S$. Each prototype S_i is modelled as a graph $G_i = (V_i, E_i, C_i)$, where:

A. V_i is a set of vertices. Each vertex can be an atomic operator or a composite operator modelled as another graph.

B. E_i is a set of data streams. Each edge is labelled with the associated variable name. There can be more than one edge between two vertices. There can also be edges from an operator to itself, representing state variable data streams.

C. C_i is a set of timing and control constraints imposed on the operators in version i of the prototype.

The prototype designer repeatedly demonstrates versions of the prototype to users, and designs the next version based on user comments. The change from the graph representing the ith version of the prototype to the graph representing the (i+1)st version can be described in terms of graph operations by the following equations:

$$S_{i+1} = (V_{i+1}, E_{i+1}, C_{i+1}) = S_i + \Delta S_i$$

$$\Delta S_i = (VA_i, VR_i, EA_i, ER_i, CA_i, CR_i) \text{ where:}$$

$V_{i+1} - V_i = VA_i$: The set of vertices to be added to S_i.

$V_i - V_{i+1} = VR_i$: The set of vertices to be removed from S_i.

$E_{i+1} - E_i = EA_i$: The set of edges to be added to S_i.

$E_i - E_{i+1} = ER_i$: The set of edges to be removed from S_i.

$C_{i+1} - C_i = CA_i$: The set of timing and control constraints to be added to S_i.

$C_i - C_{i+1} = CR_i$: The set of timing and control constraints to be removed from S_i.

The + operation above is defined as follows:

$$V_{i+1} = V_i \cup VA_i - VR_i$$

$$E_{i+1} = E_i \cup EA_i - ER_i$$

$$C_{i+1} = C_i \cup CA_i - CR_i$$

The following figures show an example of a change made to a composite operator in PSDL. Figure 1 contains a graph representation for a composite operator Op1 consisting of 4 vertices and 6 data streams. Figure 2 shows a change to be applied to Op1 to produce Op2. Figure 3 shows a graph representation of Op2, the result of applying the change to Op1.

Figure 1. Example of a composite operator in PSDL
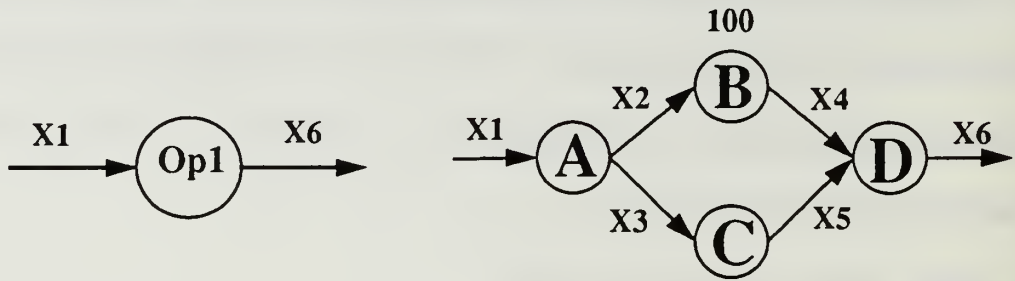
$Op1 = (V_1, E_1, T_1, C_1)$

$V_1 = \{A, B, C, D\}$

$E_1 = \{(X1: EXT\text{->}A), (X2: A\text{->}B), (X3: A\text{->}C), (X4: B\text{->}D), (X5: C\text{->}D),$

$\qquad (X6: D\text{->}EXT)\}$

$C_1 = \{max\_exec\_time(B,100ms)\}$

Figure 1. Example of a composite operator in PSDL

$\Delta_A Op1 = \{VR_A, VA_A, EA_A, ER_A, TA_A, TR_A, CA_A, CR_A\}$
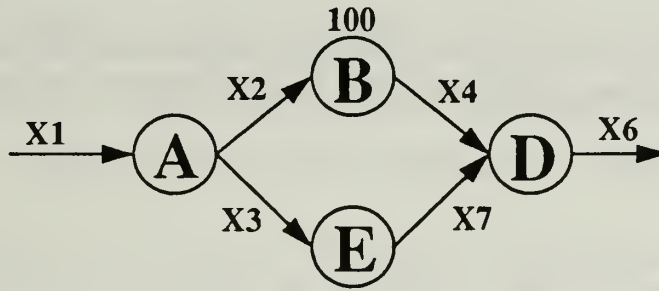
$VA_A = \{E\}$

$VR_A = \{C\}$

$EA_A = \{(X3: A\text{->}E), (X7: E\text{->}D)\}$

$ER_A = \{(X3: A\text{->}C), (X5: C\text{->}D)\}$

$CA_A = \{latency(X7, E, D, 50ms)\}$

$CR_A = \{\}$

Figure 2. Example of a change made to Op1.

$$\text{Operator } Op2 = Op1 + \Delta_A Op1$$

$Op2 = \{V_2, E_2, T_2, C_2\}$

$V_2 = V_1 \cup VA_A - VR_A = \{A, B, C, D\} \cup \{E\} - \{C\} = \{A, B, D, E\}$

$E_2 = E_1 \cup EA_A - ER_A =$

$\{(X1: EXT\text{->}A), (X2: A\text{->}B), (X3: A\text{->}C), (X4: B\text{->}D), (X5: C\text{->}D), (X6: D\text{->}EXT)\} \cup$

$\{(X3: A\text{->}E), (X7: E\text{->}D)\} - \{(X3: A\text{->}C), (X5: C\text{->}D)\} =$

$\{(X1: EXT\text{->}A), (X2: A\text{->}B), (X3: A\text{->}E), (X4: B\text{->}D), (X7: E\text{->}D), (X6: D\text{->}EXT)\}$

$C_2 = \{max\_exec\_time(B,100ms), latency(X7, E, D, 50ms)\}$

Figure 3. Example of the changed operator Op2.

IV. MERGING PSDL PROTOTYPES

Merging different versions of a program is useful in performing automatic maintenance of software systems. In prototyping, it is common for different versions to evolve from the base system. If the system designer discovers a fault in the base version of the system, it would be desirable to have the capability to automatically apply that change to all of the versions currently in use. In order to do this, the merging process must be able to apply the change to the common parts of each version without affecting the peculiar functionality in each one.

In [Be 90], a definition of merging two compatible extensions of a semantic function was given as follows:

If the functionality of the software systems are represented using sets, then the result of merging two extensions, A & C of a base version B is defined as:

$$M = A[B]C = (A - B) \cup (A \cap C) \cup (C - B)$$

In this definition, the union, intersection and difference operations are defined as normal operations on sets. The difference operation, (A - B) for example, yields the functionality present in the extension, but not inherited from the base version. The intersection operation yields the functionality preserved from the base version in both extensions. This model preserves all changes made to the base version, whether extensions or retractions.

In this section, we express our method for merging prototypes using the change model described in the previous section and the above definition. All PSDL implementations are graphs, which model their functionality. We have represented these graphs using sets. Different variations of a prototype are the result of different changes being applied to a common base version. We can merge the two new versions A and C together by applying the change which produced A from B to version C, or applying the change which produced C from B to version A. The result is the same in either case.

Earlier, we defined the $(i + 1)$st iteration of a software prototype as $S_{i+1} = S_i + \Delta S_i$. Let us now look at an ith version which has been changed in two different ways, via Δ_A and Δ_B. The result of these two changes is S_A and S_B respectively. Now let us define the $(i + 1)$st iteration as

$$S_{i+1} = S_A[S_i]S_B = (S_A - S_i) \cup (S_A \cap S_B) \cup (S_B - S_i)$$

The components of S_{i+1}; V_{i+1}, E_{i+1} and C_{i+1} can be defined similarly:

$$V_{i+1} = V_A[V_i]V_B = (V_A - V_i) \cup (V_A \cap V_B) \cup (V_B - V_i),$$

$$E_{i+1} = E_A[E_i]E_B = (E_A - E_i) \cup (E_A \cap E_B) \cup (E_B - E_i) \text{ and}$$

$$C_{i+1} = C_A[C_i]C_B = (C_A - C_i) \cup (C_A \cap C_B) \cup (C_B - C_i)$$

To demonstrate the concept of the merging operation, we provide the following example: The base prototype is as in Figure 1. Change A is outlined in Figure 2, with the result shown in Figure 3. Change B is outlined in Figures 4 and 5. The merging operation is performed in Figure 6 and the result is shown in Figure 7.

The affect of change A is to remove the operator C and replace it with operator E. Accordingly, the associated data streams must also be changed. The new data stream X7 also has a latency associated with it, so a new timing constraint is added. The sets, V_A, E_A and C_A correspond directly to V_2, E_2 and C_2 shown in Fig. 3.

$$\Delta_B Op1 = \{VR_B, VA_B, EA_B, ER_B, CA_B, CR_B\}$$
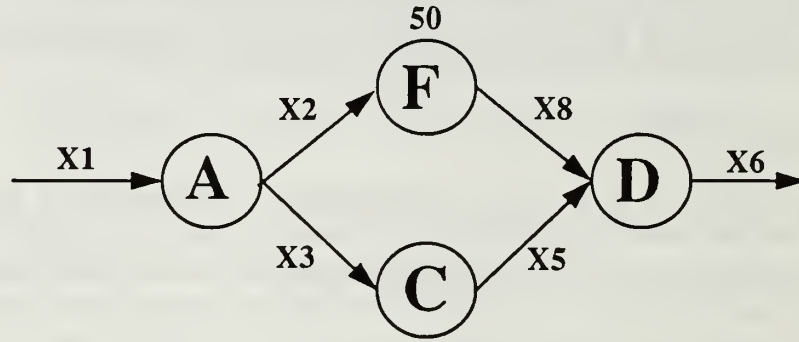
$$VA_B = \{F\}$$
$$VR_B = \{B\}$$
$$EA_B = \{(X2\ A\text{->}F),\ (X8\ F\text{->}D)\}$$
$$ER_B = \{(X2\ A\text{->}B),\ (X4\ B\text{->}D)\}$$
$$CA_B = \{max\_exec\_time(F,50ms)\}$$
$$CR_B = \{\}$$

Figure 4. Change B applied to Op1.

Figure 5. Results of applying change B to Op1.

The affect of change B is to remove the operator B and replace it with operator F. The data streams associated with these two operators also have to be changed now. A new timing constraint is also added associated with operator F.

The merge operation outlined in Figure 6 involves determining the real affect of changes made to the base, and any conflict that may arise due to similar changes between the two variations.

This is a simple example illustrating the merging of two changed prototypes which do not conflict with one another. In some cases, two changes to a prototype can conflict with one another, and the result of their merging can be an inconsistent program. In such cases, the engineer must resolve the conflict off-line. The following section describes some possible conflicts and possible methods for resolving those conflicts.

$$Op2 = OpA[Op1]OpB = (OpA - Op1) \cup (OpA \cap OpB) \cup (OpB - Op1) =$$

$$V_2 = V_A[V_1]V_B = (V_A - V_1) \cup (V_A \cap V_B) \cup (V_B - V_1),$$

$$E_2 = E_A[E_1]E_B = (E_A - E_1) \cup (E_A \cap E_B) \cup (E_B - E_1) \text{ and}$$

$$C_2 = C_A[C_1]C_B = (C_A - C_1) \cup (C_A \cap C_B) \cup (C_B - C_1)$$

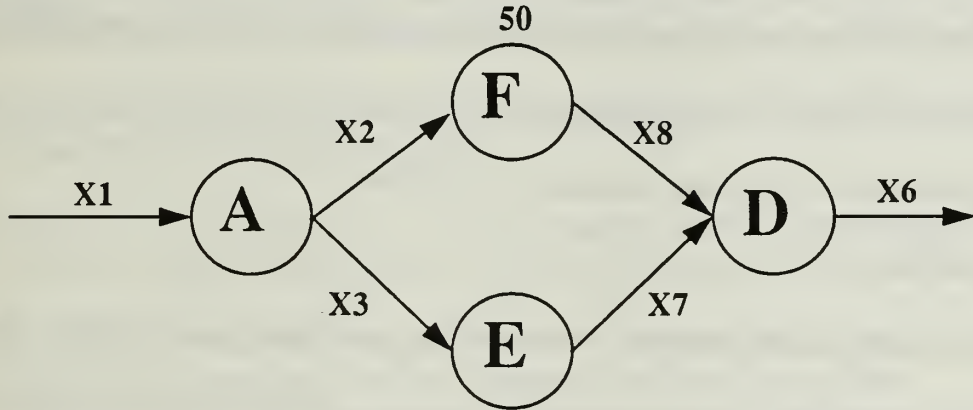**Figure 6. Performing the merge operation.**

Figure 7. Result of the merge operation.

V. Conflict Resolution

There are a number of possible conflicts which can arise during the performance of the merging operation. Conflicts arise when different changes applied to the prototype affect the same portion of the prototype in different ways. Some examples of conflicts are as follows:

1. If one change adds an output edge to a vertex A, while another change removes vertex A from the prototype. In this case, automatic resolution of the conflict is not yet possible, so the system would have to notify the designer that a conflict has occurred and give him/her the opportunity to resolve it.

2. If the two changes assigned different timing constraint values to the same operator, i.e., (max_exec_time, F, 50ms) and (max_exec_time, F, 40ms). In this case, the conflict can be handled

automatically, since any operator which executes in under 40ms would certainly execute in under 50ms. In situations where different maximum execution times have been assigned, the minimum value can always be chosen. This is also true of two different values for latency, maximum response time and finish within timing constraints. The minimum calling period timing constraint would have to be merged using the maximum of the different values. Different period values for the same operator in different changes would result in a conflict which would have to be resolved by the designer. Different control constraints for the same part of the prototype in different changes can also result in a conflict. Some of these conflicts can be resolved automatically. Current work is addressing methods for automatic resolution of conflicts.

VI.   Conclusions

Tool support for manipulating and combining specifications is especially important for computer aided prototyping. We are currently implementing the method presented here to evaluate its effectiveness in practical contexts. We are also conducting theoretical studies to evaluate its limitations and to discover improvements. The method described here works correctly whenever the functions computed by the operators are one to one. As has been pointed out in [Be 90], a global analysis of the system may be necessary to ensure that the functions computed by the operators do not interfere in the general case. For a more detailed discussion of the reasons for this, see [Be 90]. Related work on configuration management and version control is also being performed [Ba 92]. Some issues to be considered in future work are treatment of data types and component specifications, and the detection/diagnosis of semantic interference between modifications.

LIST OF REFERENCES

[Ba 92]   Badr, S. and Berzins, V., "A Design Management and Job Assignment System", Technical Report, CS, NPS, 1992.

[Be 86]   Berzins, V., "On Merging Software Extensions", Acta Informatica, Springer-Verlag, 1986.

[Be 90]   Berzins, V. "Software Merge: Semantics of Combining Changes to Programs", Submitted for publication in ACM Transactions on Programming Languages and Systems, 1990.

[Be 91a]   Berzins, V. "Software Merge: Models and Methods for Combining Changes to Programs", Journal of Systems Integration, vol. 1, no. 2, August 1991, pp. 121-141.

[Be 91b]   Berzins, V. and Luqi, SOFTWARE ENGINEERING WITH ABSTRACTIONS, Addison-Wesley, Reading, MA, 1991.

[Be 92]   Berzins, V., Luqi, Yehudai, A., "Using Transformations in Specification-Based Prototyping", IEEE Transactions on Software Engineering, August, 1992.

[Da 90]   Dampier,D., A Model for Merging Different Versions of a PSDL Program, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.

[Ho 88]   Horwitz, S., Prins, J., and Reps, T., "Integrating Non- Interfering Versions of Programs", Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, New York, 13 - 15 January 1988.

[Ho 90]   Horwitz, S., Reps, T. and Binkley, D., Interprocedural Slicing Using Dependence Graphs", ACM Transactions on Programming Languages and Systems, January 1990.

[Lu 88].   Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", IEEE Transactions on Software Engineering, pp.1409-1423, October 1988.

[Lu 89]   Luqi, "Software Evolution Through Rapid Prototyping", IEEE Computer, May 1989.

[Lu 90]   Luqi, "A Graph Model for Software Evolution", IEEE Transaction on Software Engineering. Vol. 16. NO. 8. Aug. 1990

[Lu 92]   Luqi, "Computer-Aided Prototyping for a Command-And-Control System Using CAPS", IEEE Software, Jan. 1992.

[Li 79]   Linger, R., Mill, H., Witt, B., STRUCTURED PROGRAMMING: THEORY AND PRACTICE, Addison-Wesley, Reading, MA, 1979.

[Re 88]   Reps, T. and Yang, W., "The Semantics of Program Slicing", Computer Science Technical Report #777, University Of Wisconsin-Madison, 1988.

[Re 89]   Reps, T., On the Algebraic Properties of Program Integration, Computer Sciences Technical Report #856, University of Wisconsin at Madison, June 1989.

[Si 92]   Silverberg, I., SOURCE FILE MANAGEMENT WITH SCCS, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Ta 89]   Tanik, M. and Yeh, R., "Rapid Prototyping in Software Development", Computer, vol. 22, pp. 9-10, May 1989.

[Ti 82]   Tichy, W.,"Design, Implementation, and Evaluation of a Revision Control System," in Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, Sept. 1982.

[We 84]   Weiser, M., "Program Slicing", IEEE Transactions on Software Engineering SE-10, 4(July 1984), 352-357.

DISTRIBUTION LIST

Defense Technical Information Center   2
Cameron Station
Alexandria, VA 22314

Director of Research Administration, Code 08   1
Naval Postgraduate School
Monterey, CA 93943

Library, Code 52   2
Naval Postgraduate School
Monterey, CA 93943

CPT David A. Dampier, USA   10
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Dr. Luqi   10
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Dr. Valdis Berzins   1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Dr. Mantak Shing   1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, CA 93943

Dr. Craig Rasmussen   1
Mathematics Department, Code MA
Naval Postgraduate School
Monterey, CA 93943

Dr. Dan Dolk   1
Administrative Science Department, Code AS
Naval Postgraduate School
Monterey, CA 93943