Theses and Dissertations | Thesis Collection

1994-06

A formal method for semantics-based change-merging of software prototypes

Dampier, David A.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/30841

NAVAL POSTGRADUATE SCHOOL.
Monterey, California

DISSERTATION

A FORMAL METHOD FOR SEMANTICS-BASED CHANGE-MERGING OF SOFTWARE PROTOTYPES

by

David Anthony Dampier

June 1994

Dissertation Supervisor: Valdis Berzins

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE June 1994	3. REPORT TYPE AND DATES COVERED Doctoral Dissertation	
4. TITLE AND SUBTITLE A Formal Method for Semantics-Based Change-Merging of Software Prototypes(U)			5. FUNDING NUMBERS
6. AUTHOR(S) Dampier, David Anthony			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This dissertation addresses the need for a formal method to support the merging of changes in independently developed versions of a prototype in a computer-aided rapid prototyping system. The goal is to provide the prototype developer with the ability to: combine independently developed enhancements to a prototype, check for consistency, and automatically update all derived versions of a prototype with changes made to the base version.

A useful semantics-based method is provided for change-merging that is guaranteed to detect all conflicts. Prototype slicing is used to capture the affected parts of each variation and the preserved part of the base in both variations. We then combine the affected parts with the preserved part using our model, which includes the first use of Brouwerian Algebras to formalize the merging of hard real time constraints. Our Slicing Theorem guarantees that this method produces a prototype that correctly exhibits the significant behavior of each of the input versions, provided the changes do not conflict. The method achieves correctness by comparing the slice of the change-merged version with respect to each affected part against the same slice of the appropriate changed version. If the slices are the same, the change-merge is correct, otherwise a diagnostic message results. A preliminary conditional method for programs with loops is also provided that is strictly more accurate than previous methods.

This dissertation contributes to computer-aided software maintenance by providing a model, algorithm and implementation for an automated change-merging tool for PSDL prototypes. Preliminary testing shows that this tool will enhance the ability of the prototype developer to deliver a prototype in less time by enabling more concurrency in the development effort.

14. SUBJECT TERMS FORMAL METHODS, PROGRAM MERGING, CHANGE-MERGING, PROTOTYPING, SLICING			15. NUMBER OF PAGES 225
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

NSN 7540-01-280-5500

i

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

A FORMAL METHOD FOR SEMANTICS-BASED CHANGE-MERGING OF SOFTWARE PROTOTYPES

by

David Anthony Dampier
Captain, United States Army
B.S., University of Texas at El Paso, 1984
M.S., Naval Postgraduate School, 1990

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1994

Author: _____
David Anthony Dampier
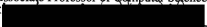
Approved by: _____
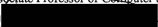
Luqi   Mantak Shing
Associate Professor of Computer Science   Associate Professor of Computer Science

_____   _____
Daniel R. Dolk   Craig W. Rasmussen
Professor of Information Systems   Assistant Professor of Mathematics

_____
Valdis Berzins
Professor of Computer Science
Dissertation Supervisor

Approved by: _____
Ted Lewis, Chairman, Department of Computer Science

Approved by: _____
Richard S. Elster, Dean of Instruction

ii

ABSTRACT

This dissertation addresses the need for a formal method to support the merging of changes in independently developed versions of a prototype in a computer-aided rapid prototyping system. The goal is to provide the prototype developer with the ability to: combine independently developed enhancements to a prototype, check for consistency, and automatically update all derived versions of a prototype with changes made to the base version.

A useful semantics-based method is provided for change-merging that is guaranteed to detect all conflicts. Prototype slicing is used to capture the affected parts of each variation and the preserved part of the base in both variations. We then combine the affected parts with the preserved part using our model, which includes the first use of Brouwerian Algebras to formalize the merging of hard real time constraints. Our Slicing Theorem guarantees that this method produces a prototype that correctly exhibits the significant behavior of each of the input versions, provided the changes do not conflict. The method achieves correctness by comparing the slice of the change-merged version with respect to each affected part against the same slice of the appropriate changed version. If the slices are the same, the change-merge is correct, otherwise a diagnostic message results. A preliminary conditional method for change-merging while programs is also provided that is strictly more accurate than previous methods.

This dissertation contributes to computer-aided software maintenance by providing a model, algorithm and implementation for an automated change-merging tool for PSDL prototypes. Preliminary testing shows that this tool will enhance the ability of the prototype developer to deliver a prototype in less time by enabling more concurrency in the development effort.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I. INTRODUCTION

During iterative development of software prototypes, different variations are generally developed where each of the versions contains a portion of the desired capability. Because these prototypes can be very large, tools that automatically determine the differences between these versions and produce a new version exhibiting significant behavior from each are desirable. This dissertation defines a change-merging method that is semantics-based and guarantees that if a conflict-free result is produced, it is semantically correct, and provides a working change-merging tool to be integrated into the Computer-Aided Prototyping System (CAPS). Traditional syntax-based merging tools fall short of providing results guaranteed to be semantically correct, and earlier semantics-based change-merging or integration methods concentrated on combining changes to simple imperative or while programs. We explore a domain of *enhanced data flow programs*, written in PSDL, which are inherently non-deterministic and parallel. Our change-merging method provides the first real change-merging capability for this domain of programs.

Software change-merging is also applicable to software maintenance activities. Assuming that a software system has been developed using the computer-aided prototyping paradigm, or can be translated into the prototyping language, different versions of that software can be automatically updated with changes made to the base version by applying our method. The fielded version would be one variation and the updated base version would be the other variation. If all of the changes made to the base version are compatible with the fielded version, applying our method results in a new fielded version updated with the changes made to the base version. If the changes are not compatible, this information is provided automatically by our method. Using this technology eliminates the need for software designers to manually check if changes are compatible before performing updates. It also

allows fewer designers to make changes to existing software systems, as well as prototypes in development. In an industry with projected costs in the billions of dollars [Ref. 40], this translates into significant savings to both the software developer and the customer.

Other uses of this technology are found in the areas of software reuse and reengineering. In software reuse, complex reusable components can be retrieved from the software repository that contain more functionality than is required for the application. The desired functionality can be isolated using prototype slicing by taking the slice of the complex component with respect to the output streams desired. The resultant slice will contain any part of the complex component that affects the output stream. In reengineering, if a program written in some high-level language can be translated into the prototyping language, PSDL, then changes made to the prototype version of the base program can be automatically incorporated into the prototype versions of the target programs, and the resultant prototype can then be used to generate new production code for the reengineered program.

A. RAPID PROTOTYPING

Rapid prototyping is an approach to software development that was introduced to overcome the following weaknesses of traditional approaches:

1. Fully developed software systems that do not satisfy the customer's needs, or are obsolete upon release.
2. No capability for accurately evaluating real-time requirements before the software system has been built.

To overcome these weaknesses, computer-aided software development methods must be developed which ensure accurate requirements engineering and emphasize efficient change incorporation both during development and after fielding of the software system. Computer-Aided Rapid Prototyping is one such methodology. Rapid prototyping overcomes these weaknesses by increasing customer interaction during the requirements engineering phase

of development, providing executable specifications that can be evaluated for conformance to real-time requirements, and producing a production software system in a fraction of the time required using traditional methods. Rapid prototyping allows the user to get a better understanding of requirements early in the conceptual design phase of development. It involves the use of software tools to rapidly create concrete executable models of selected aspects of a proposed system to allow the user to view the model and make comments early. The prototype is rapidly reworked and redemonstrated 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 designing the final product [Ref. 36]. The prototype can also be transformed into part of the final product. In some prototyping methodologies, prototypes are developed, demonstrated and then thrown away before the production system is developed. In prototyping methodologies like the one used in CAPS, the prototype is an executable shell of the final system, containing a subset of the system's ultimate functionality. After the design of the prototype is approved by the customer, the missing functionality is added and the system is delivered. In this approach to rapid prototyping, software systems can be delivered incrementally as parts of the system become fully operational. Figure 1.1 shows the life-cycle model for this prototyping methodology.

In this model, the customer provides a set of initial goals to the designer. The designer takes those initial goals and formulates a set of requirements from which the first version of the prototype is designed. This prototype is then demonstrated to the user, with the user providing feedback to the designer. The designer takes the feedback, adjusts the requirements to reflect the adjusted goals and makes whatever changes to the prototype necessary to satisfy the requirements. It is then redemonstrated to the user for more feedback. This iterative process continues until a validated set of requirements is accepted by the user. The designer then takes the prototype and implements the remainder of the functionality needed to produce the operational system. The result is an operational software system that satisfies

3
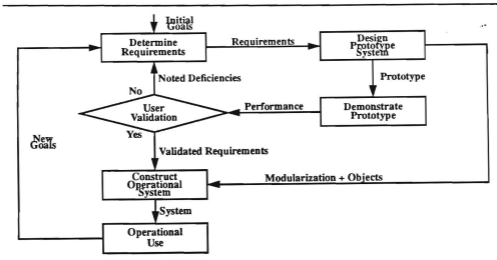
Figure 1.1: Rapid Prototyping Life-Cycle Model. [Ref. 20]

the customer's requirements and that is delivered in only a fraction of the time it would take using traditional software development methods

Change-merging is an integral part of the rapid prototyping methodology. During the *Design Prototype System* phase of prototype development, multiple variations of a large prototype are likely to be developed. This can happen when different development teams are working on different aspects of a system, or when different possible solutions to a problem are explored in different ways. In the first example, it will certainly be necessary for the separately developed pieces of the prototype to be combined into a single system before execution for the customer. In the second example, the customer may desire a system containing some or all of the aspects contained in each solution. In this case, these different prototypes must be change-merged to capture the significant parts of each variation. Our change-merging method will allow these combinations to be done automatically, ensuring that the resultant prototype is semantically correct, with respect to all of the input variations. If the pieces are not compatible with regard to the semantics of the prototype, then our method will identify the parts of the prototype containing the conflicts. This technology encourages the designer

4

to explore different solutions to a problem, and to spread the development workload in a large project without concern for the subsequent integration of these independent efforts.

B. PROTOTYPING SYSTEM DESCRIPTION LANGUAGE

Our method has been implemented for use in the CAPS development system. It is designed to operate on programs written in the Prototyping System Description Language (PSDL), associated with CAPS. PSDL is a high level specification and design language which can be translated into executable code.

PSDL is a generalization and extension of a data flow language, with the addition of control constraints and timing operations [Ref. 35]. A PSDL prototype consists of two parts: a specification and an implementation. The specification of a prototype contains the interface, and the implementation contains either a PSDL graph implementation, or a programming language implementation. The PSDL graph implementation contains a set of operators, a set of data streams through which the operators communicate with one another, and a set of control and timing constraints which specify restrictions on the execution of the operators or data streams. The programming language implementation is written in any high-level programming language like Ada or C that is supported by the environment.

All operators in PSDL prototypes are state machines. Since PSDL is, by definition, non-deterministic, the meaning of an operator in PSDL is a mathematical relation. PSDL operators with only one state, or an empty set of state variables, and only one possible outcome are functions. This meaning is defined by the operator's possibility function discussed in a later section.

A data stream in a PSDL prototype is a communications link between operators. Each data stream is either a data flow stream or a sampled stream. *Data flow streams* are FIFO buffers of lengths at least one. When a new value is written to the stream, it is appended to the buffer. Values are removed from a data flow stream only when they are read by the

5

consumer. Values on data flow streams can be read only once. *Sampled streams* are not traditional data flow streams. They have buffers of size one. When a value is written to the stream, it remains on the stream until a new value is written to the stream, at which time the old value is overwritten. A value is not removed from the sampled stream when read. Data streams can be written by more than one operator, and they can be read by more than one operator. A complete listing of the PSDL grammar is contained in Appendix D.

C. OVERVIEW

In the chapters that follow, we provide background information which we used to produce a working change-merging tool. Chapter II provides definitions of mathematical constructs used in later chapters. Chapter III provides information about related work, some of which was accomplished by others before our effort was started, and some we have accomplished during the course of the research effort. Chapter IV provides a semantic model for the PSDL computational model which we used to develop our algorithm, and it contains the discussions about this dissertation's primary contributions to the state of the art. Chapter V contains the algorithms used to implement our tool, along with a discussion of their correctness and complexity. Chapter VI outlines the development of the change-merging tool and Chapter VII provides our analysis of what we accomplished in this effort, and some future research options in this area. There are five appendices: Appendix A contains formal specifications for the constructs used in our model, Appendix B contains details about the effect of PSDL control constraints on our model, Appendix C contains proofs considered too lengthy to be included in the text of the dissertation, and Appendix D contains a listing of the PSDL grammar, and Appendix E contains the program listings of our implementation.

II. ALGEBRAIC FOUNDATION FOR MERGING

A. WHAT IS CHANGE-MERGING

Change-merging is a process that allows different changes to a software product to be combined using computer-aided tools. The result of this change-merge must contain the differences between the base version and each input version, and must be correct with respect to the method used; syntactic or semantic. Syntactic change-merging is performed on the source code of the the input versions with respect to the differences in the syntax of each version. Semantic change-merging is performed on the functions computed by the software product with respect to the behavior associated with each input version. Semantic change-merging requires a solid mathematical foundation to provide some guarantee of correctness and engender confidence in a working change-merging system. As has been pointed out in much of the previous work on merging, there is a solid foundation for representing program variations in algebra [Ref. 6, 28, 42]. This chapter introduces and explains the mathematical concepts needed to understand the work presented in later chapters. Section B describes the sets and partially ordered sets, and their relation to change-merging. Section C extends the discussion to Lattices and describes how lattices are used in change-merging. Section D builds up to Boolean and Brouwerian Algebras which are very useful in performing change-merge operations.

B. SETS AND POSETS

A set is a collection of objects, called elements. Operations on sets include $\in$ (membership test), $\cup$ (union), $\cap$ (intersection) and $-$ (difference). A partially ordered set, or *poset*, is defined as follows [Ref. 16]:

Definition 1 <u>Partially Ordered Sets</u>

A nonempty set X is said to be a *partially ordered set*, or *poset*, provided that a relation $\sqsubseteq$ is defined on X, satisfying the following:

1. $\sqsubseteq$ is *reflexive*: $x \sqsubseteq x$ for all $x \in X$;

2. $\sqsubseteq$ is *antisymmetric*: $x \sqsubseteq y$ and $y \sqsubseteq x$ imply that $x = y$;

3. $\sqsubseteq$ is *transitive*: $x \sqsubseteq y$ and $y \sqsubseteq z$ imply that $x \sqsubseteq z$.

Such a relation $\sqsubseteq$ is called a *partial ordering* of the set X.

Our method of change-merging is performed on variations of a PSDL program. Changes to PSDL programs are not always extensions of a previously defined program. Different variations can change a previous program in different ways. Since these different variations are not always compatible extensions of earlier versions, the set of all program variations does not form a completely ordered set. But since some program variations are compatible extensions of other programs, the set of all program variations forms a partially ordered set, with respect to an approximation relation, $\sqsubseteq$.

Definition 2 <u>Approximation Relation for PSDL Prototypes</u>

If x and y are two PSDL prototypes, x approximates y, written $x \sqsubseteq y$, if y exhibits any behavior that x exhibits.

Proposition 1 *The set of all possible PSDL prototypes is a poset.*

Proof:

If x and y are PSDL prototypes, let $\sqsubseteq$ be the approximation relation defined in Definition 2.

By Definition 1, for the set of all possible PSDL prototypes to be a poset, it must satisfy the three conditions, *reflexivity*, *antisymmetry*, and *transitivity*.

8

(a) Clearly $x \sqsubseteq x$, as x certainly exhibits its own behavior.

(b) Let $x \sqsubseteq y$ and $y \sqsubseteq x$. Then y exhibits any behavior that x exhibits, and x exhibits any behavior that y exhibits. Thus $x = y$.

(c) Let $x \sqsubseteq y$ and $y \sqsubseteq z$. Then y exhibits any behavior that x exhibits and possibly more, and z exhibits any behavior that y exhibits, so z exhibits any behavior that x exhibits. Thus $x \sqsubseteq z$.

Therefore by (a), (b) and (c), the set of all possible PSDL prototypes is a poset. □

C. LATTICES

A *lattice ordered* poset is a partially ordered set $(L, \sqsubseteq)$ such that for every pair of elements, $x, y \in L$, the supremum, $sup(x, y)$, and the infimum, $inf(x, y)$, exist [Ref. 34]. An example of a lattice is shown in Figure 2.1.

An *algebraic lattice* is a nonempty set L together with two binary operations, *meet* ($\sqcap$) and *join* ($\sqcup$), which satisfy the following conditions for all $x, y, z \in L$ [Ref. 34]:

(1) **Commutativity:** $x \sqcap y = y \sqcap x$ and $x \sqcup y = y \sqcup x$.

(2) **Associativity:** $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ and $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$.

(3) **Absorption:** $x \sqcap (x \sqcup y) = x$ and $x \sqcup (x \sqcap y) = x$.

(4) **Idempotence:** $x \sqcap x = x$ and $x \sqcup x = x$.

In the context of merging pure program extensions, the *meet* ($\sqcap$) operation represents the greatest common approximation of two programs, and the *join* ($\sqcup$) operations represents the least common extension. The greatest common approximation of two programs represents the functionality common to both programs, and the least common extension represents the union of both of their functionalities.

9
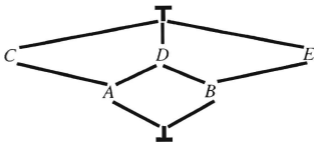
Figure 2.1: An Example of a Lattice.

According to [Ref. 34], every lattice ordered set is an algebraic lattice if we define $x \sqcap y = inf(x, y)$ and $x \sqcup y = sup(x, y)$.

A *distributive* lattice is an algebraic lattice for which at least one of the following properties holds:

1. $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.
2. $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$.

An algebraic lattice $\mathcal{L}$ is *complemented* if for every $x \in \mathcal{L}$ there is at least one element $y \in \mathcal{L}$ such that $x \sqcup y = \top$ and $x \sqcap y = \bot$. We say that y is the i *complement* of x.

D. BOOLEAN AND BROUWERIAN ALGEBRAS

A *Boolean Algebra* is a complemented, distributive lattice [Ref. 34]. Change-merging over Boolean algebras is done very simply using set operations. A very rich and well understood set of laws is available for the use of Boolean Algebras.

A *Brouwerian Algebra* is a distributive lattice with a pseudo-difference operation, $\dot{-}$, characterized by the property $x \dot{-} y \subseteq z \iff x \subseteq y \cup z$. This property states that the pseudo-difference of two sets x and y is contained in the set z if and only if x is contained in the supremum of y and z. A formal definition of Brouwerian algebras follows [Ref. 39]:

Definition 3 Brouwerian Algebras

A Brouwerian algebra is an algebra $(L, \cup, \cap, \dot{-}, \top)$ *that satisfies the following properties:*

(i) $(L, \cup, \cap)$ *is a lattice with a greatest element,* $\top$.

(ii) L *is closed under* $\dot{-}$.

(iii) For all elements $x, y, z \in L$, *the formulas* $x \dot{-} y \subseteq z$ *and* $x \subseteq y \cup z$ *are equivalent.*

[Ref. 39] also provides the following properties of Brouwerian algebras:

Theorem 1 *Let* L *be a Brouwerian algebra under* $\cup, \cap$ *and* $\dot{-}$. *Then:*

(i) L *has a zero element,* $\bot$ *determined by the formula* $\bot = \top \dot{-} \top$.

(ii) L *is a distributive lattice.*

(iii) If $x \subseteq y$, *then* $x \dot{-} z \subseteq y \dot{-} z$, $z \dot{-} y \subseteq z \dot{-} x$, *and* $\top \dot{-} y \subseteq \top \dot{-} x$.

(iv) $x \subseteq y \iff x \dot{-} y = \bot$.

(v) $x \subseteq y \cup (x \dot{-} y)$.

(vi) $(x \cup y)x \dot{-} y \subseteq x$.

(vii) $x \dot{-} z \subseteq (x \cup y) \dot{-} z$.

(viii) $z \cup (x \dot{-} y) = z + [(z \cup x) \dot{-} (z \cup y)]$.

(ix) $z \dot{-} (x \cap y) = (z \dot{-} x) \cup (z \dot{-} y)$.

(x) $(x \cup y) \dot{-} z = (x \dot{-} z) \cup (y \dot{-} z)$.

(xi) $\top \dot{-} (\top \dot{-} x) \subseteq x$.

(xii) $\top \dot{-} (\top \dot{-} (\top \dot{-} x)) = \top \dot{-} x$.

(xiii) $\top \dot{-} \bot = \top$ *and* $\top \dot{-} \top = \bot$.

(xiv) $x \cup (\top \dot{-} x) = \top$.

The proof of this theorem is contained in [Ref. 39].

Brouwerian algebras are very useful in the study of sets in which the true difference between two elements is not guaranteed to exist.

E. SUMMARY

It turns out that every component of PSDL programs that can be change-merged can be modeled using lattices or algebras. Many of the different parts of PSDL prototypes which are merged separately do not fit nicely into Boolean algebras, with the exception of some control constraints, so we introduced the concept of Brouwerian algebras. Throughout this dissertation, the concepts discussed in this chapter are used to prove different parts of the change-merging model contained in Chapters III and IV, and considered in the development of the algorithm and implementation.

III. RELATED WORK

This chapter reviews and assesses some of the work related to the change-merging problem which has already been accomplished. Since change-merging is a relatively new problem, there have been a number of research efforts aimed at defining the theoretical foundations for the problem, but not much effort has been placed on implementing a solution for real programs. Our research effort is the first to tackle a real-world problem and succeed in providing a working solution. This effort would have been nearly impossible, however, had it not been for the pioneering work reviewed in this chapter.

A. TEXT BASED MERGING

The earliest work on program merging relied on combining changes made to the text files containing the source code for the program [Ref. 43, 45]. These early systems certainly provided an advance to the then-current state of the art, but syntax-based merging did not prove useful in the general case, as syntax-based merging proved insufficient to provide any guarantee of semantic correctness [Ref. 6].

The first of the text-based merging systems was introduced as part of a software management toolkit called the *Revision Control System* or (RCS) [Ref. 45]. This system was developed as a way to maintain the update history of a file. The system saves the initial version of the file when invoked for the first time and, in subsequent invocations, saves only the changes made to the previous version. Merging is accomplished through the use of the command *RCSMERGE*. RCSMERGE tries to combine the differences between two different changes to the same base document based on the assumption that changes to disjoint portions of the text are independent. Where it is able to combine the changes, it makes

13

the change to the output file. When it is not able to combine the differences, it prints the respective piece of each version as a conflict in the output file, so the author can resolve it manually.

These systems work well for most text files with small individual changes. For programs, however, they do not provide even a guarantee of syntactic correctness, and in some cases when the changes are significant, the tool is unable to match even the parts that did not change.

B. MERGING OF PROGRAM EXTENSIONS

In [Ref. 6], Berzins presents the first definitive work on semantic-based program merging. This work is limited to considering program extensions, and does not consider changes that remove functionality from the base program. It recognizes that program extensions can be ordered using an *approximation* relation $\sqsubseteq$. If p is a base program, and q is an extension of p, then $p \sqsubseteq q$. That is to say that the functionality of q agrees with the functionality of p everywhere p is defined, but q may be defined where p is not.

With this ordering in mind, two programs p and q can be merged by finding the *least common extension* of p and q, written $p \sqcup q$, where p and q are base programs and $p \sqcup q$ is the merged program. He also recognizes that the exact least common extension of two programs is not computable in the general case, but a safe approximation is sufficient in practice.

Berzins considers four software domains: specifications, functions, programs and data types. These domains are defined in Figure 3.1. All of these domains are represented using lattices. The following sections describe the representation of these domains.

Specification:	Defines Acceptable Range of Behavior
Function:	Models Actual Behavior
Program:	Algorithms Defining Partial Functions
Data Type:	Set on which Programs Operate

Figure 3.1: Definitions of Relevant Domains

1. Functions, Specifications and Programs

Functions, specification and program domains can all be viewed as lattices with respect to the approximation ordering $\sqsubseteq$. Each lattice contains the elements of the domain together with a top element, $\top$, representing an overconstrained element, and a bottom element, $\bot$, representing an undefined element. The least common extension of two elements, x and y can then be defined in terms of lattice operations as the least upper bound of x and y, denoted $x \sqcup y$. If x and y are compatible, then $x \sqcup y \neq \top$, otherwise $x \sqcup y = \top$.

2. Data Types

The lattice for a domain representing a conventional *data type*, D_0, can be defined as a set $\mathcal{D} = D_0 \cup \{\bot, \top\}$, where $\bot$ approximates everything and $\top$ is an extension of everything. The definition of the extension relation for $\mathcal{D}$ is:

$$x \sqsubseteq y \iff (\bot \equiv x) \vee (x \equiv y) \vee (y \equiv \top)$$

The least upper bound of any two unequal elements in this domain is $\top$, the overconstrained element. This model applies to data types whose elements are either completely defined or completely undefined. An example of a type that is not covered by this construction is a list with a component selector implemented using lazy evaluation. Some components of such a list may be well defined, while other components may be undefined (i.e. cause infinite loops if they are accessed).

3. Analysis

The work presented in [Ref. 6] provides a fundamental basis for most of the current work in semantics-based program integration and merging. It looks at programs in terms of their semantic building blocks and provides a theory describing how merging occurs at the building block level. This work shows that computing a useful approximation to an ideal merge is both achievable and sufficient.

C. INTEGRATION OF CHANGES TO WHILE-PROGRAMS

In [Ref. 28], the first semantics-based algorithm for integrating two non-interfering modifications of a base program is described. This integration algorithm produces a third program which reflects both modifications, and uses *program dependence graphs* (PDGs) to abstractly represent the programs. Using *program slicing*, it then determines which portions of the two modifications are different from the base program. Based on this information, the algorithm uses a conservative approximation to determine if the changes can interfere. If they can not, the program slices are combined into one integrated PDG, which is then transformed into a final version of the integrated program.

1. Program Dependence Graphs

A PDG for a program P, as described in [Ref. 28], is a directed graph, G_P, with vertices representing statements in the program, and edges representing control and data dependencies between the vertices. There are also two special types of vertices in the PDG which are not program statements; an *entry* vertex and a *final-use* vertex for each output variable. A complete list of the types of vertices is contained in [Ref. 28].

Using these components, a PDG can be constructed for any while-program [Ref. 33]. Figure 3.2 shows an example of a simple program and its associated PDG. By analyzing the

16

parts of this graph that affect a certain variable, we are able to observe the effects of a change to the program with respect to that variable. This is done using program slicing [Ref. 47].

Figure 3.2: Example of a Program Dependence Graph [Ref. 28]

2. Program Slicing

The program slice of a graph G with respect to a vertex s is the subgraph of G induced by all vertices that can reach s by way of control ($\longrightarrow_c$) or flow ($\longrightarrow_f$) dependence edges, along with the edges that connect the vertices.

$$V(G/s) = \{w \in V(G) \mid w \longrightarrow_* s\}$$

To get the slice of a graph G with respect to one of the output variables, say x, merely take the slice with respect to the vertex labeled $FinalUse(x)$. The slice is constructed backward from the *final-use* vertex, and includes all control or flow edges which

can contribute to the final value of x. Def-order edges are contained in the slice only if the vertex which observes the dependency is also included in the slice. This construction can be extended to a set of vertices $S = \{s_1, s_2, ..., s_j\}$ by taking the union of the vertex and edge sets of all of the individual program slices. Figure 3.3 shows an example of the slice of the previous program taken with respect to the variable x at the final-use node and the corresponding PDG.
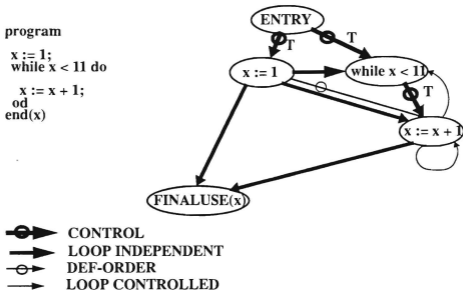
Figure 3.3: Example of a Slice of a Program Dependence Graph [Ref. 28]

3. Integration Algorithm

The integration algorithm presented in [Ref. 28] starts by creating program dependence graphs for each program and, using program slicing, identifies the part of the base program which is preserved in all three versions and the parts of the variations which are different from the base. The common part of all three versions is called the *preserved part*,

18

and the part of each variation that is different from the base is called the *affected part* of that variation.

These three slices are then combined into an integrated PDG. If the integrated PDG is feasible[1] and the two variants do not interfere with each other, then the integration is successful. One major problem identified in this work is that determining whether a PDG is feasible is NP-Complete [Ref. 28]. The other criterion for determining success is more tractable, that of determining *interference*. This is done by comparing the slices of each of the three original versions against slices in the merged version. If the slice of the merged version with respect to the affected parts of each modification is the same as the slice of that modification with respect to its affected parts, and the slice of merged version with respect to the preserved part is the same as the base version with respect to the preserved part, then the versions do not interfere, and a successful integration is possible.

The work in [Ref. 28] is supported by three theorems; the slicing theorem, the equivalence theorem and the integration theorem. The slicing theorem states that when given the same input and starting state, a slice of a program that halts produces precisely the same output as the program. The equivalence theorem states that if two programs have equivalent PDGs, then the programs are themselves equivalent. The integration theorem states that if M is the result of a successful integration, then M halts on any initial state on which the three input versions halt, and M correctly preserves the meaning of each modification to the base.

4. Meaning Functions

Meaning functions [Ref. 33] represent the semantic meaning of a program as mappings from states to states. These state changes are represented as sets of pairs including an initial state and the corresponding final state(s). In [Ref. 10], Berzins provides a theoretical

---

[1] A program dependence graph is feasible if it is a PDG for a program.[Ref. 28]

19

foundation for merging simple, imperative programs using their meaning functions. This theory uses the notion that program variations can be viewed as partial functions modeled using a powerset lattice. Since a powerset lattice is equivalent to a Boolean algebra, normal set operations, $\cup$, $\cap$ and $-$ can be used to reason about these program variations.

This theory shows that a change transformations from a base program f to a variation g, $\Delta[f, g]$, can be applied to a second variation h, $\Delta[f, g](h)$ with precisely the same results as if the change from f to h were applied to g, $\Delta[f, h](g)$. This is very useful in change-merging, as it demonstrates that independent updates to a common base version g of a software product and subsequently change-merged without regard for the order in which they were accomplished. As long as the changes made are compatible, the results in terms of the meaning functions are the same. It does show, however, that the change-merged program does not necessarily have to be similar to the input programs.

The meaning functions for the programs shown in Figure 3.4 are as follows:

$$m(B) = (x > 0 \rightarrow \{((x, y), (x, 1))\} \mid x \leq 0 \rightarrow \{((x, y), (x, -1))\})$$
$$m(A) = (x > 0 \rightarrow \{((x, y), (x, 1))\} \mid x \leq 0 \rightarrow \{((x, y), (x, 0))\})$$
$$m(C) = (x > 0 \rightarrow \{((x, y), (x, x))\} \mid x \leq 0 \rightarrow \{((x, y), (x, -1))\})$$

These three versions are merged using their meaning functions as follows [Ref. 10]:

$$m(M) = m(A[B]C) = m(A)[m(B)]m(C)\,^2$$
$$= (m(A) - m(B)) \cup (m(A) \cap m(B)) \cup (m(C) - m(B))$$

$$= (x > 0 \rightarrow \{((x, y), (x, 1))\} - \{((x, y), (x, 1))\} \mid$$
$$x \leq 0 \rightarrow \{((x, y), (x, 0))\} - \{((x, y), (x, -1))\})$$
$$\cup (x > 0 \rightarrow \{((x, y), (x, 1))\} \cap \{((x, y), (x, x))\} \mid$$
$$x \leq 0 \rightarrow \{((x, y), (x, 0))\} \cap \{((x, y), (x, -1))\})$$
$$\cup (x > 0 \rightarrow \{((x, y), (x, x))\} - \{((x, y), (x, 1))\} \mid$$
$$x \leq 0 \rightarrow \{((x, y), (x, -1))\} - \{((x, y), (x, -1))\})$$

---

²The notation A[B]C will be introduced in Section D.1

$$= (x > 0 \rightarrow \{\} \mid x \leq 0 \rightarrow \{((x,y),(x,0))\}$$

$$\cup (x > 0 \rightarrow \{((x,y),(1,1))\} \mid x \leq 0 \rightarrow \{\})$$

$$\cup (x > 0 \rightarrow \{((x,y),(x,x)) \mid x \neq 1\} \mid x \leq 0 \rightarrow \{\})$$

$$= (x > 0 \rightarrow \{((x,y),(x,x))\} \mid x \leq 0 \rightarrow \{((x,y),(x,0))\})$$

$$= m(\text{if } x > 0 \text{ then } y := x \text{ else } y := 0) = m(M)$$

Base version B:	if $x > 0$ then $y := 1$ else $y := -1$ fi
First change version A:	if $x > 0$ then $y := 1$ else $y := 0$ fi
Second change version C:	if $x > 0$ then $y := x$ else $y := -1$ fi

Figure 3.4: A Program and Two Variations [Ref. 10]

5. Analysis

The work presented in this section shows that a method can be developed for integrating real programs. The work contained in [Ref. 28, 29, 48] illustrates a method for integrating programs in a simple imperative programming language that has been developed and works. This demonstrates that a practical method is possible for imperative programs, but falls short of providing a method which is useful to solve any real world problems. In particular, the method fails to provide any sort of conflict location or resolution. If a conflict is detected, then it is reported to the user, and the integration fails. It is up to the user to determine the nature of the conflict and how it should be resolved. Our methods address these problems as shown in the next section and in Chapter IV.

D. CHANGE-MERGING OF PSDL PROGRAMS

In [Ref. 20], an initial attempt at developing a model for change-merging PSDL programs is presented. Although crude, this model provides us with an important part of the

specification change-merging model, and insight into the current effort defined in subsequent chapters.

1. Change-Merge Operation

This change-merge operation is defined by the operation $A[B]C$, where A, B and C are sets of pairs representing the functionality of three different versions of a PSDL program. The operation $A[B]C$ was initially introduced by Berzins in [Ref. 9] and is defined as:

$$A[B]C = (A - B) \sqcup (A \sqcap C) \sqcup (C - B)$$

where $\sqcap$, $\sqcup$ and $-$ represent the *greatest common approximation*, *least common extension*, and *semantic difference* respectively, between two programs.

The set of all PSDL programs, together with a $\top$ and $\bot$, forms a lattice using the relation *approximates* [Ref. 20]. If A is an extension of B, then we say that B approximates A, written $B \sqsubseteq A$. The $\top$ element in the lattice is an extension of every PSDL program, and the $\bot$ element approximates all PSDL programs. For example consider the lattice in Figure 3.5. In this example, O and P are extensions of A and A approximates both O and P. P and Q are both extensions of B and B approximates both P and Q. P is a common extension for both A and B. In fact, P is the least common extension of A and B.
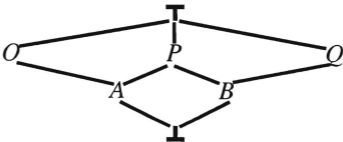
Figure 3.5: A Lattice of Program Extensions

The least common extension of two PSDL programs, $A \sqcup B$, is the smallest possible PSDL program P such that $A \sqsubseteq P$ and $B \sqsubseteq P$, and represents the union of the functionalities found in both A and B. In Figure 3.5, P is the least common extension of A and B. The greatest common approximation of two PSDL programs, $P \sqcap Q$, is the largest possible PSDL program B such that $B \sqsubseteq P$ and $B \sqsubseteq Q$, and represents the common functionality found in both P and Q. In Figure 3.5, B is the least common extension of P and Q. The semantic difference between two programs, $A - B$, represents the functionality found in A, but not in B. The semantic difference exists if the lattice is a Boolean algebra, and a pseudo-difference can be defined if the lattice is a Brouwerian algebra.

It has been shown that the least common extension of two programs is not computable in the general case [Ref. 6]. In [Ref. 20], we demonstrated that an approximation that is computable is sufficient to provide a useful change-merge for most cases. The following sections outline the model defined in [Ref. 20].

2. Interfaces

The interface of a PSDL operator P is the definition of the operator's external contacts. It defines I_P, the set of inputs expected by the operator, O_P, the set of outputs that can be expected, and in the case of generic templates, GN_P, the set of generic parameters used to instantiate the prototype. I_P, O_P, and GN_P are all ordered sets (sequences). The interface may also contain a set St_P, of internal state variables, a set E_P, of possible exceptions, and a maximum execution time constraint that is met by the program. St_P and E_P are sets.

a. *Sequences*

Sequences are a significant building block for many programming languages, including PSDL. A sequence is a totally ordered collection. Since the order of the collection

23

is significant, any change made to the sequence is an incompatible change and creates a sequence which is neither an approximation nor an extension of the original sequence. A correct mathematical representation for a sequence would be a *flat lattice*, like the one in Figure 3.6. This means that the only approximation for the sequence is the undefined sequence, $\perp$, and the only extension of the sequence is the unconstrained set, $\top$, and the greatest common approximation of any two sequences is the undefined element, $\perp$.
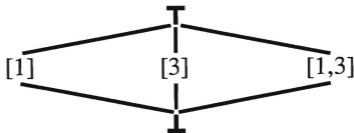
Figure 3.6: A Flat Lattice Representation for a Sequence

(1) Input and Output. Input and Output interfaces are sequences of input and output streams. The order of these sequences is significant because actual parameters are associated with formal parameters based on the order in which they appear. In change-merging I_A, I_B, and I_{Base} into I_M, any change between the interface sequence of the base version and the two modified versions is significant, and must be preserved in the change-merged version. The change-merged sequence of inputs, or outputs, is determined by the following rules:

1. If both of the modified versions have the same interface sequence as the base, then: $I_M = I_{Base}$.

2. If one of the two modified versions, say I_A, is the same as the base, and I_B is not, then: $I_M = I_B$.

3. If all three versions are different from each other, then: $I_M = \top$.

24

The first situation is the case in which no changes were made between the inputs of the Base and the two modifications. In this case, the change-merged version should have all of the same inputs, or outputs. The second situation is the case in which only one of the modifications changed from the base. In this case, the change from the base is significant and must be preserved in the change-merged version. The third situation is the case where both of the modifications changed from the base. The result is a conflict because there is no proper PSDL specification that is consistent with both modifications. The result of a change-merge which produces a conflict for this situation would be an input declaration which contains a $\top$ where the input stream declarations would be.

The type declarations of the streams also have to be merged. Because the types are significant, any change to the type declaration must be preserved in the merged version. Types are also change-merged using a flat lattice structure. Figure 3.7 contains an example of a change-merge on Input Sets.

$$S_{Base} = \textbf{INPUT}$$
$$x : integer,$$
$$y : real$$
$$\textbf{OUTPUT}$$
$$w : integer,$$
$$z : string$$

$$S_A = \textbf{INPUT}$$
$$x : integer$$
$$\textbf{OUTPUT}$$
$$w : integer,$$
$$t : integer,$$
$$z : string$$

$$S_B = \textbf{INPUT}$$
$$x : integer,$$
$$y : real$$
$$\textbf{OUTPUT}$$
$$w : integer$$

$$S_M = \textbf{INPUT}$$
$$x : integer$$
$$\textbf{OUTPUT}$$
$$\top$$
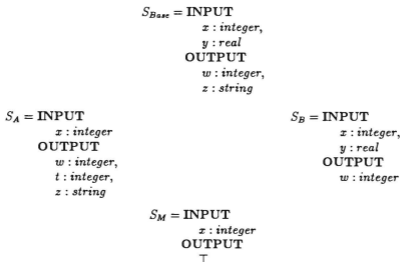
Figure 3.7: Example of a Change-Merge on Input Sets [Ref. 20]

(2) Generic Parameters. The Generic interface is contained only in template operators and PSDL type specifications. Template operators are operators in the Software Base used to instantiate software components. Change-merging generic parameters is similar to change-merging input and output parameters with the exception that, in addition to value parameters, generic parameter sequences may also contain operator parameters and type parameters. Changes to generic sequences follow the same rules as Input and Output sequences. Figure 3.8 shows an example of a change-merge operation on generic parameters.

$$GN_{Base} = \textbf{GENERIC}$$
$$t1 : type,$$
$$t2 : type,$$
$$o1 : operation[i1, i2 : t1, o1 : t2],$$
$$v1 : integer$$

$$GN_A = \textbf{GENERIC} \qquad\qquad GN_B = \textbf{GENERIC}$$
$$t1 : type, \qquad\qquad\qquad t1 : type,$$
$$t3 : type, \qquad\qquad\qquad t2 : type,$$
$$o2 : operation[i1 : t1, o1 : t3], \qquad o1 : operation[i1, i2 : t1, o1 : t2],$$
$$v1 : integer \qquad\qquad\qquad v1 : integer$$

$$GN_M = \textbf{GENERIC}$$
$$t1 : type,$$
$$t3 : type,$$
$$o2 : operation[i1 : t1, o1 : t3],$$
$$v1 : integer$$

Figure 3.8: Example of a Change-Merge on Generic Parameters [Ref. 20]

b. *Sets*

Sets are modeled using a "Powerset Lattice" as shown in Figure 3.9, and thus more freedom can be exercised in change-merging them. Change-merge operations do not follow the same rules for sets as for sequences.
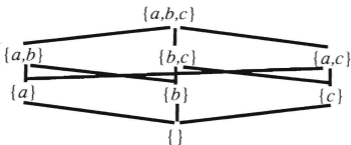
Figure 3.9: A Powerset Lattice Representation for a Set Containing Three Elements

   (1) States.   State variables differ from input and output variables in that, abstractly, they are tuples, containing a name, a type and an initial value. As the set of state variables is unordered and invisible to the rest of the program, the state set can be increased or decreased without affecting the parts of the program outside the modified component. In change-merging state variable sets, the operations ⊓, ⊔, and − are equivalent to the corresponding set operations, ∪, ∩ and −. The third part of the tuple, the initial value, requires an additional check in the change-merging process. These initial values are ordered using a flat lattice, because they are ordinary data values. The initial value of a change-merged state variable follows the same change-merging rules as input and output variables. If all three versions have different initial values for the same state variable, then the change-merged version contains a ⊤ in the place where the initial value is assigned. If only one of the modifications assigns a different initial value than the base version, then the change-merged version contains the initial value of the one that was different.

   (2) Exceptions.   The exceptions interface is a list of identifiers which denote exception values which may be returned by the operator. Consequently ⊓, ⊔, and − can be interpreted as the corresponding set operations, ∪, ∩ and −. Exceptions that appear in one or both of the modified versions, and not in the base, appear in the change-merged program. Exceptions that appear in the base and do not appear in at least one of the modifications are not included in the change-merged program.

27

(3) **Maximum Execution Time.** Maximum Execution Time (MET) is the only timing constraint that appears in PSDL specifications. MET is the maximum CPU time that an operator can use to perform its assigned task. Change-merging two MET constraints, t_1 and t_2, can be done as follows:

$$
\begin{aligned}
t_1 \sqcup t_2 &= min(t_1, t_2) \\
t_1 \sqcap t_2 &= max(t_1, t_2) \\
t_1 \mathbin{\dot{-}} t_2 &= if \ t_2 \leq t_1 \ then \ \infty \ else \ t_1 \\
\top &= 0 \\
\bot &= \infty
\end{aligned}
$$

Proposition 2 *The set of METs form a Brouwerian Algebra*

Proof:

Let $\mathcal{M}$ be the set of all possible METs.

We must show that $\mathcal{M}, \sqcup, \sqcap$ is a distributive lattice, that $\mathcal{M}$ is closed under $\dot{-}$, and that

$\forall a, b, c \in \mathcal{M}, a \mathbin{\dot{-}} b \leq c \Longleftrightarrow a \leq (b \sqcup c)$.

1. $(\mathcal{M}, \sqcup, \sqcap)$ is a distributive lattice:

Clearly, $a \sqcup b$ and $a \sqcap b$ exist for any $a, b \in \mathcal{M}$, and the reflexive, antisymmetric, and transitive properties hold, so $(\mathcal{M}, \sqcup, \sqcap)$ is a lattice.

$\mathcal{M}$ is distributive: Let $a, b, c \in \mathcal{M}$. We use a table to illustrate:

	$a \sqcap (b \sqcup c)$	$(a \sqcap b) \sqcup (a \sqcap c)$
$a \leq b \leq c$	b	b
$a \leq c \leq b$	c	c
$b \leq a \leq c$	a	a
$b \leq c \leq a$	a	a
$c \leq a \leq b$	a	a
$c \leq b \leq a$	a	a

From the table it is easy to see that $\mathcal{M}$ is distributive.

2. $\mathcal{M}$ is closed under $\dot{-}$:

Since $a \mathbin{\dot{-}} b$ is always either a or ∞ for any a and b, $\mathcal{M}$ is certainly closed under $\dot{-}$.

3. For any $a, b, c \in \mathcal{M}$, $a \overset{.}{-} b \leq c \iff a \leq (b \sqcup c)$:

Assume $a \overset{.}{-} b \leq c$. Then, $a \leq b$, since otherwise, $a \overset{.}{-} b = \infty$. Since $a \leq b$, then $a \overset{.}{\leq} c$. Thus, $a \leq (b \sqcup c)$.

Now, assume $a \leq (b \sqcup c)$. Then $a \leq b$ and $a \leq c$, and $a \overset{.}{-} b = a$. Thus $a \overset{.}{-} b \leq c$.

Therefore, $\mathcal{M}$ is a Brouwerian Algebra.

3. Functionality

The functionality of an operator specification is a description of the behavior of an operator. It consists of a set of keywords, an informal description, and/or a formal description. Through the use of keywords, the operator can be distinguished from other operators in the database during the retrieval process. Informal text descriptions are provided for use by the engineer. Formal axiomatic descriptions are provided to support automatic retrieval.

The set of keywords can be change-merged using the appropriate set operations, $\cup$, $\cap$, and $-$. The informal description is a sequence and must be change-merged using the same method described for input and output parameters. Formal descriptions can be change-merged using the Boolean algebra structure of the logic in which they are expressed:

$$
\begin{aligned}
x \sqcup y &= x \vee y \\
x \sqcap y &= x \wedge y \\
x - y &= x \wedge \neg y
\end{aligned}
$$

4. Data Flow Graphs

In [Ref. 20], a PSDL implementation graph for an operator A is viewed as a graph $D_A = \{O, L\}$, where O is a set of vertices that represent the component operators of A, including the constant operator EXT representing external contacts, and where L is a set of links (labelled edges) which represent the data streams entering and leaving the elements of O. The labels for the links are the names of the data streams they represent.

The change-merging operation on PSDL data flow graphs is defined in terms of a bipartite graph $B_A = \{V, S, LI, LO\}$, where V is the set of operators in D_A, S is a set of vertices which represent the data streams of operator A, LI is a set of edges from a stream vertex to an operator vertex, representing input links, and LO is a set of edges from an operator vertex to a stream vertex, representing output links.

Change-merging the data flow diagrams is done by change-merging the graphs G_{Base}, G_A and G_B by subsets V, S, LI and LO. The operations $\sqcup$, $\sqcap$, and $-$ can be interpreted as the corresponding operations $\cup$, $\cap$, and $-$. This change-merge is accomplished using the following equation:

$$G_M = [G_A - G_{Base}] \sqcup [G_A \sqcap G_B] \sqcup [G_B - G_{Base}]$$

This equation defines a structural or syntactic change-merging operation that does not necessarily correspond to a semantic change-merging operation.

The greatest common approximation is obtained for the Base and the two modifications by taking the intersection on all components of the graph. Then these common components are added to the disjoint components of each modification by subtracting out the parts of the two modifications which are also in the base. This operation preserves the parts of the program common to all the versions, while ensuring that significant changes made by the two modifications are included in the change-merged graph.

This method of change-merging the implementation graph of a PSDL program fails to adequately consider the semantic effects of the changed modifications, as does the approximate method shown later in this chapter. Although these methods produce a change-merge that is useful in some cases, they are not nearly as useful as the slicing method described in Chapters IV and V.

5. Data Streams and Control Constraints

a. *Data Streams*

A set of data stream declarations DS_A, defines local data streams that are used only within the implementation of a composite operator, A, and that are not defined in the specification. The order in which the declarations appear is not significant. They have the same structure as exception declarations, and can be change-merged using the same rules. If a stream appears in DS_{Base}, then it appears in DS_M if and only if it appears in both DS_A and DS_B. If a stream does not appear in DS_{Base}, then it appears in DS_M if and only if it appears in at least one of the sets DS_A and DS_B. These rules are:

$$
\begin{aligned}
x \in DS_{Base} \quad &\wedge \quad x \in DS_A \wedge x \in DS_B \quad \implies \quad x \in DS_M \\
x \in DS_{Base} \quad &\wedge \quad \neg(x \in DS_A \wedge x \in DS_B) \quad \implies \quad \neg(x \in DS_M) \\
\neg(x \in DS_{Base}) \quad &\wedge \quad (x \in DS_A \vee x \in DS_B) \quad \implies \quad x \in DS_M \\
\neg(x \in DS_{Base}) \quad &\wedge \quad \neg(x \in DS_A \vee x \in DS_B) \quad \implies \quad \neg(x \in DS_M)
\end{aligned}
$$

The type declarations of data streams are also significant, as with Input and Output Streams, and changes to those declarations must be preserved in the merged version. The type declarations can be merged using a flat lattice structure just as the Input and Output streams are merged.

b. *Control Constraints*

Control constraints are a set of pre-conditions, which control the firing of particular components, and post-conditions, which filter the output provided by those components. The control constraints appear in the change-merged operator according to the same rules as the data stream definitions. Any control constraint that appears in all three input versions in exactly the same way appears in the change-merged operator without change. Any constraint which appears in one or both of the modifications, but not in the base,

appears unchanged as long as the conditions of the constraint are the same. Changes in conditions are handled differently depending on the type of constraint. Input and output guards, conditional exceptions, "TRIGGERED IF", "OUTPUT IF", "EXCEPTION IF", and timer operations have logical predicates as conditions. Timer operations are not change-merged as straightforwardly as other predicate constraints. Different operations exist for different activities. Start, stop, and reset are the three timer operations used in PSDL. The timer operations affect the state of the timer. The start and stop operations affect the run state of the timer, and the reset operation affects the value state of the timer. The reset operation is thus independent of the others, and can be merged independently. If a reset operation appears in all three versions, or appears in at least one of the modifications, but not in the base, then it appears in the changed merged version as well. The start and stop operations must be change-merged using a flat lattice ordering relation, as with inputs and outputs. The predicates that accompany the control constraints are change-merged according to the usual rule, $A[Base]B = (A - Base) \sqcup (A \sqcap B) \sqcup (B - Base)$, where the operations $\sqcap$, $\sqcup$, and $-$ are interpreted as follows:

$$a \sqcup b \implies a \lor b$$
$$a \sqcap b \implies a \land b$$
$$a - b \implies a \land \neg b$$

The constraints "PERIOD", "FINISH WITHIN", "MAXIMUM RESPONSE TIME", and "MINIMUM CALLING PERIOD" have integer values as conditions. These values are ordered using a flat lattice and can be change-merged as follows. For "PERIOD" constraints, if the value is the same in all three input versions, then it appears unchanged in the merged version. If it is different from the base in one of the modifications and the same as the base in the other modification, then the change must be preserved and the value appearing in the modification where it is different appears in the merged version. If all three versions have different values for the period, then a $\perp$ or undefined value appears in the merged version, indicating an unresolvable conflict. "FINISH WITHIN" and "MAXIMUM RESPONSE TIME" constraints are upper bounds and can be change-merged using the

same method described for "MAXIMUM EXECUTION TIME". "MINIMUM CALLING PERIOD" is a lower bound and two MCP constraints, t_1 and t_2, can be change-merged using the equations shown below:

$$
\begin{aligned}
t_1 \sqcup t_2 &= max(t_1, t_2) \\
t_1 \sqcap t_2 &= min(t_1, t_2) \\
t_1 - t_2 &= if \ t_2 \geq t_1 \ then \ \infty \ else \ t_1 \\
\top &= 0 \\
\bot &= \infty
\end{aligned}
$$

Proposition 3 *The set of all MCPs form a Brouwerian Algebra*

Proof: See the proof of Proposition 2.

6. Analysis

The work presented in [Ref. 20] was a first look at providing a change-merging capability for PSDL prototypes. It explored some critical issues in the problem and provided valuable information for work presented later in this dissertation. The work on change-merging specifications has proven to be very valuable and remains virtually unchanged in the current model. Only the parts of the model concerning timing constraints have been improved in the current model. The work on change-merging implementations was unsuccessful in providing a useful method. The next sections provide a look at an improvement over this method.

E. CHANGING PSDL PROTOTYPES

In [Ref. 21], another attempt at formulating a model for representing PSDL implementations is explored. In this model, 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 $\lim_{i \to \infty} S_i = S$.

1. Prototypes as Graphs

Each prototype implementation S_i is modeled as a graph $G_i = (V_i, E_i, C_i)$, where:

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

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

2. Changes to Graphs

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:

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

- $\Delta S_i = (VA_i, VR_i, EA_i, ER_i, CA_i, CR_i)$ 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.

$S_{i+1} = S_i + \Delta S_i$ is defined in terms of the individual components of S_{i+1} 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 3.10 contains a graph representation for a composite operator $Op1$ consisting of 4 vertices and 6 data streams. Figure 3.11 shows a change to be applied to $Op1$ to produce $Op2$. Figure 3.12 shows a graph representation of $Op2$, the result of applying the change to $Op1$.
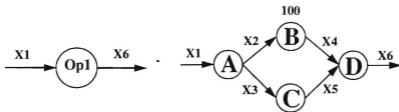
$$Op1 = \{V_1, E_1, C_1\}$$
$$V_i = \{A, B, C, D\}$$
$$E_1 = \{(X1 : EXT \rightarrow A), (X2 : A \rightarrow B), (X3 : A \rightarrow C), (X4 : B \rightarrow D),$$
$$(X5 : C \rightarrow D), (X6 : D \rightarrow EXT)\}$$
$$C_1 = \{max\_exec\_time(B, 100ms)\}$$

Figure 3.10: Example of a composite operator in PSDL

$$
\begin{aligned}
\Delta_A Op1 &= \{VR_A, VA_A, EA_A, ER_A, CA_A, CR_A\} \\
VA_A &= \{E\} \\
VR_A &= \{C\} \\
EA_A &= \{(X3 : A \to E), (X7 : E \to D)\} \\
ER_A &= \{(X3 : A \to C), (X5 : C \to D)\} \\
CA_A &= \{latency(X7, E, D, 50ms)\} \\
CR_A &= \{\}
\end{aligned}
$$

Figure 3.11: Example of a change made to a composite operator in PSDL

Operator Op2 = Op1 + Δ_AOp1
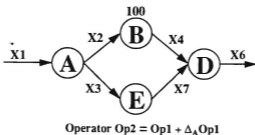
$$
\begin{aligned}
Op2 &= \{V_2, E_2, C_2\} \\
V_2 &= \{A, B, D, E\} \\
E_2 &= \{(X1 : EXT \to A), (X2 : A \to B), (X3 : A \to E), (X4 : B \to D), \\
&\quad (X7 : E \to D), (X6 : D \to EXT)\} \\
C_2 &= \{max\_exec\_time(B, 100ms), latency(X7, E, D, 50ms)\}
\end{aligned}
$$

Figure 3.12: Example of the changed operator

F. AN APPROXIMATE METHOD FOR CHANGE-MERGING PSDL PROTOTYPES

1. Method

In [Ref. 21, 23, 25], an approximate method for change-merging PSDL prototypes is explored. This method is useful in providing a rough approximation to the ideal change-merge, but was abandoned in favor of the more useful (and provably correct) slicing method [Ref. 23, 24]. It is included to record the effort expended in this endeavor.

Recall the merging function introduced in [Ref. 9], and reintroduced in section D:

$$\mathbf{M} = \mathbf{A[B]C} = (\mathbf{A} - \mathbf{B}) \cup (\mathbf{A} \cap \mathbf{C}) \cup (\mathbf{C} - \mathbf{B}).$$

If the semantic function of a program is represented as a set of pairs, then two compatible modifications of a semantic function can be merged using this equation.

In this equation, the union, intersection and difference operations are defined as normal operations on sets. The difference operation, $(\mathbf{A} - \mathbf{B})$ for example, yields the part of the function present in the modification, but not in the base version. The intersection operation yields the part of the function preserved from the base version in both modifications. This model preserves all changes made to the base version, whether extensions or retractions. In this model, two changes conflict if the construction produces a relation that is not a single valued function.

In this section, we outline an approximate method for merging prototypes using the change model described in the previous section and the above definition. This method is approximate, in the sense that the change merging construction is applied to the structure of a PSDL program rather than to the mathematical function it computes. This method is simple, corresponds to common programmer practice, and produces semantically correct results most of the time.

37

The approximate method can be understood as follows. All PSDL implementations are graphs, whose structure roughly models their functionality. We have represented these graphs using sets. Different variations of a prototype are the results of different changes being applied to a common base version. We can merge the two new versions **A** and **C** by applying the change that produced **A** from **B** to version **C**, or by applying the change that produced **C** from **B** to version **A**. The result is the same in either case. Earlier, we expressed the $(i + 1)$st iteration of a software prototype as $S_{i+1} = S_i + \Delta S_i$. Let us consider an ith version which has been changed in two different ways, via Δ_A and Δ_B. The results of these two changes are denoted as S_A and S_B, respectively. Now let us consider a case where the $(i + 1)$st iteration is the result of merging these two changes:

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

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

$$
\begin{aligned}
V_{i+1} &= V_A[V_i]V_B &&= (V_A - V_i) \cup (V_A \cap V_B) \cup (V_B - V_i) \\
E_{i+1} &= E_A[E_i]E_B &&= (E_A - E_i) \cup (E_A \cap E_B) \cup (E_B - E_i) \\
C_{i+1} &= C_A[C_i]C_B &&= (C_A - C_i) \cup (C_A \cap C_B) \cup (C_B - C_i)
\end{aligned}
$$

To demonstrate the concept of the merging operation, we provide the following example: The base prototype is as in Figure 3.13. Change **A** is outlined in Figure 3.14, with the result shown in Figure 3.15. Change **B** is outlined in Figures 3.16 and 3.17. The merging operation is performed in Figure 3.18 and the result is shown in Figure 3.19.

The merge operation outlined in Figure 3.18 involves determining the real effect 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.
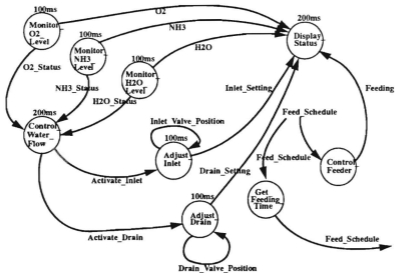
Figure 3.13: Fish Farm Control System, *Fishies*

39

$$\Delta_A Fishies \;=\; VA_A, VR_A, EA_A, ER_A, CA_A, CR_A$$

$$
\begin{aligned}
VA_A \;=\;& \{Monitor\_Bacteria\_Level, Control\_Water\_Flow\_2, Display\_Status\_2\} \\[2pt]
VR_A \;=\;& \{Control\_Water\_Flow, Display\_Status\} \\[2pt]
EA_A \;=\;& \{(Bacteria\_Status : Monitor\_Bacteria\_Level \rightarrow Control\_Water\_Flow\_2), \\
& \;\;(Bacteria : Monitor\_Bacteria_Level \rightarrow Display\_Status\_2) \\
& \;\;(O2_{status} : Monitor\_O2_Level \rightarrow Control\_Water\_Flow\_2), \\
& \;\;(NH3\_Status : Monitor\_NH3\_Level \rightarrow Control\_Water\_Flow\_2), \\
& \;\;(H2O\_Status : Monitor\_H2O\_Level \rightarrow Control\_Water\_Flow\_2), \\
& \;\;(O2 : Monitor\_O2\_Level \rightarrow Display\_Status\_2), \\
& \;\;(NH3 : Monitor\_NH3\_Level \rightarrow Display\_Status\_2), \\
& \;\;(H2O : Monitor\_H2O\_Level \rightarrow Display\_Status\_2), \\
& \;\;(Activate\_Inlet : Control\_Water\_Flow\_2 \rightarrow Adjust\_Inlet), \\
& \;\;(Activate\_Drain : Control\_Water\_Flow\_2 \rightarrow Adjust\_Drain), \\
& \;\;(Inlet\_Setting : Adjust\_Inlet \rightarrow Display\_Status\_2), \\
& \;\;(Drain\_Setting : Adjust\_Drain \rightarrow Display\_Status\_2), \\
& \;\;(Feeding : Control\_Feeder \rightarrow Display\_Status\_2)\} \\[2pt]
ER_A \;=\;& \{(O2\_Status : Monitor\_O2\_Level \rightarrow Control\_Water\_Flow), \\
& \;\;(NH3\_Status : Monitor\_NH3\_Level \rightarrow Control\_Water\_Flow), \\
& \;\;(H2O\_Status : Monitor\_H2O\_Level \rightarrow Control\_Water\_Flow), \\
& \;\;(O2 : Monitor\_O2\_Level \rightarrow Display\_Status), \\
& \;\;(NH3 : Monitor\_NH3\_Level \rightarrow Display\_Status), \\
& \;\;(H2O : Monitor\_H2O\_Level \rightarrow Display\_Status), \\
& \;\;(Activate\_Inlet : Control\_Water\_Flow \rightarrow Adjust\_Inlet), \\
& \;\;(Activate\_Drain : Control\_Water\_Flow \rightarrow Adjust\_Drain), \\
& \;\;(Inlet\_Setting : Adjust\_Inlet \rightarrow Display\_Status), \\
& \;\;(Drain\_Setting : Adjust\_Drain \rightarrow Display\_Status), \\
& \;\;(Feeding : Control\_Feeder \rightarrow Display\_Status)\} \\[2pt]
CA_A \;=\;& \{max\_exec\_time(Monitor\_Bacteria\_Level, 100ms), \\
& \;\;max\_exec\_time(Display\_Status\_2, 100ms), \\
& \;\;max\_exec\_time(Control\_Water\_Flow\_2, 200ms), \\
& \;\;period(Control\_Water\_Flow\_2, 2000ms)\} \\[2pt]
CR_A \;=\;& \{max\_exec\_time(Display\_Status, 100ms), \\
& \;\;max\_exec\_time(Control\_Water\_Flow, 200ms), \\
& \;\;period(Control\_Water\_Flow, 2000ms)\}
\end{aligned}
$$

Figure 3.14: Example of change ΔA applied to *Fishies*
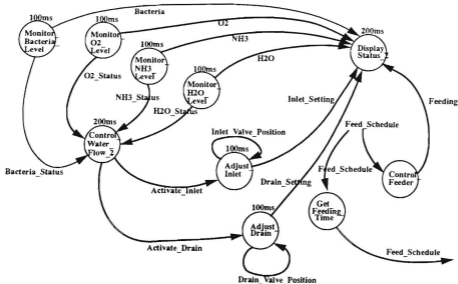
Figure 3.15: $Fishies_A$

$$
\begin{aligned}
\Delta_B Fishies &= \{VR_B, VA_B, EA_B, ER_B, CA_B, CR_B\} \\
VA_B &= \{\} \\
VR_B &= \{Get\_Feeding\_Time\} \\
EA_B &= \{\} \\
ER_B &= \{(Feed\_Schedule : EXT \rightarrow Get\_Feeding\_Time), \\
&\qquad (Feed\_Schedule : Get\_Feeding\_Time \rightarrow EXT)\} \\
CA_B &= \{\} \\
CR_B &= \{\}
\end{aligned}
$$

Figure 3.16: Example of change ΔB applied to $Fishies$
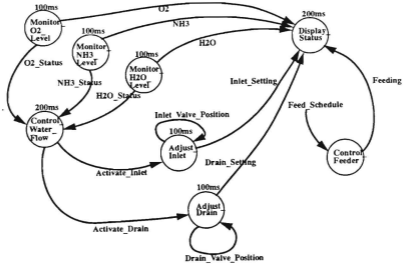
Figure 3.17: $Fishies_B$

$$Fishies_M = Fishies_A[Fishies]Fishies_B =$$
$$(Fishies_A - Fishies) \bigcup (Fishies_A \cap Fishies_B) \bigcup (Fishies_B - Fishies)$$
$$V_{Fishies_M} = V_{Fishies_A}[V_{Fishies}]V_{Fishies_B} =$$
$$(V_{Fishies_A} - V_{Fishies}) \bigcup (V_{Fishies_A} \cap V_{Fishies_B}) \bigcup (V_{Fishies_B} - V_{Fishies})$$
$$E_{Fishies_M} = E_{Fishies_A}[E_{Fishies}]E_{Fishies_B} =$$
$$(E_{Fishies_A} - E_{Fishies}) \bigcup (E_{Fishies_A} \cap E_{Fishies_B}) \bigcup (E_{Fishies_B} - E_{Fishies})$$
$$C_{Fishies_M} = C_{Fishies_A}[C_{Fishies}]C_{Fishies_B} =$$
$$(C_{Fishies_A} - C_{Fishies}) \bigcup (C_{Fishies_A} \cap C_{Fishies_B}) \bigcup (C_{Fishies_B} - C_{Fishies})$$

Figure 3.18: Performing the Change-Merge Operation
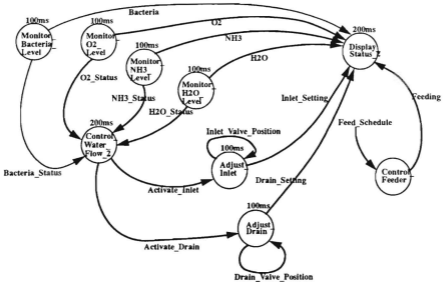
Figure 3.19: $Fishies_M$

There are a number of possible conflicts that can arise during 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. 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 announce that a conflict has occurred and give the designer the opportunity to resolve it. In the case of such a conflict the construction produces a graph that is not well formed, in the sense that it has edges whose endpoints do not belong to the vertex set of the graph and are distinct from the artificial node EXT that serves as an endpoint for external flows.

2. The two changes assign 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 that executes in under 40ms must also 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 result in a conflict that 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.

2. Analysis

The approximate method described above provides a method of change-merging PSDL implementations that is closer to the semantically correct version than the first attempt, but impossible to prove correct. The next two chapters detail a slicing method for

change-merging which is easily proven correct. Chapter IV details a semantic model of PSDL, a method of slicing PSDL programs, and a change-merge model which utilizes these slices to create a merged version which preserves the significant changes in each of the two modified versions. Chapter V details the algorithm developed to implement this slicing method for change-merging. This new slicing method has been implemented and integrated into the CAPS development system.

G. CONDITIONAL MERGING OF WHILE-PROGRAMS

One of the main weaknesses of traditional approaches to data flow analysis and slicing is insensitivity to the conditions under which data flows actually take effect. This problem has prevented conflict-free merging of software changes that affect the same output variable, even in cases where the changes affect disjoint portions of the input space. One way to improve on this is to augment the dependency graphs with flow guards, so that disjoint partial flows can be distinguished, and successfully merged. A software merge technique based on conditional slices captures a finer-grain picture of the threads in a program than merging based on unconditional slices, and hence can produce more accurate program merges.

1. Conditional Flow Dependencies

There is a flow dependency between two statements in a while-program if a value assigned by the first statement can be read by the second statement. Determining flow dependencies exactly is undecidable in the general case [Ref. 13]. Conventional data flow analysis calculates a weak approximation to the exact flow dependencies by assuming that all paths in the control flow graph of a program are feasible. This method ignores the possibility of infeasible paths and non-terminating loops because of its assumption that all control predicates are satisfiable along all possible paths through the control flow graph.

Conventional flow analysis is guaranteed to find all flow dependencies, but it may report some dependencies that are not really there.

In [Ref. 13], Berzins introduces conditional flow dependencies to provide more accurate computable approximations to exact data flow dependencies. A conditional flow dependency is a conventional flow dependency augmented with a predicate describing the conditions under which the data flow can take place. The predicates associated with the data flows enable us to recognize disjoint flows and hence provide a more discriminating model of the data flow dependencies in a program.

a. Flow Guards

The predicates associated with each conditional flow dependency are called *flow guards*. The exact flow guard associated with a flow dependency carried by a variable v from a program statement $s1$ to another program statement $s2$ is true in a program state S if and only if all of the following conditions hold:

1. Statement $s1$ assigns a value to variable v when executed in state S.
2. Program execution will subsequently reach the statement $s2$.
3. Statement $s2$ will read the value assigned by statement $s1$ to variable v.

An approximate flow guard must be true whenever the exact flow guard is true, and can be true in some cases where the exact flow guard is false. The set of all approximate flow guards forms a lattice with respect to the ordering defined by the logical implication relation. The weakest approximate flow guard is true for all states, and the strongest approximate flow guard is the exact flow guard. Conventional data flow analysis is equivalent to using the weakest approximate flow guards.

Checking whether exact flow guards are disjoint is undecidable in the general case, as demonstrated by the program shown in Figure 3.20. Statements are identified by the

line numbers shown on the left margin. The flow guard for the flow of x from statement 1 to statement 4 is disjoint from the flow guard for the flow of y from statement 2 to statement 4 if and only if the program fragment P shown on line 3 terminates, which is an undecidable question. Since program merging algorithms based on conditional flow dependencies need to check whether flow guards are disjoint, we seek representations for which disjointness checks are decidable.

$$
\begin{array}{ll}
1 & x := 1 \\
2 & y := 2 \\
3 & P \\
4 & z := x + y
\end{array}
$$

Figure 3.20: Undecidability of Disjointness for Guard Conditions

We can get approximate flow guards with decidable disjointness relations by using a logic with restricted expressive power to represent the flow guards. One way to do this is to use propositional guard predicates.

Propositional guard predicates are constructed from the Boolean constants true and false, Boolean condition variables associated with the control predicates, the Boolean connectives &, |, and $\sim$, and the modal operators of the form $\langle P \rangle$, where P is the condition variable associated with the control predicate of a while loop in the program.

Propositional guard predicates are interpreted as follows. Condition variables represent the value produced by the most recent evaluation of the associated control predicate. The connectives &, |, and $\sim$ represent the "and", "or" and "not" operators of standard propositional logic.

$$P = (V*) \text{ "begin" } (V*) \text{ "is" } S \text{ "end" } (V*)$$
$$S = V := E$$
$$\qquad | \ S;S$$
$$\qquad | \ \text{"if" } E \text{ "then" } S \text{ "else" } S \text{ "fi"}$$
$$\qquad | \ \text{"while" } E \text{ "do" } S \text{ "od"}$$

Figure 3.21: While Program Grammar

b. Conditional Dependency Graphs

Conditional flow dependencies are represented by a conditional program dependency graph. A conditional program dependency graph consists of a set of vertices and a set of edges. The set of vertices contains a vertex for each assignment statement, an initial-state vertex, and a final vertex for each output variable. The set of edges represent conditional flow dependencies and control dependencies. Control dependency edges are needed to provide a flow path between two sequential parts of a program which do not share any variables. Control dependency edges are identical to flow dependency edges that do not carry a variable.

We illustrate the construction of a conditional flow graph in terms of a simple imperative programming language that provides assignments to scalar variables, sequencing, conditionals, and while loops. This language of while-programs does not have any explicit input or output statements, and is defined by the grammar shown in Figure 3.21.

The nonterminals P, S, E, and V represent while-programs, statements, expressions, and variables, respectively. The Kleene star (*) denotes zero or more instances of the preceding symbol.

The input variables of a while-program are listed before the "begin", and the output variables are listed after the "end". All other program variables are listed between the keywords "begin" and "is". The meaning of a program is characterized by the final values of its output variables. The meaning of a program statement is characterized by its effect

48

on the program state. The program state consists of the values bound to all of the program variables. The meaning of an expression is characterized by the value of the expression in the current program state. The evaluation of an expression cannot affect the program state.

An attribute grammar is provided in [Ref. 13] for constructing the conditional flow graph for a while-program. The nodes of the flow graph correspond to the assignment statements in the program, along with an extra initial vertex and a final vertex for each output variable. Each node is associated with an *execution guard*. The execution guard is a predicate that represents the set of program states in which the statement can be executed. Each edge of the flow graph is associated with a variable name and a *flow guard*. The variable name identifies the data carried by the edge. The flow guard is a predicate that represents the set of program states in which the value of the variable flows along the edge. The flow guard is the conjunction of the conditions that the source node is executed, that the destination node is executed, and that all loops on the control path from the source node to the destination node terminate. Since each node can define the value of at most one variable, there can be at most one edge between any pair of nodes in the flow graph. An example of a Conditional Flow Graph is shown in Figure 3.22.
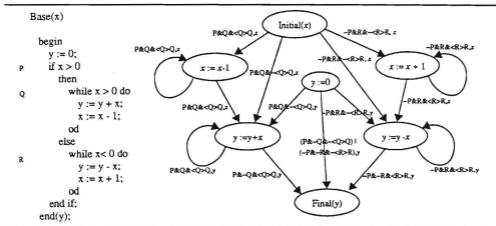
Figure 3.22: An Example of a Conditional Dependency Graph

49

2. Conditional Slices

A *slice* of a program isolates that portion of the code which affects the program behavior with respect to some program statement. A conditional slice must differentiate between portions of the code that affect the meaning of that program statement, but under different conditions. We define a conditional slice of a while-program, with respect to a program statement and a flow guard, on the program's conditional dependence graph, G.

For program statement, S and flow guard, P, the slice of G with respect to S and P, $G/\{S, P\}$ is a subgraph of G and contains all vertices $v_i \in G$, such that there is a path from v_i to S along control dependence edges or flow dependence edges not labeled with the flow guard $\sim P$. The edges in the slice are all of the edges that connect the vertices in the slice. An example of a conditional slice is shown in Figure 3.23.
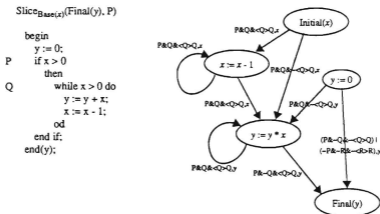
Figure 3.23: Slice $Base/\{\mathbf{Final(y)}, P\}$

50

A conditional slice of a program is itself a program, as it contains all of the original program code which affects the values computed at the final vertex when the input is restricted to only those inputs which satisfy the given conditions.

3. Conditional Program Merging

Other approaches to merging while-programs use pieces of each of the input versions to perform the merge[Ref. 28, 48]. One of these program pieces is the part of the two modified versions which is the same. This part is known as the *preserved part.* The remainder of the merged program comes from that part of each of the modified versions which is different from the base. These parts are called the *affected parts* of each modification. Construction of these program pieces is done using program slicing.

The preserved part is constructed by comparing slices of each of the modified versions with respect to subsets of the program statements. The largest subset of program statements that has the same slice in all three versions defines the preserved part. The affected part of each of the modified versions is constructed by comparing the slice of the modification with respect to each of its vertices against the same slice of the base version. If the slices are different in the modification and the base, then that slice is in the affected part.

One of the problems inherent in this method of program merging is its inability to distinguish between different changes to the same slice which cannot interfere. Conditional slicing alleviates this problem by allowing the different computation paths which can never be executed for the same input to be considered separately.

Using conditional slicing, we calculate the affected part of a modified version by comparing slices of the modified version with respect to the program statements and the set of all possible truth values of the conditional guard predicates at that statement. In this way, two different paths to the same statement which cannot be taken on a single input are

not contained in the same slice, thus changes to one path do not necessarily affect the slice containing the other path.

The merged program is then constructed in the same way as the unconditional method, by taking the graph union of the preserved part and the affected parts of both modifications.

Consider the example outlined in Figures 3.24 through 3.29. In this example, the base version is the same as that shown in Figure 3.22 and contains a conditional expression that partitions the input space into positive and negative integers. If the input value of x is negative, then one set of statements is executed and if it is positive, then another set of statements is executed. In Figure 3.24, you see a change made to the then branch of the conditional expression. In Figure 3.25, you see a change made to the else branch of the conditional. Since both of these branches affect the same output variable, y, the traditional approach to merging would report a conflict and the merge would fail. This should not be the case, however, since these two changes can never interfere.

Figure 3.24: Version A

```
B(x)

    begin
        y := 0;
P       if x > 0
        then
Q           while x > 0 do
                y := y + x;
                x := x - 1;
            od
        else
R           while x < 0 do
                y := y / x;
                x := x + 1;
            od
        end if;
    end(y);
```
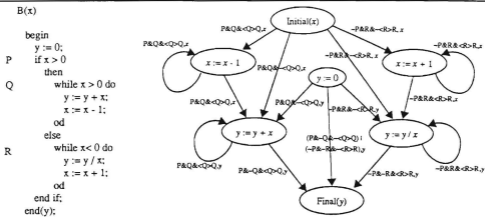

Figure 3.25: Version B
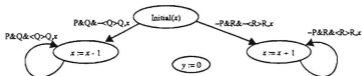
Figure 3.26: Preserved Part of all Three Versions

53
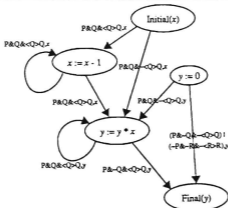
Figure 3.27: Affected Part of Version A
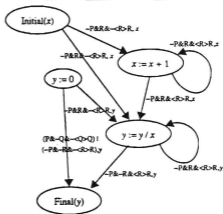
Figure 3.28: Affected Part of Version B

```
Merge(x)

        begin
          y := 0;
P       if x > 0
          then
Q           while x > 0 do
              y := y * x;
              x := x - 1;
            od
          else
R           while x < 0 do
              y := y / x;
              x := x + 1;
            od
          end if;
        end(y);
```
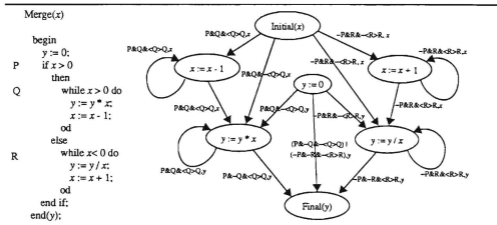
Figure 3.29: Merged Version

55

IV. SEMANTIC MODEL

In this chapter we describe our model for the behavior of prototypes, present our slicing method for change-merging prototypes, and present an invariance theorem that guarantees our method is correct.

A. PROTOTYPING SYSTEM DESCRIPTION LANGUAGE

The Prototyping System Description Language (PSDL) is an enhanced data flow language that can be used to specify and implement prototypes of real-time embedded software. PSDL programs are inherently non-deterministic and can be executed in parallel [Ref. 32]. This section describes a semantic execution model for PSDL programs.

1. Overview of PSDL Semantics

Our change-merging method is based on the behavior of the input programs and not on their syntax. In this section we define a behavior model for PSDL that we can use to prove our invariance theorem. The semantics of PSDL have been modeled using algebraic high-level Petri nets [Ref. 32]. We chose a different model which is more applicable to our problem. We chose to model the behavior of a prototype by observing the data flow history over its data streams. A prototype's behavior is represented by sets of possible histories over the streams we call *trace_tuples*. These trace_tuples are composed of sequences of data_tuples called *traces*. Each trace_tuple contains precisely one trace per stream. Since PSDL prototypes are non-deterministic, one trace_tuple does not necessarily reflect the set of possible histories associated with a prototype, thus we must consider the behavior of a prototype to be the set of all possible trace_tuples over its data streams. Since PSDL prototypes are intended

to prototype embedded real-time systems, which may never be turned off, this behavior is likely to be of infinite length. We use trace_tuples as the base unit for our inductive proof[2] of the invariance theorem in Section B.2 of this chapter. The following subsections describe the model starting with traces and building up to the behavior of a prototype, and the possibility functions we use to construct the behaviors.

2. Traces

The history of a PSDL computation can be described by the histories of all the data streams, called traces. A *trace* on a data stream x, denoted τ_x, is the sequence of all data tuples on the stream. Each *data tuple* contains a data element x_i, the name o_i of the operator responsible for writing x_i to the stream, the time tw_i that x_i was written to the stream, and the time tr_i at which o_i read its input streams to start the computation that produced x_i. A data tuple represents the assertion that the value x_i was produced by an execution of o_i that started at time tr_i and finished at time tw_i.

Example 1 *Trace on a stream x*

$$\tau_x = [[x_0, o_0, tr_0, tw_0], [x_1, o_1, tr_1, tw_1], ..., [x_i, o_i, tr_i, tw_i], ...]$$

Since PSDL was designed for writing prototypes of real-time embedded software systems that may never be turned off once started, traces can be finite or countably infinite. The initial data tuple on a data stream is $[x_0 \rightarrow \bot, o_0 \rightarrow \bot, tw_0 \rightarrow 0, tr_0 \rightarrow 0]$, where $\bot$ represents an undefined value, unless the stream is declared as a state variable with an initial value, in which case the initial data tuple would be $[x_0 \rightarrow v, o_0 \rightarrow \textbf{DECL\_OP}, tw_0 \rightarrow 0, tr_0 \rightarrow 0]$. $\textbf{DECL\_OP}$ is the operator in which the state declaration appears, and v is the initial value assigned in that declaration. For example, if the state stream is declared in an operator p by the declaration statement $\textbf{STATE } x \textbf{ INITIALLY }$ 3, then the initial data tuple on the stream would be $[3, p, 0, 0]$. Since every trace contains an initial data tuple, we see that all traces are non-empty and that the minimum length of a trace is one. In a

57

data flow stream, when a data element is removed from the stream and there is not another element on the stream, the value and the operator name elements are replaced by $\perp$.

The write times tw_i for a given stream form a monotonically increasing sequence of numbers that represent the amount of time elapsed between when the prototype began execution and when the value was available on the stream. The read times tr_i for a given stream form a monotonically increasing sequence of numbers; the ith element in the sequence represents the amount of time elapsed between when the prototype began execution and when the operator o_i read its input streams at the start of the computation responsible for x_i.

If an operator fails to terminate on any firing, then the trace on any of its output streams contains only the values which were written to the stream before the firing in which the operator failed to terminate. If the failure to terminate occurs during the first firing and no other operator can write to the stream, the trace contains only the data tuple representing the initial value.

A trace, τ_x can also be represented by a stream function from a *write time* to a *triple* containing the *value*, the id of the *operator* which wrote the value and the *read time*: $\Psi : TIME \longrightarrow TYPE(x) \times OP\_ID \times TIME$, where $TYPE(x)$ denotes the set of all possible values that can be written to the stream x, OP_ID is the set of all possible operators that can write to the stream, and $TIME$ is a non-negative real number. We chose time to be a continuous value since prototypes can be executed in parallel, and we cannot guarantee that different processors will execute a precisely the same speeds.

Example 2 *Stream Function Representation for a Trace*

A trace for a stream x is:

$$\tau_x = [[\perp, \perp, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]]$$

58

The stream function representation for this trace would be:

$$\Psi_x = \begin{cases} [0,3) & \longrightarrow & [\bot, \bot, 0] \\ [3,7) & \longrightarrow & [x_1, o, 2] \\ [7, \infty) & \longrightarrow & [x_2, p, 6] \end{cases}$$

In order to use these different representations interchangeably, we need to show that they are equivalent. Consider the function, Φ, shown in Figure 4.1. Φ is a bijection that maps a sequence of data tuples into a step function, Ψ, which is continuous only from the right.

$$\Psi(t) = \lim_{t' \to t^+}, \forall t$$

Limits from the left are not preserved at the boundaries between the data tuples, however.

Theorem 2 Φ *is well-defined and a bijection when restricted to right continuous functions with countable range sets.*

<u>Proof</u>: See Appendix C.

$\Phi(\tau_x) = \Psi_x$, where $\Psi_x(t) = [x_\mathbf{n}, o_\mathbf{n}, tr_\mathbf{n}]$
      where $[x_\mathbf{n}, o_\mathbf{n}, tw_\mathbf{n}, tr_\mathbf{n}] \in \tau_x$, $n \in \aleph$ & $tw_\mathbf{n} \le t$ & ($\mathbf{n}$ = length(τ_x) or $tw_{\mathbf{n+1}} > t$)

$\Phi^{-1}(\Psi_x) = \tau_x$ where $[x_\mathbf{i}, o, tw_\mathbf{i}, tr_\mathbf{i}] \in \tau_x$ iff

$$\Big(\ \Psi_x(tw_\mathbf{i}) = [x_\mathbf{i}, o, tr_\mathbf{i}] \ \ \& \ \ tw_\mathbf{i} = \min_t(\Psi_x(t) = [x_\mathbf{i}, o, tr_\mathbf{i}]) \ \Big)$$

Figure 4.1: $\Phi : Traces \longrightarrow FunctionRepresentations$

The meaning of an operator is characterized by a relation between the traces on the input streams and the traces on the output streams. If a data stream receives input from more than one producer, then we must have a method for merging multiple traces into one to determine the behavior of the entire system. The *merge* function, defined in Appendix A, Section 3, provides this method for two traces, A and B. It can easily be generalized to any finite number of traces.

59

A *stream behavior* for a stream x, β_x, is the set of all possible traces for x. Since a PSDL computation can be non-deterministic, the history of a computation is represented by the set of all possible traces for a given PSDL stream. Since the complete stream behavior for a data stream in a PSDL prototype may not be visible from outside the prototype, it is necessary for us to consider both visible and generated stream behaviors for a stream. A *visible* stream behavior for a stream x is a set of traces written to x by an external producer. Each trace in the visible stream behavior of x is a subsequence of some trace in the complete stream behavior for x. The part of the stream behavior which is not produced externally, we call the *generated* stream behavior. The traces in the generated stream behavior for x are also subsequences of traces in the complete stream behavior for x. For example, consider either of the prototypes in Figure 4.2. Each trace in the stream behavior of x, is a sequence which contains as subsequences the traces on the hidden and visible parts of x. Thus the visible behavior and the generated behavior are both projections of the complete stream behavior.

A *truncated* trace for a stream x, $\tau_x \mid k$, is a finite prefix of τ_x for which $length(\tau_x \mid k = min(length(\tau_x), k)$ A *truncated* stream behavior for a stream x, $\beta_x \mid k$ is the set of all possible truncated traces, $\tau_x \mid k$.

3. Trace Tuples and Prototype Behaviors

A *trace tuple* is a tuple containing a trace for each stream in a prototype. A trace tuple can be projected downward to any subset of the streams in a prototype, say X, by including in the projected trace tuple only those traces on the streams in X. A trace tuple, T, projected downward to a subset X of the streams of the prototype is represented as T_X.
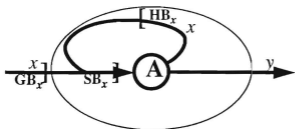
HB - Hidden Behavior
GB - Given Behavior
SB - Stream Behavior
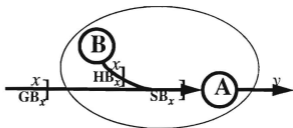
Figure 4.2: Example of prototypes with generated stream behaviors.

An example of a trace tuple over the set of data streams in a prototype $E(P)$ is $\langle \tau_{x_1}, \tau_{x_2}, ..., \tau_{x_k} \rangle$, $x_i \in E(P)$. A *visible* trace tuple is a tuple of visible traces for each stream in the prototype. A *truncated* trace tuple is a trace tuple containing only truncated traces:

$$\langle \tau_{x_1}, ..., \tau_{x_j} \rangle \mid k = \langle (\tau_{x_i} \mid k), ..., (\tau_{x_j} \mid k) \rangle$$

A trace tuple can also be viewed as a vector-valued stream function by extending the function Φ to trace tuples according to the rule:

$$\Phi(\langle \tau_{x_i}, ..., \tau_{x_j} \rangle)(t) = \langle \Phi(\tau_{x_i})(t), ..., \Phi(\tau_{x_j})(t) \rangle = \Psi(t)$$

$\Psi(t)$ is a vector containing one data tuple from each trace in the trace tuple, the value present on each stream at time t. Using Φ, we can also view a trace tuple as a sequence of vectors, where each vector contains the data tuple present on each stream at a write time t for one of the streams in the tuple.

Example 3 *Example of a Trace Tuple on two streams.*

For a set of streams $X = \{x, y\}$ with $\tau_x = [[\bot, \bot, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]]$ and $\tau_y = [[\bot, \bot, 0, 0], [y_1, o, 3, 5], [y_2, o, 7, 9]]$, the resulting trace tuple is:

$$\langle [[\bot, \bot, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]], [[\bot, \bot, 0, 0], [y_1, o, 3, 5], [y_2, o, 7, 9]] \rangle$$

and the corresponding function representation is:

$$
\begin{array}{rcl}
[0, 3) & \longrightarrow & \langle [\bot, \bot, 0], [\bot, \bot, 0] \rangle \\
[3, 5) & \longrightarrow & \langle [x_1, o, 2], [\bot, \bot, 0] \rangle \\
[5, 7) & \longrightarrow & \langle [x_1, o, 2], [y_1, o, 3] \rangle \\
[7, 9) & \longrightarrow & \langle [x_2, p, 6], [y_1, o, 3] \rangle \\
[9, \infty) & \longrightarrow & \langle [x_2, p, 6], [y_2, o, 7] \rangle
\end{array}
$$

Since PSDL is non-deterministic, there may be many possible trace tuples for a prototype P. We call the set of all possible trace tuples for the data streams in P, the *prototype behavior* of P, and we represent it as **B**.

A prototype behavior can also be projected downward to any subset of the streams in a prototype, say X, by including in the projected data flow history only the possible projected trace tuples over X. A projected data flow history over a set of streams X is represented as $\mathbf{B}_X$. An *input* prototype behavior for a prototype P is the set of all possible visible trace tuples over the streams in P.

Example 4 *Example of a projected prototype behavior on a set containing two streams.*

Consider the following set $X = \{x, y\}$, where the stream behavior for x is a single trace, $[[\perp, \perp, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]]$ and the stream behavior for y contains two different traces, $\{[[\perp, \perp, 0, 0], [y_1, o, 3, 5], [y_2, o, 7, 9]], [[\perp, \perp, 0, 0], [y_1, o, 4, 6], (y_2, o, 6, 8)]\}$. Then the resulting prototype behavior, $\mathbf{B}_X$ is:

$$\{\langle[[\perp, \perp, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]], [[\perp, \perp, 0, 0], [y_1, o, 3, 5], [y_2, o, 7, 9]]\rangle,$$

$$\langle[[\perp, \perp, 0, 0], [x_1, o, 2, 3], [x_2, p, 6, 7]], [[\perp, \perp, 0, 0], [y_1, o, 4, 6], [y_2, o, 6, 8]]\rangle\}$$

The function representation corresponding to $\mathbf{B}_X$ is:

$$\begin{array}{rcl}
\{ \ [0, 3) & \longrightarrow & \langle[\perp, \perp, 0], [\perp, \perp, 0]\rangle \\
[3, 5) & \longrightarrow & \langle[x_1, o, 2], [\perp, \perp, 0]\rangle \\
[5, 7) & \longrightarrow & \langle[x_1, o, 2], [y_1, o, 3]\rangle \\
[7, 9) & \longrightarrow & \langle[x_2, p, 6], [y_1, o, 3]\rangle \\
[9, \infty) & \longrightarrow & \langle[x_2, p, 6], [y_2, o, 7]\rangle, \\
\\
[0, 3) & \longrightarrow & \langle[\perp, \perp, 0], [\perp, \perp, 0]\rangle \\
[3, 5) & \longrightarrow & \langle[x_1, o, 2], [\perp, \perp, 0]\rangle \\
[5, 6) & \longrightarrow & \langle[x_1, o, 2], [y_1, o, 4]\rangle \\
[6, 8) & \longrightarrow & \langle[x_2, p, 6], [y_1, o, 4]\rangle \\
[8, \infty) & \longrightarrow & \langle[x_2, p, 6], [y_2, o, 6]\rangle \ \}
\end{array}$$

A *truncated* prototype behavior, $\mathbf{B} \mid k$ is the set of all possible truncated trace tuples, up to length k.

To prove our slice behavior invariance theorem, we also need to extend truncated trace tuples of length k to length $k + 1$ by adding one data tuple onto selected traces in

63

the trace tuple. We define an *incremental trace tuple* to be a vector of sequences of data tuples over a set of streams, where the length of each sequence is either zero or one, and the write times for all of the data tuples are the same. An incremental trace tuple represents the output caused by one firing of a set of zero or more operators writing to different streams of the prototype. We need a function $\oplus$ for appending sets of possible incremental trace tuples on to the end of a truncated trace tuple. The $\oplus$ function is defined in Appendix A. This function takes as operands, a truncated trace tuple over the streams in a prototype and a set of possible incremental trace tuples over the streams in the prototype, and it produces the set of all possible trace tuples resulting from adding each of the incremental trace tuples onto the end of the corresponding sequence in the original trace tuple, for each data stream. Figure 4.3 shows a summary of the constructs defined in the semantic model of PSDL.

$$
\begin{array}{lcl}
time & = & \{x \in \Re \mid x \geq 0\} \\
data\_tuple\{t : type\} & = & tuple\{x : t, o : op\_id, tr, tw : time\} \\
trace & = & sequence\{data\_tuple\} \\
stream\_behavior & = & set\{trace\} \\
trace\_tuple & = & tuple\{stream_i : trace\} \\
incremental\_trace\_tuple & = & vector\{t : trace\} :: length(t) \leq 1 \\
prototype\_behavior & = & set\{trace\_tuple\}
\end{array}
$$

Figure 4.3: Summary of Model Constructs

4. Possibility Functions

Each operator in a PSDL prototype has an input history and an output history. The *input history* of an operator o is defined as the prototype's behavior projected over the input streams of o, $\mathbf{B}_{I(o)}$, and the *output history* of o is the set of all possible trace tuples written by o to its output streams.

In a PSDL prototype, when an operator fires, it reads one data value from each of its input streams and writes at most one output value to each of its output streams. The data

values written and the streams they are written to are determined by the semantic meaning of the PSDL operator and the associated control constraints. Since PSDL operators are non-deterministic, their meanings are possibility functions. For every possible input, there is a set of possible outputs.

To define the possibility function for an operator o, we look at a trace tuple projection of the behavior $\sigma \in \mathbf{B}_{I(o)}$ as a sequence of input vectors to o. For every finite prefix of σ applied to o, the result is a set of possible incremental trace tuples over the output streams of o. This is the *possibility function* for o, $\mathcal{F}_o : \mathbf{B}_{I(o)} \rightarrow \mathbf{B}_{O(o)}$. $\mathcal{F}_o$ takes as input a projected trace tuple over the input streams of o and a read time, and produces a set of possible behavior projections over the output streams of o. The read time is the time at which the last read operation was performed by o on its input streams, and defines which values were read by o to perform this computation.

Example 5 *Possibility function for an operator p which implements the function:*

$$y_k = x_k{}^2$$

$$\mathcal{F}_p = \{(\{3\}, 9), (\{3, -4\}, 16), (\{3, -4, 9\}, 81), ..., (\{3, -4, 9, ..., x_k\}, x_k{}^2), ...\}$$

Example 6 *Possibility function for an operator q which implements the state machine:*

$$y_k = \sum_{i=1}^{k} x_i$$

$$\mathcal{F}_q = \{(\{3\}, 3), (\{3, -4\}, -1), (\{3, -4, 9\}, 8), ..., \left(\{3, -4, 9, ..., x_k\}, (\sum_{i=1}^{k-1} x_i) + x_k \right), ...\}$$

In example 5 you will notice that the y value of each pair is dependent only on the most current value written to the input stream x. In example 6 the y value of each pair is dependent not only on the most current value written to the input stream x, but also on the previous value of y.

The effects of all PSDL control constraints can be expressed as transformations on the possibility function of a bare primitive operator. The effect of each type of control constraint on a possibility function is defined explicitly in Appendix B. In the rest of this chapter, we assume that the possibility function for each operator includes the effects of any associated control constraints.

To analyze the effects of various approaches to change merging, we assume that the possibility function for a network of PSDL operators can be derived from the possibility functions for the individual operators in the network. This can be done as follows.

We consider a prototype P to be a network of operators connected by the data streams of P, with behavior $\mathbf{B}$. $\mathbf{B}$ is the behavior of the entire prototype. Each operator contributes to this behavior by reading from its input streams and writing to its output streams. The values written are determined by the possibility function of the operator. We can derive the possibility function for the prototype P from the possibility functions of the individual operators using the following construction:

$$\mathcal{F}_P = \bigcup_{T \in \mathbf{B}} \left[\bigcup_{S \in \mathcal{P}(V(P))} \left(\bigoplus_{o \in S} \left(\bigcup_{\rho(T,o) < tr} \left(\bigcup_{tr < t} \Delta \left(t, fill \left(E(P), \mathcal{F}_o(T_{I(o)}, tr) \right) \right) \right) \right) \right) \right]$$

This construction produces a set of incremental_trace_tuples over all of the streams in P. The possibility function for each individual operator $\mathcal{F}_o$ is at the heart of this construction. It produces a set of incremental_trace_tuples over its output streams. This incremental_trace_tuple is extended to cover all of the streams in the prototype by the function $fill$. The function Δ is then used to isolate each incremental_trace_tuple attributable to a particular read time tr and these are combined over all possible read times up to the current time t. These incremental_trace_tuples are then combined using the function ρ to pick out the latest possible write time or read time depending on whether the operator contains a feedback loop. Each of these sets of incremental_trace_tuples for individual operators are then combined with sets produced by other operators in the subset S using the function $\oplus$. This is done for every possible subset S in the powerset of the vertices of P. Finally, these

66

sets of incremental_trace_tuples are combined for every possible trace_tuple in the behavior B of P.

To construct the truncated behavior of P of size k, we have to not only produce the set of incremental_trace_tuples, we have to append them to the set of truncated trace tuples of size $k - 1$. That can be done as follows:

$$\mathbf{B} \mid k = \bigcup_{T \in \mathbf{B}|(k-1)} \left[\bigcup_{S \in \mathcal{P}(V(P))} \left[T \oplus \left(\bigoplus_{o \in S} \left(\bigcup_{\rho(T,o) < t_r} \left(\bigcup_{t_r < t} \Delta \left(t, fill \left(E(P), \mathcal{F}_o(T_{I(o)}, t_r) \right) \right) \right) \right) \right) \right] \right]$$

This construction is identical to the previous construction up to the point where the combination over subsets occurs. At this point, we must append the set of incremental_trace_tuples produced by the $\oplus$ function for each subset of the operators to the trace_tuple T in the truncated behavior $\mathbf{B} \mid (k - 1)$. This guarantees us that we have considered every possible combination of operators firing at precisely the same time. The result of this construction is then a set of possible trace tuples truncated at length k. This construction assumes that the truncated $\mathbf{B}$ of size $k - 1$ is known. Precise definitions for the functions $\oplus$, Δ, ρ and $fill$ can be found in Appendix A.

In our work, we assume that execution of the prototype is "fair", in the sense that, any operator which terminates in isolation will terminate when executed as part of a prototype. Failure of an operator to terminate is represented by a possibility function that gives the same set of possible output sequences for all possible extensions of an input sequence that fails to terminate.

B. SLICING OF PSDL PROTOTYPES

As we saw in Chapter III, Section C.2, a portion of a program's behavior can be captured by a slice of the program with respect to a single point in the program. We have developed a similar method that is also valid for isolating a portion of the behavior of a prototype. This section describes our method for taking slices of PSDL prototypes. One

67

of the differences between slicing for PSDL prototypes and slicing for while programs is that PSDL programs are inherently concurrent and non-deterministic. While programs represent individual deterministic sequential processes. This represents a major contribution of this work.

1. Prototype Dependence Graphs

Since PSDL implementations are graphs, we do not need a deep transformation to translate our prototypes into graphs as is the case for while programs. The only information we need to add to the current PSDL implementation graph are dependencies resulting from timer interactions, and an external vertex. The external vertex is added to allow slices of prototypes with vertices that have no outputs to include those vertices. The following defines our Prototype Dependence Graph (PDG):

Definition 4 <u>PSDL Prototype Dependence Graph:</u>

A Prototype Dependence Graph (PDG) for a prototype P is a fully expanded [1] *PSDL implementation graph* G_P. *In the* **PDG**, $G_P = (V, E, C)$, *the set of vertices has been augmented with an external vertex,* **EXT**, *and the set of edges,* E, *has been augmented with a timer dependency edge from* o_i *to* o_j, *for each pair of vertices* o_i, $o_j \in V$ *such that the control constraints of* o_i *contain timer operations which affect the state of a timer read by the control constraints of* o_j.

Values on a timer dependency edge can be modeled precisely in the same way as values on a data stream. A data_tuple on a timer dependency edge can be viewed as a tuple containing the following for each of the tuple components:

v: A pair (c, t) containing the operation $c \in (\textbf{Start}, \textbf{Stop}, \textbf{Reset})$ that last changed the state of the timer and the value t of the timer at the time of the state change.
op: The id of the operator which last changed the state of the timer.
tw: The time of the last state change.
tr: The time that op read its input streams before the firing that produced the state change in v.

---

[1] A fully expanded PSDL implementation graph is one in which every vertex represents an atomic operator.

68

For example, consider the data tuple $[(\mathbf{start}, 25), p, 36, 34]$ on the timer stream $Timer1$. In this example, the v element of the data tuple is the pair $(\mathbf{start}, 25)$, the op element is p, the write_time is 36, and the read_time is 34. This means that the operator p read its streams at time 34, started the timer $Timer1$ at time 36, and the current value of the timer when the state change was executed was 25. This view of timer dependency edges allows us to treat them the same as any other edge in the graph.

Top level prototypes do not contain inputs or outputs, so there will always be vertices which do not write to a stream. Since we construct our slices from sets of streams and not from vertices, as in slicing of while programs, these vertices could never be included in a slice. The external vertex is added to provide a way to capture these vertices during slicing. During construction of the **PDG** for a prototype, an artificial edge is added to the graph from any vertex which does not write to an output stream to the external vertex EXT. These edges are then considered in the construction of the slices of the prototype, thus allowing those terminal vertices an opportunity to be included. Only one external vertex is needed for this graph, because each artificial edge added is given a unique name, and considered separately in the construction of the slice.

2. Slicing Theorem

A *slice* of a PSDL prototype is defined in terms of the prototype's dependence graph. It contains the portion of the prototype which affects the history of a set of streams. This is useful in isolating changes made to a base version of a prototype in a modification. If the slices of two versions with respect to the same set of streams are different, then there are significant changes that have been made to one version and not the other.

Informally, a slice is an upstream closure of a set of edges in the graph that includes all the source nodes for the edges in the slice. A formal definition of a slice follows:

Definition 5 <u>Slice of a PSDL Prototype</u>:

A slice $S_P(X)$ of a PSDL prototype P with respect to a set of data streams $\vec{X}$ is the subgraph (V, E, C) of the PDG $\mathbf{G}_P$ where:

(1) V is the smallest set that contains all vertices $o_i \in \mathbf{G}_P$ that satisfy at least one of the following conditions:

a) o_i writes to one of the data streams in X.

b) o_i precedes o_j in $\mathbf{G}_P$, and $o_j \in V$.

(2) E is the smallest set that contains all of the edges $x_k \in \mathbf{G}_P$ which satisfy at least one of the following conditions:

a) $x_k \in X$.

b) x_k is directed to some $o_i \in V$.

(3) C is the smallest set that contains all of the timing and control constraints associated with each operator in V and each data stream in E.

Example 7 *Figure 4.4 shows a prototype for a fish farm control system called Fishies. Figures 4.5, 4.6 and 4.7 display different slices of Fishies.*

Theorem 3 <u>Slicing Theorem for PSDL Prototypes</u>:

Let $S_P(X)$ be the slice of a prototype P with respect to a set of streams X. Then $S_P(X)$ and P have the same behavior on any subset of the streams in $S_P(X)$.

The proof of this theorem is contained in Appendix C, Section 2. The significance of this theorem is that a slice captures a fragment of the semantic behavior of a prototype, and the behavior captured by that slice remains the same even if that slice is made a part of a different prototype, provided that it is also a slice with respect to that new prototype. This property is the basis for constructing a change merging operation that can provide semantic guarantees of correctness.
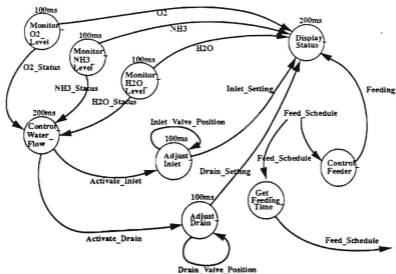
Figure 4.4: Fish Farm Control System, $Fishies_{1.1}$
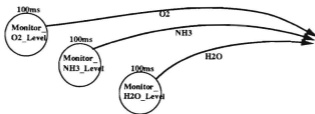
Figure 4.5: $S_{Fishies_{1.1}}(O2, NH3, H2O)$
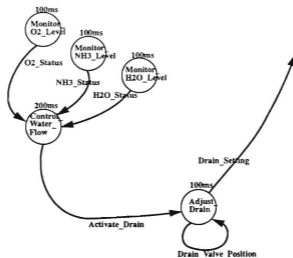
Figure 4.6: $S_{Fishies_{1.1}}(Drain\_Setting)$
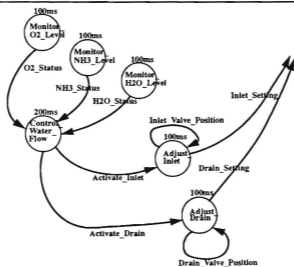
Figure 4.7: $S_{Fishies_{1.1}}(Drain\_Setting, Inlet\_Setting)$

C. A SLICING METHOD FOR CHANGE-MERGING PSDL PROTOTYPES

Our change-merging method for PSDL prototypes, illustrated in Figures 4.12 through 4.15 on the prototype versions originally introduced in Chapter III and shown again in Figures 4.4, 4.8 and 4.9 uses prototype slicing to determine automatically which parts of the prototype have been affected by a change and which parts have been preserved.
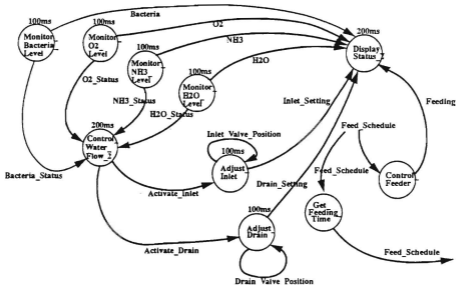
Figure 4.8: $Fishies_{1.2}$

If the slice of a changed version of a prototype with respect to a stream present in both the base version and the modified version is different than the same slice of the base version, then the behavior on that slice is likely to be different. Therefore that change is significant, and must be preserved in the merged version. For example, consider the slice of $Fishies_{1.1}$ with respect to the stream $Activate\_Drain$, illustrated in Figure 4.10, and the

73
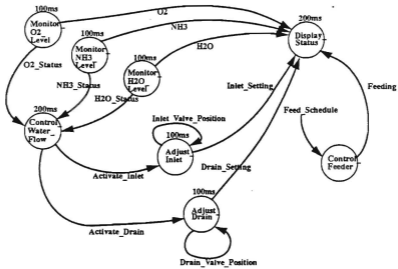
Figure 4.9: $Fishies_{2.2}$

same slice of $Fishies_{1.2}$, illustrated in Figure 4.11. It is easy to see a portion of the effect of the change which produced $Fishies_{1.2}$ from $Fishies_{1.1}$. If we were to take the same slice of $Fishies_{2.2}$, we would discover that it is identical to the slice of the base version of $Fishies$. This illustrates that this part of the $Fishies$ prototype is not affected by the change which produced $Fishies_{2.2}$. Since this change is significant, it must be reflected in the merged version.
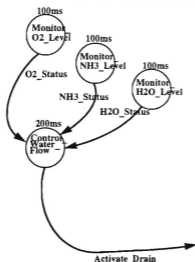
Figure 4.10: $S_{Fishies1.1}(Activate\_Drain)$

Slices are important because they capture all of the parts of a program that can affect the behavior visible in a set of data streams. If two different programs have the same slice for a set of streams, they also have the same behavior over that set of streams. The preserved part of a prototype is then the largest set of streams that have the same single stream slice in all three versions, and the affected streams of each modification are those that have a different single stream slice in the modified version than in the base version. Performing a change-merge using $Fishies_{1.1}$ as the base version, and $Fishies_{1.2}$ and $Fishies_{2.2}$ as the modified versions, we get the preserved part as shown in Figure 4.12 and affected parts as shown in Figures 4.13 and 4.14.
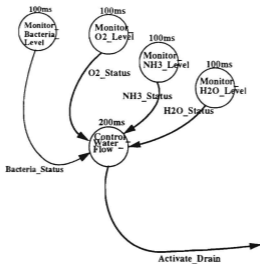
Figure 4.11: $S_{Fishies_{1,2}}(Activate\_Drain)$
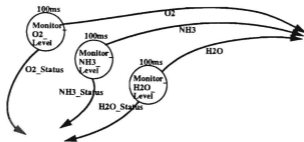
Figure 4.12: Preserved Parts of $Fishies_{1.1}$ in Both Modifications
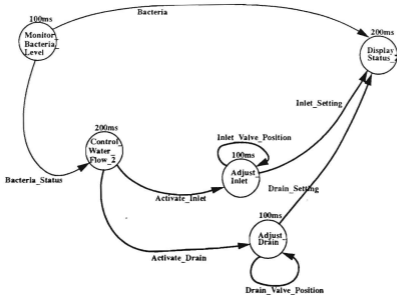
Figure 4.13: Affected Part of $Fishies_{1.2}$
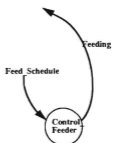
Figure 4.14: Affected Part of $Fishies_{2.2}$

In constructing the preserved part, we consider each stream individually, taking the slice of each version with respect to that stream. If the slices are the same, then that slice is added to the preserved part. After all streams have been checked, the preserved part is complete.

The affected parts are constructed by comparing the slices of each stream in the modified version against the same slice of the base version. The stream is included in the affected part if the slices are different.

The merged version is formed by taking the union of the preserved part of all three versions and the affected parts of the two modified versions. If the slice of the merged version with respect to the streams affected by each modification is the same as the corresponding slice of the modified version, then semantic correctness of the merged version with respect to the modifications is established. The result of change-merging $Fishies_{1.1}$, $Fishies_{1.2}$ and $Fishies_{2.2}$ is shown in Figure 4.15.

Our slicing method has the advantage of a clear semantic criterion for correctness, and the disadvantage of reporting conflicts whenever two changes can affect the same stream, regardless of whether there exists a computation history in which the two changes actually interact or conflict with each other.
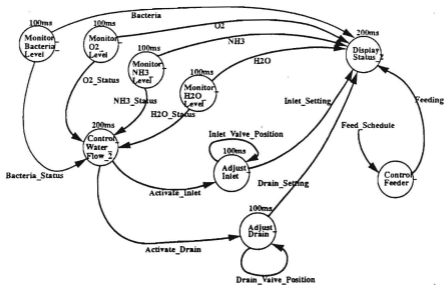
Figure 4.15: The Change-Merged Version of the Fishies Prototype.

V. CHANGE-MERGE ALGORITHM

From the change-merging models for both the specification, shown in Chapter III, and the implementation, shown in Chapter IV, we developed a change-merging algorithm. This change-merging algorithm takes advantage of the fact that the specification and implementation can be change-merged separately to create a correctly change-merged program. This chapter outlines the change-merging algorithm in detail and provides a piece by piece analysis of the algorithm for correctness, complexity and coverage. This algorithm was written to accept a base version and two modifications as input. It is easily extended to change-merge the result of n modifications to a base version by applying the algorithm iteratively using the result of the most recent application as one input and the next modification as the other. The result of a successful iterative application on n versions is a merged version containing the significant behaviors of each of the inputs.

The algorithm *change_merge* accepts three expanded versions of a PSDL program as input. It then extracts all of the PSDL components from each version of the program. The atomic components are held in storage to be included in the change-merged version of the program if needed, and the composite component of each program is divided into a specification part and an implementation part.

Each of these parts are change-merged separately and the results are recombined to create the change-merged composite component. From the implementation part of the change-merged composite component, the algorithm can deduce which of the atomic components need to be included in the change-merged program. The change-merged program is then returned. If a conflict is detected during the change-merging process, the *CONFLICT* variable is set to true, and a flag is placed into the change-merged program at the location of the conflict to aid the designer in locating and resolving it. Figure 5.1 shows the change-merge algorithm.

Algorithm change_merge($BASE, A, B$: in psdl_program; $CONFLICT$: out boolean)
    return psdl_program
begin
    1. Extract the psdl_components from each of the input psdl_programs.
    2. Change-merge the specification parts for the three input composite components.
        a. Change-merge the state declarations.
        b. Change-merge the exception declarations.
        c. Change-merge the maximum execution times.
        d. Change-merge the formal and informal descriptions.
    3. Change-merge the implementation parts for the three input composite components.
        a. Create the prototype dependency graphs for each version.
        b. Create the affected parts of each modified version.
        c. Create the preserved part of the base in all three versions.
        d. Change-merge the graphs.
        e. Change-merge the stream declarations.
        f. Change-merge the timer declarations.
        g. Change-merge the control constraints.
            (1) Change-merge the trigger constraints.
            (2) Change-merge the execution guard constraints.
            (3) Change-merge the periods.
            (4) Change-merge the finish_withins.
            (5) Change-merge the minimum calling periods.
            (6) Change-merge the maximum response times.
            (7) Change-merge the output guard constraints.
            (8) Change-merge the exception trigger constraints.
            (9) Change-merge the timer operations.
    4. Create the change-merged program.
        a. Combine the change-merged specification and implementation.
        b. From the resulting implementation, determine which of the atomic components
           from each of the input versions is to be included in the change-merged program.
    5. Return the change-merged program.
end Change_Merge;

Figure 5.1: Algorithm *change_merge*.

A. EXTRACTING THE COMPONENTS

Extracting the components from each of the input PSDL programs is done using a map *fetch* operation. The algorithm loops through each of the input programs and retrieves the set of components each one contains. The atomic components are placed in a holding program so they can be retrieved later if needed for the merged program, and the composite component is extracted for change-merging. The algorithm fragment used to extract the components is shown in Figure 5.2.

---

a. For every component in the *Base* Version loop
    (1) Fetch the component;
    (2) If the component is atomic then bind to holding program for base version;
    (3)   else extract the component; end if; end loop;
b. For every component in the *A* Version loop
    (1) Fetch the component;
    (2) If the component is atomic then bind to holding program for base version;
    (3)   else extract the component; end if; end loop;
c. For every component in the *B* Version loop
    (1) Fetch the component;
    (2) If the component is atomic then bind to holding program for base version;
    (3)   else extract the component; end if; end loop;

Figure 5.2: Algorithm Fragment for Extracting the Component.

The extraction part of the algorithm requires a loop through the components of each version to perform the fetch. The correctness of this algorithm fragment can be shown using simple induction. Since the operations inside the loop are constant and the loop is executed only once for each component of each program, the worst-case complexity of this part of the algorithm is $\mathcal{O}(n)$, where n is the number of components in the program. This algorithm fragment can be used for all fully expanded PSDL programs, since they contain only one composite component.

B. CHANGE-MERGING THE SPECIFICATIONS

Change-merging the specification of the top level component requires five operations. The five operations are responsible for change-merging the components of the specification: the state declarations, the maximum execution times, the exception sets, and the informal and formal descriptions.

1. Change-Merging the State Declarations

Change-merging the state declarations is done with the procedure *merge_states*. Since the state declarations are a set, normal set operations may be used to merge the state declarations themselves, but the initial values of the state variables conform to a flat lattice structure and any change must be preserved. The algorithm *merge_states* is shown in Figure 5.3.

To show correctness of *merge_states*, we must show that it correctly implements the equation $(A - Base) \cup (A \cap B) \cup (B - Base)$. Two internal loops construct this equation. The first loop captures any state variable declaration which appears in A, but not in $BASE$; the $(A - Base)$ part of the equation, and then captures any state variable declaration which appears in both of the modified versions; the $(A \cap B)$ part of the equation. The second loop captures any state variable declaration which appears in B, but not in $BASE$; the $(B - Base)$ part. Since both loops add state variable declarations to the same set $MERGE$, the union part of the equation is satisfied.

The execution of *merge_states* requires a membership test and add operation for every state declared, and these are both linear time operations with the current linked-list implementation of sets. Thus the entire algorithm requires $\mathcal{O}(s^2)$ time, where s is the number of states declared. This can be improved to $\mathcal{O}(s\log s)$ if balanced trees are used for the sets.

```
Algorithm merge_states(MERGE: in out type_declaration;
                       BASE, A, B: in type_declaration;
                       MERGE_INIT: in out init_map;
                       A_INIT, B_INIT: in init_map)
begin
    for every state variable, s, declared in A
        if s is not in BASE, and s is not in B then
            add s to MERGE; add initial value to MERGE_INIT;
        end if;
        if s is in B then
            add to MERGE;
            if the initial values are the same in A_INIT and B_INIT then
                add initial value to MERGE_INIT;
            else add conflict_expression to MERGE_INIT;
            end if;
        end if;
    end loop;
    for every state variable, s, declared in B
        if s is not in BASE and s is not in A then
            add to MERGE; add initial value to MERGE_INIT;
        end if;
    end loop;
end merge_states;
```

Figure 5.3: Algorithm merge_states.

2. Change-Merging the Maximum Execution Times

Change-merging the maximum execution time constraints is done with the function *merge_met*, shown in Figure 5.4. Maximum execution times follow a Brouwerian Algebra structure as shown by Proposition 2 in Chapter III, Section D.2, and must be merged according to those rules.

```
Algorithm merge_met(BASE, A, B : millisec) return millisec
 A_DIFF_BASE, B_DIFF_BASE, A_INT_B: millisec;
 begin
    if A ≤ B then
       A_INT_B := B;
       else A_INT_B := A;
    end if;
    if BASE ≤ A then
       A_DIFF_BASE := ⊥;
       else A_DIFF_BASE := A;
    end if;
    if BASE ≤ B then
       B_DIFF_BASE := ⊥;
       else B_DIFF_BASE := B;
    end if;
    if A_DIFF_BASE ≤ A_INT_B then
       if A_DIFF_BASE ≤ B_DIFF_BASE then
          return A_DIFF_BASE;
          else return B_DIFF_BASE;
       end if;
       else if A_INT_B ≤ B_DIFF_BASE then
          return A_INT_B;
          else return B_DIFF_BASE;
       end if;
    end if;
 end merge_met;
```

Figure 5.4: Algorithm *merge_met*.

The algorithm for change-merging maximum execution times must also satisfy the change-merging equation $(A - Base) \sqcup (A \sqcap B) \sqcup (B - Base)$. It uses a series of conditional expressions to calculate the values of A_DIFF_BASE, B_DIFF_BASE, and A_INT_B,

which represent the $(A - Base)$, $(B - Base)$ and $A \sqcap B$ parts of the equation shown above. It then combines them according to the rules outlined in Chapter III, Section D.2. This adherence to the mathematical model guarantees the correctness of the algorithm. Since this algorithm contains no loops, it requires constant time to execute, so the worst-case time complexity of $merge\_met$ is $\mathcal{O}(1)$.

3. Change-Merging the Exception Declarations and Keywords

Change-merging the exception declarations and the keyword sets is done using the $merge\_id\_sets$ function shown in Figure 5.5. This algorithm calculates the equation $(A - Base) \cup (A \cap B) \cup (B - Base)$ in precisely the same way as $merge\_states$ calculates the merge of the state declarations without the initial values.

Algorithm $merge\_id\_sets(BASE, A, B : id\_set)$ return id_set
  begin
    Calculate $A - BASE$.
    Calculate $B - BASE$.
    Calculate $A \cap B$.
    Return $(A - BASE) \cup (A \cap B) \cup (B - BASE)$.
  end $merge\_id\_sets$;

Figure 5.5: Algorithm $merge\_id\_sets$.

The correctness and complexity analyses of $merge\_id\_sets$ are identical to those of $merge\_states$, so merging id_sets requires worst case $\mathcal{O}(x^2)$ time for exception declarations and $\mathcal{O}(k^2)$ time for keywords.

4. Change-Merging the Descriptions

Change-merging both the informal and the formal descriptions is accomplished using the function $merge\_text$ shown in Figure 5.6. $merge\_text$ implements a flat lattice change-merge, and any change from the base version in one modification must be identical to

any change in the other modification or a conflict is produced. This function has a constant time complexity.

```
Algorithm merge_text(BASE, A, B : text) return text
  begin
    if BASE = A
      then return B
      else if BASE = B
        then return A
        else if A = B
          then return A
          else return ("Conflict in text. Must be change-merged manually!")
        end if;
      end if;
    end if;
  end merge_text;
```

Figure 5.6: Algorithm merge_text.

5. Analysis of Specification Change-Merge

Correctness of the specification change-merge part of the algorithm is guaranteed by the correctness of the individual algorithms which make up the specification change-merge. The worst-case time complexity of the specification part of the algorithm is obtained by adding the complexities of the individual parts as follows:

$$\mathcal{O}(s^2) + \mathcal{O}(x^2) + \mathcal{O}(k^2) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(s^2 + x^2 + k^2)$$

where s is the number of state declarations, x is the number of exception declarations, and k is the number of keywords.

This algorithm is capable of performing change-merge operations on all PSDL operator specifications.

C. CHANGE-MERGING THE IMPLEMENTATIONS

Change-merging the implementation parts is also accomplished by change-merging the individual parts of the implementation separately. It requires five main operations; change-merging the graphs, change-merging the stream declarations, change-merging the timer declarations, change-merging the control constraints, and change-merging the informal descriptions.

1. Change-Merging the Graphs

To change-merge the PSDL implementation graphs, we must first convert them to prototype dependency graphs that accurately reflect all of the timer dependencies between operators in the prototype as well as the data dependencies. We do this with the *build_PDG* function shown in Figure 5.7. Next we must construct the preserved and affected parts of the three input graphs according to the slicing rules defined in Chapter IV. The algorithms for these constructions are contained in Figures 5.9 and 5.8, respectively. Finally, we must combine these three parts into a change-merged prototype dependency graph using a graph-union operation, shown in Figure 5.12.

In building the prototype dependency graph, *build_PDG* adds an external vertex, EXT, to the prototype implementation graph, then for every vertex with no outputs, it creates an edge from that vertex to EXT. This is necessary to ensure that these terminal vertices are included in the slices, since slices are constructed based on edges not vertices. Then for every timer declaration in the prototype implementation, *build_PDG* creates an edge from every vertex which affects the state of that timer to every vertex which reads its value.

The algorithm *build_PDG* contains two loops. The first loop iterates through the vertices in the graph, and determines if the vertex has any outputs. For every vertex with no outputs, the algorithm then adds an edge to the graph from that vertex to the artificial

```
Algorithm build_PDG(P : psdl_component) return prototype_dependency_graph
  G : prototype_dependency_graph;
  O : vertex;
  source, dest : id_set;
begin
  G := graph(P);
  add external vertex, EXT;
  for every terminal vertex, O, add an edge from O to EXT;
  for every timer declaration in the implementation of P loop
    initialize source and dest to empty.
    add every vertex which affects the state of the timer to source;
    add every vertex which reads the timer to dest;
    add an edge to G from every vertex in source to every vertex in dest;
  end loop;
  return G;
end build_PDG;
```

Figure 5.7: Algorithm build_PDG.

vertex EXT. The correctness of this loop can be established by showing that at the end of

the loop, there are no vertices in the graph without output edges, except EXT. Since the

loop cycles through all vertices in the graph and adds an output edge to the graph from any

vertex which does not have one to EXT, this proof is trivial.

The second loop iterates through the set of timer declarations, and builds two

sets for each timer, $source$ and $dest$. It then adds an edge to the graph from every vertex

in $source$ to every edge in $dest$. The $source$ set contains all of the vertices using timer

operations that affect the state of the timer. The $dest$ set contains all of the vertices that

read the value of the timer.

To show correctness of this loop, we must show that at the end of each iteration

through the loop, the graph contains all timer dependency edges associated with the timer

declarations thus far encountered, and at the end of the loop, the graph contains all timer

dependency edges associated with the timer declarations in the implementation. We do this

by the following induction proof:

89

Proof:

Basis: Since *source* and *dest* are initialized at the beginning of each iteration through the loop, they are empty before the first iteration, thus the graph contains no timer dependency edges before the first iteration.

Induction Hypothesis: At the end of the kth iteration, all timer dependency edges associated with the first k timer declarations are included in the graph.

Induction Step: At the beginning of the $k + 1$st iteration of the loop, the *source* and *dest* sets are reinitialized to empty. The vertices that affect the state of the $k + 1$st timer are added to *source*, and the vertices that read the value of the $k + 1$st timer are added to *dest*. Now, for every vertex in *source*, the algorithm adds an edge to every vertex in *dest*. Thus at the end of the $k + 1$st iteration, the graph contains all timer dependency edges associated with the first k timer declarations, by the induction hypothesis, plus it now contains all timer dependency edges associated with the $k + 1$st timer declaration. Thus, we can conclude that for any number n of timer declarations, at the end of the nth iteration of the loop, the graph contains all timer dependency edges associated with the first n timer declarations. □

The complexity of this algorithm is determined by the sum of the complexities of the two loops. Since the first loop iterates through all vertices in the graph, performing worst-case linear operations on each iteration, its worst case time complexity is $\mathcal{O}(n^2)$, where n is the number of vertices in the graph, excluding *EXT*. The second loop contains three inner loops that iterate through the vertex set of the graph. The first two of these inner loops contain worst-case linear operations. The third inner loop contains another inner loop that could also possibly iterate through all vertices in the graph, making its worst case time complexity $\mathcal{O}(n^2)$. Thus, the worst case time complexity of the *second* outer loop is $\mathcal{O}(tn^2)$, where t is the number of timer declarations contained in the implementation and n is the number of vertices in the graph. The algorithm then contains two loops, one with

90

complexity $\mathcal{O}(n^2)$, and one with complexity $\mathcal{O}(tn^2)$, therefore, the worst case time complexity of $build\_PDG$ is $\mathcal{O}(tn^2)$.

The next step in change-merging the graphs is finding out the parts of the modified versions which are different from the base. This is accomplished using the algorithm $affected\_part$, shown in Figure 5.8. This algorithm returns the set of edges in the modified version for which the slice of the modified version is different than the slice of the base version. First, each edge in the modified version is checked to see if it is the base version. If it is not, then it is added to the affected part. Next, the algorithm checks to see if the edge recieves input from different sources in the modified version than in the base version. If the sources are different, then the edge is added to the affected part. Finally, the algorithm adds any edge to the affected part which receives input via an edge already in the affected part.

It is sufficient to include in the affected part of modified version, only those edges which are different in the modified and base versions of the graph, and the edges which follow them. Any edge which precedes an affected edge will produce the same slice in both versions since slices are constructed backward from the edge. The correctness of $affected\_part$ is established by showing that, every in edge in $Slice$ produces a slice which is different in both G and B. We prove this by an induction over the while loop.

Proof:

Basis: At the beginning of the first iteration of the loop, $Slice$ gets one edge from E which is either in G and not in B, or is written to by a different set of vertices in G and B. This edge will certainly produce a different slice in the two graphs, so $Slice$ contains only edges which produce different slices in G and B.

Induction Hypothesis: After the first k iterations of the loop, every edge in $Slice$ produces a different slice in G than in B.

91

```
Algorithm affected_part(G, B : prototype_dependency_graph)
 return edge_set
 Slice, C, D, E : edge_set;
 x, y : edge;
 begin
    C := edges(B);
    D := edges(G);
    E := difference(D, C);
    for every edge x in D loop
       if sources(x) in G are different from the
          sources(x) in B then
          add x to E;
       end if;
    end loop;
    while E not empty loop
       select and remove an edge x from E;
       add x to Slice;
       for each edge y ∈ D loop
          if x.destination ∈ sources(y, G) then
             add y to E;
             remove y from D;
          end if;
       end loop;
    end loop;
    return Slice;
 end affected_part;
```

Figure 5.8: Algorithm affected_part

Induction Step: During the kth iteration of the loop, every edge in G which follows the kth edge added to *Slice* in the data flow of G is added to E. At the beginning of the $k + 1$st iteration of the loop, one of the edges in E is removed from E and added to *Slice*. Since we know that any edge which follows an affected edge in the data flow will certainly produce a different slice in G and B, we know that this edge will as well. Thus by the induction hypothesis, all of the elements in *Slice* before this iteration produced different slices in G and B, and the current iteration adds an edge which produces a different slice in G and B, therefore after the $k + 1$st iteration of the loop, every edge in *Slice* produces a different slice in G and B. Since E is a finite set, and no edge already in *Slice* can be added back into E, the loop will terminate. □

The complexity of *affected_part* is determined by the complexity of the loops inside. The first *for* loop iterates over all of the edges in the input graph, G, and adds any edge to E which is different in G and B or recieves input from different sources in G and B. The worst-case time complexity of this loop is $\mathcal{O}(e * n)$, where e is the number of edges in G and n is the number of vertices in G. The second **while** loop iterates over all of the edges in E, which we know to contain at most the edges of G, and any edge which follows this edge in G is added to E. This makes the worst-case time complexity of this loop, $\mathcal{O}(e^2)$. Therefore the worst-case time complexity of *affected_part* is $\mathcal{O}(e^2)$, where e is the number of edges in G.

The next step in change-merging the graphs is constructing the part of the base version that is preserved in both of the modifications. This is done using the algorithm *preserved_part* shown in Figure 5.9. This algorithm loops over all of the edges in *Base* and checks to see if they are in the affected parts of either modification, or have been removed in one of the modifications. If the edge is not in either affected part and it is in both modifications, then the slice it produces is the same in all three versions and it is added to the preserved part.

```
Algorithm preserved_part(BASE, A, APA, B, APB : prototype_dependency_graph)
  return edge_set
PP : edge_set := empty_set;
e : edge;
begin
    for every E in edges(BASE) loop
        if not e ∈ APA ∪ APB and e ∈ edges(A) ∩ edges(B)
            then add e to PP;
        endif;
    end loop
    return PP;
end preserved_part;
```

Figure 5.9: Algorithm preserved_part

Only those edges which appear in all three versions and are not part of either affected part are added to the preserved part. The correctness of this algorithm is established by showing that after each iteration of the **for** loop, PP only contains edges which will produce the same slice in all three versions. We offer the following proof:

Proof:

Basis: Before the first iteration of the loop, PP is empty. Since the slice with respect an empty edge is an empty graph, this slice is certainly the same in all three versions.

Induction Hypothesis: After the first k iterations of the loop, every edge in PP produces the same slice in all three versions of the graph.

Induction Step: During the $k + 1$st iteration of the loop, if the edge e is in the edge sets of all three versions, and it is not contained in an affected part, then it was not affected nor removed by either version, thus it is preserved in all three versions. Since after the kth iteration of the loop, all of the edges in PP produced slices which were the same in all three versions, and the $k + 1$st iteration adds another which produces the same slice in all three versions, after $k + 1$ iterations, every edge in PP produces the same slice in all three versions. Therefore the correctness of preserved_part is established. □

94

The complexity of *preserved_part* is determined by the complexity of the **for** loop. Since all of the operations inside of the loop are at worst $\mathcal{O}(e)$ operations, and the loop iterates once for every edge in the base version of the graph, the worst-cast time complexity of *preserved_part* is $\mathcal{O}(e^2)$, where e is the number of edges in $BASE$.

Once the preserved part of the base and the affected parts of both modified versions have been calculated, the slices produced by these sets can be change-merged into a single graph. The slices are constructed using the algorithm *create_slice* shown in Figure 5.10. This algorithm takes a graph and an edge as input and constructs the slice backward from the edge, according to the definition for a slice given in Chapter IV, Section B.2.

```
Algorithm create_slice(G : prototype_dependency_graph; E : edge)
  return prototype_dependency_graph
  S : prototype_dependency_graph;
  V : vertex_set;
  w : vertex;
  begin
    if e in G
        then add e to S;
        else return empty_graph;
    end if;
    for every vertex w in G loop
        if w writes to e
            then add w to V;
        end if;
    end loop;
    while V not empty loop;
        select and remove vertex w from V;
        add w to S;
        add parents of w to V if not in S;
        add edges between parents and w to S;
    end loop;
    return S;
  end create_slice;
```

Figure 5.10: Algorithm *create_slice*

The correctness of *create_slice* is established by showing that the algorithm produces a correct slice of G with respect to E, according to our definition of a slice given in Chapter IV.

Proof:

If e is not an edge in G, then *create_slice* returns an empty graph, which is the correct slice of G with respect to e. If e is an edge in G, then e is added to the slice, and any vertex which writes to e is added to the set of vertices V. Then the algorithm iterates over a **while** loop as long as V is not empty. The correctness of the **while** loop is established by showing that at the end of every iteration of the loop, the slice S contains only the vertices and edges which affect the edge e, and after the last iteration of the loop, S contains all of the vertices and edges in G which affect the edge e.

Basis: Before the first iteration of the loop, the only edge in S is e, and certainly every edge in S affects e.

Induction Hypothesis: After the kth iteration of the loop, all of the edges and vertices in S affect the values written to e.

Induction Step: During the $k + 1$st iteration of the loop, a vertex w is removed from V and added to S. Since only those vertices which write to an edge in S are in V, and only edges which affect the values written to e are in S by the Induction Hypothesis, we are guaranteed that w is a correct addition to S. So after the $k + 1$st iteration of the loop, S contains only edges and vertices which affect values written to e.

Since V contains at most the number of vertices in G, and one vertex is removed on every iteration of the loop, we are assured that the loop will terminate. Since during every iteration, any edge which provided input to a vertex in S is added to S and every iteration adds any vertex which writes to an edge in S, we are assured that after the last iteration of the loop, all of the vertices and edges which affect the values written to e will be in the slice. $\square$

The time complexity of *create_slice* is determined by the two inner loops. The for loop iterates over the vertices of the graph one time. This makes the worst-case time complexity of this loop, $\mathcal{O}(n)$, where n is the number of vertices in the graph. The while loop iterates over a set of vertices, however through all of the iterations of the loop at most each edge is visited once, making the worst-case time complexity of this loop $\mathcal{O}(e)$, where e is the number of edges in the base version of the graph. Therefore, the worst-case time complexity of *create_slice* is $\mathcal{O}(n + e)$.

Once the slices are constructed, they are merged using the function *graph_merge*, shown in Figure 5.11. This is a very simple graph merging algorithm which uses successive calls to *graph_union*, shown in Figure 5.12 to combine the preserved part of the base with the affected parts of both modifications into a change-merged prototype dependency graph.

Algorithm *graph_merge*(G1, G2, G3 : *prototype_dependency_graph*)
  return *prototype_dependency_graph*
  G : *prototype_dependency_graph* := *empty_psdl_graph*;
  begin
    G := *graph_union*(G1, G2);
    G := *graph_union*(G, G3);
    return G;
  end *graph_merge*;

Figure 5.11: Algorithm *graph_merge*

The *graph_union* algorithm is used to combine two graphs into one. It accepts two prototype dependency graphs as input and adds the edges and vertices of one to the other.

The algorithm *graph_union* makes use of two successive union operations, and union operations are very well defined. Therefore, the correctness of *graph_union* is easily established.

The complexity of *graph_union* is determined solely by the complexities of the set union operations, which are linear in the worst case. Therefore, the worst case time

```
Algorithm graph_union(G1, G2 : prototype_dependency_graph)
 return prototype_dependency_graph
 G : prototype_dependency_graph := empty_psdl_graph;
 begin
    G.vertices := vertices(G1) ∪ vertices(G2);
    G.edges := edges(G1) ∪ edges(G2);
    return G;
 end graph_union;
```

Figure 5.12: Algorithm *graph_union*

complexity of *graph_union* is the sum of the complexities of the two union operations, or $\mathcal{O}(e + n)$, where e is the number of edges in the largest of $G1$ and $G2$, and n is the number of vertices in the largest of $G1$ and $G2$.

The correctness and complexity of *graph_merge* depend solely on the correctness and complexity of *graph_union*, which have previously been established. Thus, *graph_merge* is a correct algorithm with worst-case time complexity of $\mathcal{O}(e + n)$, where e and n are the number of edges and vertices in the largest input graph.

Once the graphs have been change-merged, the remainder of the implementation parts must be change-merged. The stream declarations and the timer declarations are change-merged using the functions *merge_streams* and *merge_timers*, shown in Figures 5.13 and 5.14, respectively. Then the control constraints are change-merged using functions appropriate to their map type. These algorithms are shown in Figures 5.15 through 5.26.

2. Change-Merging the Stream and Timer Declarations

The stream and timer declaration parts are modeled as sets, so change-merging them is done using common set operations. In *merge_streams*, the two *for* loops construct the three pieces of the change-merging equation for sets, $(A - Base)$, $(A \cap B)$, and $(B - Base)$. The correctness and complexity of *merge_streams* are identical to those of *merge_states*. In *merge_timers*, the three pieces of the change-merging equation are constructed separately

and combined to provide the result. The correctness and complexity of this algorithm is identical to that of *merge_id_sets*.

Algorithm *merge_streams*(*BASE*, *A*, *B* : *type_declaration*) return *type_declaration*
  MERGE : *type_declaration*;
 begin
   MERGE := *empty_type_declaration*;
   for every stream *s* in *A* loop
     if *s* is not in *BASE* and *s* is not in *B* then
       add *s* to *MERGE*;
     end if;
     if *s* is in *B* then
       add to *MERGE*;
     end if;
   end loop;
   for every stream *s* in *B* loop
     if *s* is not in *BASE* and *s* is not in *A* then
       add to *MERGE*;
     end if;
   end loop;
   return *MERGE*;
 end *merge_streams*;

Figure 5.13: Algorithm *merge_streams*

Algorithm *merge_timers*(*BASE*, *A*, *B* : *id_set*) return *id_set*
 begin
   Calculate *A* − *BASE*.
   Calculate *B* − *BASE*.
   Calculate *A*∩*B*.
   Return (*A* − *BASE*)∪(*A*∩*B*)∪(*B* − *BASE*).
 end *merge_timers*;

Figure 5.14: Algorithm *merge_timers*

3. Change-Merging the Control Constraints

Change-Merging the Control Constraints is accomplished by a series of algorithms that implement the models defined in Chapter III, Section D. Their correctness is established

by their conformance to the mathematical models. Each one of these algorithms has worst-case time complexity of $\mathcal{O}(n)$, except *merge_trigger_maps* and *merge_timer_op_maps*, where n is the number of vertices in the largest input prototype. *merge_trigger_maps* has worst-case time complexity of $\mathcal{O}(ns^2)$, where n is the number of vertices in the largest input prototype and s is the largest number of streams read by an operator in the prototype. *merge_timer_ops* has worst-case time complexity of $\mathcal{O}(nt^2)$, where n is the number of vertices in the largest input prototype and t is the largest number of timer operations in the prototype. Since these algorithms all execute independently, the worst-case time complexity for the entire control constraints section is $\mathcal{O}(ns^2 + nt^2)$.

```
Algorithm merge_trigger_maps(VERTS : id_set; BASE, A, B : trigger_map)
  return trigger_map
  MERGE : trigger_map;
  op_id : psdl_id;
  base_trig, a_trig, b_trig, merge_trig : trigger:
begin
  for every op_id in VERTS loop
    retrieve base_trig from BASE;
    retrieve a_trig from A;
    retrieve b_trig from B;
    merge_trig := merge_triggers(base_trig, a_trig, b_trig);
    bind merge_trig to op_id in MERGE;
  end loop;
  return MERGE;
end merge_trigger_maps;
```

Figure 5.15: Algorithm *merge_trigger_maps*

There is also an informal description of the implementation part that must be change-merged. The implementation descriptions are change-merged using the *merge_text* function shown in Figure 5.6.

```
Algorithm merge_triggers(BASE, A, B : trigger) return trigger
  MERGE : trigger;
  streams : id_set;
begin
    if BASE = A then
       if BASE = B then
          MERGE.tt := BASE.tt
          MERGE.streams := merge_id_sets(BASE.streams, A.streams, B.streams);
          return MERGE;
       else return B;
       end if;
    else if BASE = B then
       return A;
       else if A = B then
          return A;
          return conflict;
       end if;
    end if;
  end if;
end merge_triggers;
```

Figure 5.16: Algorithm merge_triggers

```
Algorithm merge_exec_guard_maps(VERTS : id_set; BASE, A, B : exec_guard_map)
  return exec_guard_map
  MERGE : exec_guard_map;
  op_id : psdl_id;
  base_eg, a_eg, b_eg, merge_eg : expression:
begin
    for every op_id in VERTS loop
       retrieve base_eg from BASE;
       retrieve a_eg from A;
       retrieve b_eg from B;
       merge_eg := merge_expressions(base_eg, a_eg, b_eg);
       bind merge_eg to op_id in MERGE;
    end loop;
    return MERGE;
end merge_exec_guard_maps;
```

Figure 5.17: Algorithm merge_exec_guard_maps

```
Algorithm merge_expressions(BASE, A, B : expression) return expression;
  begin
    if equal(BASE, A) then
      if equal(BASE, B) then return BASE else return B; end if;
    else if equal(BASE, B) then return A;
      else if equal(A, B) then return A;
        return conflict;
      end if;
    end if;
  end if;
end merge_expressions;
```

Figure 5.18: Algorithm merge_expressions

```
Algorithm merge_output_guard_maps(VERTS : id_set; BASE, A, B : out_guard_map) re-
turn out_guard_map
  MERGE : out_guard_map;
  op_id : psdl_id;
  base_og, a_og, b_og, merge_og : expression:
  begin
    for every op_id in VERTS loop
      retrieve base_og from BASE;
      retrieve a_og from A;
      retrieve b_og from B;
      merge_og := merge_expressions(base_og, a_og, b_og);
      bind merge_og to op_id in MERGE;
    end loop;
    return MERGE;
  end merge_output_guard_maps;
```

Figure 5.19: Algorithm merge_output_guard_maps

```
Algorithm merge_excep_trigger_maps(VERTS : id_set; BASE, A, B : excep_trigger_map)
 return excep_trigger_map;
 MERGE : excep_trigger_map;
 op_id : psdl_id;
 base_et, a_et, b_et, merge_et : expression:
 begin
     for every op_id in VERTS loop
         retrieve base_et from BASE;
         retrieve a_et from A;
         retrieve b_et from B;
         merge_et := merge_expressions(base_et, a_et, b_et);
         bind merge_et to op_id in MERGE;
     end loop;
     return MERGE;
 end merge_excep_trigger_maps;
```

Figure 5.20: Algorithm merge_excep_trigger_maps

```
Algorithm merge_timer_op_maps(VERTS : id_set; BASE, A, B : timer_op_map) return
timer_op_map;
 MERGE : excep_trigger_map;
 op_id : psdl_id;
 base_set, a_set, b_set, merge_set : expression:
 begin
     for every op_id in VERTS loop
         retrieve base_set from BASE;
         retrieve a_set from A;
         retrieve b_set from B;
         merge_set := merge_timer_op_set(base_set, a_set, b_set);
         bind merge_set to op_id in MERGE;
     end loop;
     return MERGE;
 end merge_timer_op_maps;
```

Figure 5.21: Algorithm merge_timer_op_maps

```
Algorithm merge_timer_op_sets(BASE, A, B : timer_op_set) return timer_op_set
  MERGE : timer_op_set;
  t_op : timer_op;
  begin
    for every t_op in BASE loop
      if member(t_op, A) then
        if member(t_op, B) then
          add(t_op, MERGE);
        end if;
      end if;
    for every t_op in A loop
      if notmember(t_op, MERGE) then
        if member(t_op, B) then
          add(t_op, MERGE);
        end if;
      end if;
    for every t_op in B loop
      if notmember(t_op, MERGE) then
        if member(t_op, A) then
          add(t_op, MERGE);
        end if;
      end if;
    end loop;
    return MERGE;
  end merge_timer_op_sets;
```

Figure 5.22: Algorithm *merge_timer_op_sets*

```
Algorithm merge_period(VERTS : id_set; BASE, A, B : timing_map) return timing_map
  MERGE : timing_map;
  op_id : psdl_id;
  base_val, a_val, b_val, merge_val : millisec := 0;
begin
    for every op_id in VERTS loop
        retrieve base_val from BASE;
        retrieve a_val from A;
        retrieve b_val from B;
        merge_val := merge_timing_data(base_val, a_val, b_val);
        bind merge_val to op_id in MERGE;
    end loop;
    return MERGE;
end merge_period;
```

Figure 5.23: Algorithm merge_period

```
Algorithm merge_fw_or_mrt(VERTS : id_set; BASE, A, B : timing_map)
  return timing_map
  MERGE : timing_map;
  op_id : psdl_id;
  base_val, a_val, b_val, merge_val : millisec := 0;
begin
    for every op_id in VERTS loop
        retrieve base_val from BASE;
        retrieve a_val from A;
        retrieve b_val from B;
        merge_val := merge_met(base_val, a_val, b_val);
        bind merge_val to op_id in MERGE;
    end loop;
    return MERGE;
end merge_fw_or_mrt;
```

Figure 5.24: Algorithm merge_fw_or_mrt

```
Algorithm merge_min_call_per(VERTS : id_set; BASE, A, B : timing_map)
 return timing_map
 MERGE : timing_map;
 op_id : psdl_id;
 base_val, a_val, b_val, merge_val : millisec := 0;
begin
   for every op_id in VERTS loop
      retrieve base_val from BASE;
      retrieve a_val from A;
      retrieve b_val from B;
      merge_val := merge_mcp(base_val, a_val, b_val);
      bind merge_val to op_id in MERGE;
   end loop;
   return MERGE;
end merge_min_call_per;
```

Figure 5.25: Algorithm merge_min_call_per

4. Analysis of Implementation Change-Merge

Change-merging the implementation of the top level component requires four main operations; change-merging the graphs, change-merging the stream declarations, change-merging the timer declarations, and change-merging the control constraints.

Change-merging the graphs requires that each graph be converted to a PDG using build_PDG which requires $\mathcal{O}(tn^2)$ time, where n is the number of vertices in the graph and t is the number of timers declared in the implementation.

After the prototype dependency graphs are constructed, the affected parts of each modification are constructed using affected_part which has worst-case time complexity $\mathcal{O}(e^2)$, where e is the number of edges. Then the preserved part is constructed, and it has worst-case time complexity $\mathcal{O}(e^2)$.

106

```
Algorithm merge_mcp(BASE, A, B : millisec) return millisec
  A_DIFF_BASE, B_DIFF_BASE, A_INT_B : millisec;
  begin
    if A ≥ B then
        A_INT_B := B;
        else A_INT_B := A;
    end if;
    if BASE ≥ A then
        A_DIFF_BASE := ⊤;
        else A_DIFF_BASE := A;
    end if;
    if BASE ≥ B then
        B_DIFF_BASE := ⊤;
        else B_DIFF_BASE := B;
    end if;
    if A_DIFF_BASE ≥ A_INT_B then
        if A_DIFF_BASE ≥ B_DIFF_BASE then
            return A_DIFF_BASE;
            else return B_DIFF_BASE;
        end if;
        else if A_INT_B ≥ B_DIFF_BASE then
            return A_INT_B;
            else return B_DIFF_BASE;
        end if;
    end if;
  end merge_m cp;
```

Figure 5.26: Algorithm merge_mcp.

After all three of the pieces required for change-merging the graphs have been built, then they must be change-merged using *graph_merge*, which contains two successive calls to *graph_union*, which we already know requires worst-case $\mathcal{O}(n + e)$ time. Therefore, the worst-case time complexity of change-merging three graphs is:

$$\mathcal{O}(tn^2) + \mathcal{O}(e^2) + \mathcal{O}(e^2) + \mathcal{O}(e^2) + \mathcal{O}(n + e) = \mathcal{O}(tn^2 + e^2)$$

The edges in the graph almost always outnumber the vertices, so we call this $\mathcal{O}(e^2)$.

The correctness of the Implementation Change-Merge is established by the correctness of the individual parts. The complexity of the Implementation Change-Merge is dominated by the complexity of the change-merging of the graphs, so the worst-case time complexity of the Implementation Change-Merge is $\mathcal{O}(e^2)$.

D. CREATING THE CHANGE-MERGED PROGRAM

The last algorithm used in this change-merging tool is *build_prototype*, shown in Figure 5.27. This algorithm takes the change-merged graph and removes the artificial timer edges and external vertex. It then sets the change-merged graph in the change-merged prototype.

```
Algorithm build_prototype(P : in out psdl_component; G : prototype_dependency_graph)
 A : psdl_graph;
 begin
    assign G to A;
    remove external vertex;
    remove timer dependency edges;
    set_graph(A, P);
 end build_prototype;
```

Figure 5.27: Algorithm *build_prototype*

The timer dependency edges are removed by iterating through the edges of the graph and removing the appropriate edges. This requires iteration over the edges of the change-merged graph, making the worst-case time complexity of this algorithm $\mathcal{O}(e)$.

E. ANALYSIS OF THE CHANGE-MERGING ALGORITHM

The correctness of the algorithm *change_merge* is established by the correctness of the individual parts. Since these individual algorithms are executed independently of one another, there are no dependencies between them, other than those already discussed. The complexity of this algorithm is calculated by adding the complexities of the individual parts. It is easy to see that the complexity is dominated by change-merging the graphs in the implementation part, which requires $\mathcal{O}(e^2)$ time, where e is the number of edges in the largest graph. Therefore, the worst-case time complexity of the entire algorithm is $\mathcal{O}(e^2)$.

VI. CAPS MERGE TOOL

In this chapter we describe an implementation for a change-merging tool resulting from this effort. The tool has been almost fully implemented and can easily be integrated into the CAPS Prototyping Environment. It is invoked through the Manager's Interface and provides a substantially beneficial tool for effective management of large software prototypes. Section A of this chapter describes the requirements for the tool. Section B provides the instructions for using the tool. Section C describes the testing performed on the tool.

A. REQUIREMENTS

The requirements for this tool are divided into three parts; interface, functionality and conflict reporting. Each of these parts are discussed separately in the subsections that follow:

1. Interface Requirements

a. *Interface must be consistent with other CAPS interfaces*: CAPs uses a menu-driven interface at the top level and windows with selection lists and pushbuttons at lower levels. To be consistent, a pushbutton type interface was required for the change-merge tool as well.

b. *User must be able to choose any prototype currently in the working directory*: The interface should provide a list of the prototypes currently in the user's working directory, and the capability for the user to select one of these prototypes.

c. *User must be able to select different versions and assign them to the different merge parameters by pushbutton*: After the prototype has been selected, the interface should

provide a list of the current versions of the prototype. The user should then be able to select each version by clicking with the mouse, and assign the selected version to one of the merge parameters, *base*, *version_a* or *version_b*, by pushing an assign button.

  d. *User should be satisfied that the selection made has been assigned to the correct parameter:* The interface should provide visual reinforcement that the selection made has been assigned to the correct parameter by showing the selected version in a window labeled with the parameter name.

  e. *User should be able to initiate the merge tool by pushing a button:* The interface should provide a button labeled "merge" which, when pushed, will call the merge tool for the parameters given.

  f. *User should be notified when merge is complete:* The interface should provide a pop-up window that alerts the user that the merge is complete. The result of the merge should be printed in a window labeled "result".

  g. *User should be notified if a conflict occurs:* The interface should provide a pop-up window that alerts the user that a conflict has occurred during the merge.

  h. *User should be able to commit the result to the design database directly from the merge interface:* A pushbutton should be provided that allows the manager using the tool to commit the result of the merge to the database, even if conflicts have occurred.

2. Functionality Requirements

  a. *Tool must be able to retrieve the three versions of the prototype when provided only the paths to their locations:* The interface will provide the full directory names for each of the input versions as input to the tool. The tool must be able to combine all of the PSDL source files in each of the version directories into one single file and call the PSDL parser to convert the text version of the prototype into an ADT representation of the prototype.

111

b. *Tool must call the PSDL expander to provide fully expanded PSDL programs as input to the change-merge procedure*: The PSDL expander is a tool which takes a multi-leveled prototype and converts it into a flat prototype with only one composite component.

c. *Tool must change-merge the three versions of the prototype according to the models provided by this dissertation*: The change-merge tool must be able to retrieve the composite component of each version, and perform the change-merge operation using these three components as input. It then must provide a new composite component for the merged prototype. The atomic components will then be added to the merged prototype according to which version supplied those components to the merged implementation graph.

d. *Tool must split the final version of the prototype into separate files for each of the component implementations and specifications*: The tool must be able to take the merged prototype and output it into separate files in the result directory. Each file should contain either the specification part or implementation part of one component. If the component is the composite component, then the name of the file will be "*prototype_name*.imp.psdl" or "*prototype_name*.spec.psdl", depending on whether it contains the implementation or specification part of the component, and where "*prototype_name*" is the name of the composite component. If the component is an atomic component, then the name of the file will be "*prototype_name.component_name*.imp.psdl" if it contains an implementation or "*prototype_name.component_name*.spec.psdl" if it contains a specification, where "*component_name*" is the name of the atomic component.

3. Conflict Reporting Requirements

a. *Tool must report to the user where in the merged component a conflict has occurred*: In each piece of the change-merged program where a conflict has occurred, the tool must place a flag indicating to the designer where the conflict occurred. This will prevent the user from having to search for the conflict in order to resolve it.

b. *Tool must provide at least a partially change-merged program in the case of all conflicts*: The tool will provide the most change-merged program possible whenever a conflict has occurred. Conflicts in one part of the program should not affect other parts of the program which are not dependent on the part with the conflict.

B. USING CAPS MERGE TOOL

To invoke the CAPS merge tool, select the merge prototypes option from the manager's interface. The CAPS merge tool window will be displayed as shown in Figure 6.1. The list of currently available prototypes will be displayed in the prototypes box at the lower left of the window.
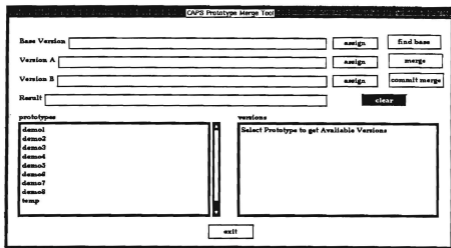
Figure 6.1: CAPS Prototype Merge Tool Interface

113

1. Selecting Prototypes and Versions

To select a prototype, click the left mouse button over the name of the prototype to be selected. Clicking twice on the same prototype will deselect the prototype. After selecting a prototype name, a list containing all of the versions of the selected prototype will appear in the versions box at the lower right of the merge tool window, as shown in Figure 6.2. To select a version, click the left mouse button on top of one of the versions. Again, double clicking on the same version will deselect the version.
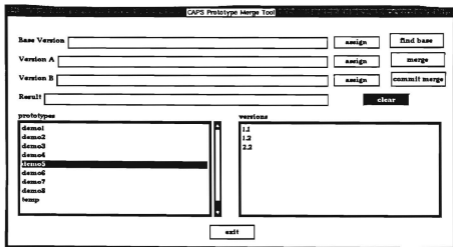
Figure 6.2: CAPS Prototype Merge Tool Interface with List of Versions

2. Performing the Merge Operation

To perform a merge, three versions must be selected and assigned to the merge parameter boxes. To assign parameters, first select the version, then click the left mouse button on the "assign" button next to the parameter to be assigned. Figure 6.3 shows the window after all three parameters have been assigned. Changing a parameter assignment is

114

done by selecting the correct version and clicking on the "assign" button again. Every push of the "assign" button reassigns the parameter to the selected version. To clear all of the parameter assignments, use the clear button located on the right center of the window.
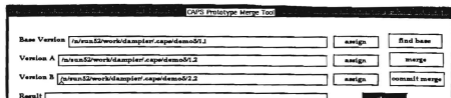
Figure 6.3: Assignment of Parameters

When all of the parameters have been assigned, the "merge" button located on the right side of the window must be pushed. This will invoke the change-merge tool. When the change-merge tool has completed its execution, one or two windows will appear on the screen. The "merge complete" window, shown in Figure 6.4, will always appear after execution of the tool. If a conflict was detected during the change-merge, the "conflict notification" window, as shown in Figure 6.5, will also appear. The manager can either choose to keep the result and manually resolve the conflicts or abandon the result and start again.
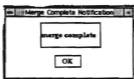
Figure 6.4: Merge Complete
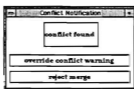
Figure 6.5: Notification of Conflict

3. Commit Merge

Once the merge has been completed, the manager has to specifically commit the result to the design database. To commit the merged result, the manager clicks the left mouse button on the "commit merge" button on the right side of the CAPS Merge Tool window. A new version number will be assigned to result and it will be added to design database as a permanent part of the prototypes configuration.

At this point, the manager can choose to perform another merge operation or exit the tool. If another merge is desired repeat the process described above as many times as desired. To exit the CAPS Merge Tool, click the left mouse button on the "exit" button at the bottom of the window, and control will be returned to the CAPS Manager Interface.

C. TESTING

We tested the change-merging tool by applying it to a series of sample prototype projects each testing a different part of the tool. These projects included real prototypes which were developed by students in the CAPS Research Team as well as examples constructed specifically for this test. The largest of these prototypes is the Command and Control System described in [Ref. 38]. The implementation for this prototype contains 27 vertices and 35 edges with a full range of control constraints. Four modified versions of this prototype were

prototype were created, and the change-merge tool was applied to different combinations of the four, each testing a different part of the tool.

Timing tests were conducted to provide a realistic assessment of the speed with which the tool would operate. During the test it was determined that the time required to process each of the input files (combining the multiple files into one, parsing the input files, and expanding the prototype to a flat graph) took a significant amount of the time for the system to run. In the case of the Command and Control prototype, the system took on average six seconds to process each file and 25 seconds to change-merge them. In the case of the smaller prototypes, the times were significantly less.

Since the current implementation is not as efficient as an optimal one outlined by the algorithms in Chapter V, we can expect a significant speedup for the optimal implementation. In each test of the change-merge tool, the results were exactly as expected. The tool produced conflicts whenever expected and correct results when they were possible.

VII. CONCLUSION

A. WHAT WE HAVE ACCOMPLISHED AND WHY IT IS IM-PORTANT

The purpose of this research was to provide a computer-aided method for combining and integrating the contributions of different people working on the same prototype. It is commonly known that one of the most time consuming and problematic parts of developing large software systems is combining independently developed pieces of the system and ensuring they do not conflict. We developed a computer-aided method for merging changes to a prototype which will always produce a correct result or report a potential conflict. Using this method provides a prototype development manager with the ability to assign different development tasks for the same prototype to different members of the development team and be assured that the pieces can be integrated together after their completion in a safe manner. This method will either produce a change-merged prototype that is correct with respect to the different updates or it will notify the manager that a conflict has occurred.

We found a solution to an analog of this problem in previous work done on integrating different versions of while programs at the University of Wisconsin [Ref. 28, 42, 29]. The main difference between their method and ours is that while programs are very different from data flow programs. Data flow programs are inherently parallel and non-deterministic, and the class of enhanced data flow programs used in PSDL also include hard real-time constraints.

We proved our method correct by observing that slices of prototypes which isolate a portion of the prototype's behavior will always behave the same in any prototype where they are well defined slices. Using the Slicing Theorem in Chapter IV, we were able to show

that as long as the slice of the merged version of the prototype with respect to the affected parts of each modification was the same as the same slice of the modification, the changes introduced by that modification were preserved by the method.

To prove this theorem, we had to develop a computational model of the PSDL language. Chapter IV provides a detailed development of the model of the language from defining the behavior on a single stream in the prototype to constructing the behavior of the entire prototype from the behaviors of the individual operators in the prototype. This construction is possible because we showed in the Independent Operator Lemma in Appendix B that the possibility function for an operator is not determined by the context in which it is placed. As long as the operator is given the same input, it will behave in precisely the same way in any prototype.

From the model, we developed an algorithm which can perform the change-merge in $\mathcal{O}(e^2)$ time and $\mathcal{O}(n^2)$ space, and an implementation which provides a working change-merge tool to be used in the Computer-Aided Prototyping System. The algorithm and tool demonstrates the feasibility of our method for problems of practical size.

During the course of this research, we also proposed an improved method for slicing and merging while programs which provides a strictly more accurate method than previously defined methods. No proof of this method is provided however. That will be left to future work.

B. WHAT STILL NEEDS TO BE DONE

We couldn't possibly solve all of the world's problems in the short amount of time provided, so there are still many out there to be tackled. Some of the problems that we intend to continue working on are providing a method for change-merging different versions of an abstract data type written in PSDL. Our method will currently handle the operator implementations for ADTs, but fails to provide a method for integrating the data representations.

Another area which needs consideration is the area of merging programs in high level programming languages, like Ada. In prototyping large systems, it is very important to automate as many of the development tasks as possible to minimize the drain on resources caused by monotonically decreasing budgets. One of the first tasks to be finished is completing the formalization of the conditional program merging we proposed in Chapter III.

Another area that warrants further study is in further improving the conflict detection methods used in change-merging. Automatic conflict resolution tools would provide project managers with an even greater degree of confidence in the change-merging tools.

APPENDIX A

FORMAL DEFINITIONS

This appendix contains formal definitions of the types, properties, and functions used in our behavioral model of PSDL.

1. TYPE DEFINITIONS

a. **data_tuple{t: type}** =
    $tuple\{value : \text{t}, operator : \textbf{op\_id}, write\_time : \textbf{real}, read\_time : \textbf{real}\}$

b. **trace{t: type}** = $sequence\{\textbf{data\_tuple}\{\textbf{t}\}\}$

c. **stream_behavior{t : type}** = $set\{\textbf{trace}\{\textbf{t}\}\}$

d. **trace_tuple{P : prototype}** = $tuple\{\textbf{trace}\{type\{\textbf{s}\} :: \textbf{s} \in E(P)\}\}$

e. **prototype_behavior{P : prototype}** = $set\{\textbf{trace\_tuple}\{P\}\}$

f. **incremental_trace_tuple{t : write_time}** =
    $tuple\{\textbf{data\_tuple}\{type\{\textbf{s}\} \text{ SUCH THAT } \textbf{s} \in E(P)\}$
        $:: \textbf{data\_tuple}.write\_time = t\}$

2. INVARIANT DEFINITIONS

We assume that the implementation of an operating system where a PSDL prototype is being executed will guarantee mutual exclusion when two operators executing in parallel wish to write to the same stream at the same time. Because we assume this control on write access for data streams, we can guarantee the following invariant is true for all data streams in a PSDL implementation.

a. monotonic_time(t : **trace**)
    ALL (i, j: **nat** SUCH THAT $1 \leq i < j \leq$ length(t)
      :: $t[i].write\_time < t[j].write\_time$
-- The write times in a trace are monotonically increasing.

Since an operator writing to a data stream had to read from its input streams before it completes execution, we can guarantee that any data tuple in a trace will satisfy the following invariant:

b. firing_invariant(t : **trace**)
    ALL (i: **nat** SUCH THAT $1 \leq i \leq$ length(t)
      :: ($t[i].read\_time < t[i].write\_time$) | ($t[i].read\_time = t[i].write\_time = 0$))
-- The write time in any data tuple is strictly greater than the read time
-- in that data tuple, unless it is the initial data tuple.

If an operator receives input from a feedback loop, then the vertex associated with that operator is on a cycle in the PSDL implementation graph. It is necessary to know that an operator is on a cycle because this information affects the possibility function of that operator.

c. on_a_cycle(o : **op_id**)
    -- o provides output to a feedback loop which in turn provides input to o.

3. FUNCTION DEFINITIONS AND PROPERTIES

a. Merging Traces

In PSDL, it is possible for more than one operator to write into the same stream. If this is the case, each of these operators independently writes a sequence of data tuples to that stream. These sequences merge to form a single trace for the stream. The function *merge* specified below shows that this combination of sequences is well-defined. Propositions 4 through 8 state properties about the function *merge* which are needed for our discussion of possibility functions in chapter IV.

$merge(t1, t2 : \textbf{trace SUCH THAT ALL } (i, j : \textbf{nat} :: t1[i].write\_time \neq t2[j].write\_time)$
    REPLY $(t3 : \textbf{trace})$

        WHERE $monotonic\_time(t3)$ & $firing\_invariant(t3)$ &
           $length(t3) = length(t1) + length(t2) - 1$ & $t3[0] = t2[0] = t3[0]$
           -- Every trace contains an initial data_tuple with index zero.

        ALL $(i: \textbf{nat SUCH THAT } 1 \leq i \leq length(t3)$
        :: SOME$(j: \textbf{nat}:: (t3[i] = t1[j]$ & $1 \leq j \leq length(t1))$
              $| (t3[i] = t2[j]$ & $1 \leq j \leq length(t2)))),$

        ALL $(i: \textbf{nat SUCH THAT } 1 \leq i \leq length(t1)$
        :: SOME$(j: \textbf{nat}:: t3[j] = t1[i]$ & $1 \leq j \leq length(t3))),$

        ALL $(i: \textbf{nat SUCH THAT } 1 \leq i \leq length(t2)$
        :: SOME$(j: \textbf{nat}:: t3[j] = t2[i]$ & $1 \leq j \leq length(t3)))$

Proposition 4 $merge$ **is well-defined**

    merge is a total, single-valued function over the specified domain.

Proof

    Let $t1$ and $t2$ be traces on a stream SUCH THAT

    $\forall i, j \in \aleph, t1[i].write\_time \neq t2[j].write\_time.$

    Suppose $t3 = merge(t1, t2)$ and $t4 = merge(t1, t2)$ SUCH THAT $t3 \neq t4.$

    Since $t3$ and $t4$ are both valid results of $merge(t1, t2)$, we conclude that they both
satisfy $monotonic\_time$ and $length(t3) = length(t4) = length(t1) + length(t2) - 1.$

    Since $t3 \neq t4, \exists i \in \aleph \mid i < length(t3)$ SUCH THAT

    $t3[i].value \neq t4[i].value$ or $t3[i].operator \neq t4[i].operator$ or

    $t3[i].write\_time \neq t4[i].write\_time$ or $t3[i].read\_time \neq t4[i].read\_time$ or

    But, by the definition of $merge$, every element in both $t3$ and $t4$ are elements
of either $t1$ or $t2$, which have no common write_times, and since both $t3$ and $t4$ satisfy

monotonic_time, there is only one way to combine the elements of $t1$ and $t2$ into a single trace that satisfies *monotonic_time*. Thus $t3 = t4$, and we have a contradiction. *merge* is well-defined. $\Box$

Proposition 5 *merge* **satisfies** *monotonic_time*

If *monotonic_time*($t1$) and *monotonic_time*($t2$) then *monotonic_time*(*merge*($t1, t2$)).

Proof

We assume that $t1$ and $t2$ have no common write times, and that *monotonic_time* is satisfied for both $t1$ and $t2$. Let $t3 = merge(t1, t2)$. We show $t3$ satisfies *monotonic_time* by induction.

Basis: $length(t3) = 1$

Since every trace has a data_tuple with index zero, then $t3$ is the trace with only an initial_data_tuple, and *monotonic_time* is satisfied.

Induction Step: Assume that $t3 \mid k$ satisfies *monotonic_time*. Since $t1$ and $t2$ satisfy *monotonic_time*, and they do not contain data_tuples with the same write times, we know that no matter which of $t1$ and $t2$ the $k + 1$st element of $t3$ comes from, its write time will be greater than tw_k. So, we can conclude that the $k + 1$ element of $t3$, when added will have *write_time* greater than $t3[k]$. Since $t3 \mid k$ satisfies *monotonic_time*, and $t3[k + 1]$.*write_time* $> t3[k]$.*write_time*, we conclude that *monotonic_time*($t3$) is satisfied. $\Box$

Proposition 6 *merge satisfies firing_invariant*

> *If firing_invariant is satisfied for both t1 and t2 then firing_invariant is satisfied for merge(t1, t2).*

Proof

We assume that $t1$ and $t2$ have no common write times, and that *firing_invariant* is satisfied for both $t1$ and $t2$. Let $t3 = merge(t1, t2)$. We show *firing_invariant(t3)* by induction.

Basis: $length(t3) = 1$

Since every trace has a data_tuple with index zero, then $t3$ is the trace with only an initial data tuple, and $tw_0 = tr_0 = 0$ by the definition of initial data tuples. So *firing_invariant(t3)* is satisfied.

Induction Step: Assume that $t3 \mid k$ satisfies *firing_invariant*. Since $t1$ and $t2$ satisfy *firing_invariant* and each element of $t3$ is also an element of $t1$ or $t2$, we know that no matter which of $t1$ and $t2$ the $k + 1$st element of $t3$ comes from, $tw_{k+1} > tr_{k+1}$. We can conclude therefore that *firing_invariant(t3)* is satisfied. □

Proposition 7 *merge is commutative.*

$$merge(t1, t2) = merge(t2, t1)$$

Proof

We assume that $t1$ and $t2$ have no common write_times. We further assume that since $t1$ and $t2$ are traces, they both satisfy the trace invariants *monotonic_time* and *firing_invariant*.

We show that the *merge* function applied to $t1$ and $t2$ satisfies these conditions and the length of the result is exactly the same regardless of the order of the parameters.

Let $t3 = merge(t1, t2)$.

Then $length(t3) = length(merge(t1, t2)) = length(t1) + length(t2) - 1$.

But, since $+$ is commutative, then

$$length(t1) + length(t2) - 1 = length(t2) + length(t1) - 1 = length(merge(t2, t1)).$$

We know by Proposition 5 that $monotonic\_time$ is satisfied for both $merge(t1, t2)$ and $merge(t2, t1)$. We know by Proposition 6 that $firing\_invariant$ is satisfied for both $merge(t1, t2)$ and $merge(t2, t1)$. Therefore $merge$ is commutative. $\square$

Proposition 8 $merge$ is associative.

    If $t1$, $t2$ and $t3$ each satisfy monotonic_time and firing_invariant, and they have no common times, then $merge(t1, merge(t2, t3)) = merge(merge(t1, t2), t3)$.

<u>Proof</u>

We assume that $t1$, $t2$ and $t3$ have no common write_times. We further assume that since $t1$, $t2$ and $t3$ are traces, they all satisfy the conditions $monotonic\_time$ and $firing\_invariant$.

Let $t4 = merge(t1, merge(t2, t3))$.

Then $length(t4) = length(merge(t1, merge(t2, t3))) =$
$length(t1) + length(merge(t2, t3)) - 1 =$
$length(t1) + (length(t2) + length(t3) - 1) - 1 =$
$length(t1) + length(t2) + length(t3) - 1 - 1 =$
$(length(t1) + length(t2) - 1) + length(t3) - 1 =$
$length(merge(t1, t2)) + length(t3) - 1 =$
$length(merge(merge(t1, t2), t3))$.

So the lengths of $merge(t1, merge(t2, t3))$ and $merge(merge(t1, t2), t3)$ are the same. We know by Proposition 5 that $monotonic\_time$ is satisfied for both $merge(t2, t3)$ and $merge(t1, t2)$. Thus using the same logic, we can conclude that $monotonic\_time$ is satisfied for $merge(t1, merge(t2, t3))$ and for $merge(merge(t1, t2), t3)$. We know by Proposition 6 that $firing\_invariant$ is satisfied for both $merge(t2, t3)$ and $merge(t1, t2)$. Thus using the

same logic, we can conclude that $firing\_invariant$ is satisfied for $merge(t1, merge(t2, t3))$ and for $merge(merge(t1, t2), t3)$. Therefore $merge$ is associative. □

b. Other Functions

This section contains definitions for other functions used to construct the possibility functions for prototypes.

(1) ⊕

The "⊕" function extends trace_tuples by appending incremental_trace_tuples to them. It takes as input a trace_tuple T and an incremental_trace_tuple_set S. The output of the function is a set of trace_tuples where the prefix of each element in the set is T, and the remainder of each element is an element of S.

"⊕"(T: **trace_tuple**, S: **incremental_trace_tuple_set**)
REPLY (D: **trace_tuple_set**)
 ALL (tt: **trace_tuple** SUCH THAT $tt \in D$
  :: SOME(d: **incremental_trace_tuple** SUCH THAT $d \in S$
   :: $tt = $ **append**(T, d)))

(2) Δ

The Δ function is used to select incremental_trace_tuples which have a particular write_time.

Δ(t: **time**, $S1$: **incremental_trace_tuple_set**)
REPLY $S2$: **incremental_trace_tuple_set**
 ALL (D: **incremental_trace_tuple** SUCH THAT $D \in S2$
  :: ALL(d: **data_tuple** SUCH THAT $d \in D$
   :: $d.write\_time = t$))

127

(3) ρ

The ρ function provides the earliest possible time that an operator can read its input streams based on its output history. If the operator is on a cycle in the graph, then it must complete every firing to include writing its output streams before it can read its input streams. If it is not on a cycle, then it does not have to wait for a previous firing to be complete before it can read its input streams again.

$\rho(T: \text{\bf trace\_tuple}, o: \text{\bf op\_id}) \text{ REPLY } t: \text{time}$
$\quad \text{SOME}(s: \text{\bf stream\_set} \text{ SUCH THAT } s \subseteq O(o)$
$\quad\quad :: \text{ALL}(\tau: \text{\bf trace} \text{ SUCH THAT } \tau \in T_s$
$\quad\quad\quad :: \text{WHEN on\_a\_cycle}(o)\ t \geq \tau[length(\tau) - 1].write\_time$
$\quad\quad\quad :: \text{OTHERWISE } t \geq \tau[length(\tau) - 1].read\_time$

(4) *fill*

The *fill* function takes as input an incremental_trace_tuple_set from the output streams of an operator and for each **incremental_trace_tuple** in the set, it creates an empty data_tuple for all other streams in the prototype.

128

APPENDIX B

EFFECTS OF CONTROL CONSTRAINTS ON POSSIBILITY FUNCTIONS

This appendix defines the effect of each form of constraint contained in the PSDL grammar on the possibility function of an operator. The possibility function for an operator o is a function of the form, $\mathcal{F}_o(I_o, rt)$, where I_o is the input history of the operator o, and rt is the last possible time that o could have read its streams for the current firing. The output of the possibility function is a set of possible incremental_trace_tuples written to the output streams of o by the current firing of o. Examples of possibility functions can be found in Chapter IV, Section A.3, Examples 5 and 6.

The main result in this appendix is Lemma 1, the **Independent Operator Lemma**, which states that the possibility function of an operator is not dependent on the context in which it is placed.

Lemma 1 Independent Operator Lemma

Given the same input history and an unlimited number of processors, an operator has the same possibility function regardless of whether it is contained in a larger prototype, as long as the larger prototype does not introduce input to the operator from a feedback loop.

Proof:

This proof is a structural induction over all of the different control constraints in the PSDL grammar. First, let us look at the possibility function for an operator o, $\mathcal{F}_o$. This possibility function produces a set of possible incremental trace tuples for every finite prefix of input vectors written to the operator's input streams.

Each of the following sections discusses how each of the control constraints available in PSDL affects the possibility function for o.

1. Triggers & Input Guards

The "Triggered" control constraint defines the conditions which trigger the execution of o. The two options, "by all" and "by some" identify any input streams listed after them as data flow streams or sampled streams respectively, and any time a value is written to one of those streams, it can only be removed from the stream by a firing of the operator o for data flow streams, and a producer operator for sampled streams. Another option which may appear in a triggering constraint is an input guard. These appear as boolean expressions that, if satisfied, allow the operator o to fire.

a. "by all"

The "by all *stream_set*" trigger appearing in a control constraint limits the execution of the operator o to fire only when there is a new value on each of the streams in *stream_set*. The effect of this on $\mathcal{F}_o$ is that it limits the read times for which o can produce a set of non-empty incremental_trace_tuples. Since the output of $\mathcal{F}_o$ is determined only by the input history of o, which we have assumed to be the same in any context, this only serves to limit the possible output histories. These output histories are the same regardless of whether o is contained in a larger prototype or functions independently.

b. "by some"

A similar argument can be made for the "by some *stream_set*" trigger. The input sequences processed by o are limited to only those sequences of vectors in which at least one of the streams listed in *stream_set* contains a new value. The effect of this limitation is the same as in the previous section.

Input guards in the triggering condition of an operator, o, define which input values can trigger the execution of o. Their effect is to limit the read times at which the operator can fire. Since that effect only serves to limit the execution of o, it is the same whether or not the operator is contained in a larger prototype or not.

2. Period

Operators with a *period* constraint are declared with a time t. After an initial delay of as much as t time, the operator is given a window of t amount of time in which to fire. As long as the operator fires early enough to complete its execution before the end of the period, a set of possible outputs will be written to its output streams. This set of outputs is produced non-deterministically because of the flexibility the operator has in starting its execution. This non-deterministic start time will change the time that the operator reads its input streams, thereby changing the possible outcome. Since, we are assuming that the number of processors is unlimited, we conclude that no matter whether the operator is contained in a larger prototype or not, all choices for read times are possible, thus the possibility function for the operator will be the same in either case.

3. Finish Within, Minimum Calling Period & Maximum Response Time

Operators with a *finish within, minimum calling period* or *maximum response time* constraint are declared with a time, t. The minimum calling period constraint serves to limit the possible read times of the operator, and the finish within and maximum response time constraints limit the possible write times of the operator, however these constraints are not dependent on the context in which the operator is placed as long as the number of

processors is not limited. Therefore, the possibility function for the operator is the same whether it is contained in a larger prototype or not.

4. Constraint Options

Constraint options include *output guards*, *exceptions* and *timer operations*. Output guards can affect the possibility function of an operator, but these are part of the definition of the operator. Thus, the effect of these output guards on the possibility function for the operator is the same regardless of whether the operator is contained in a larger prototype. Exception triggers contained in the implementation of an operator can affect outputs on streams of type exception, and their triggering is affected only by the inputs provided to the operator, so the exception outputs resulting from possible inputs to the operator would be the same regardless of whether the operator is contained in a larger prototype. Timer operations affect the outputs on timer dependency edges only. These timer operations affect the state of a timer if some predicate evaluated on the inputs to the operator is satisfied. Since the inputs to the operator are the same when the operator is contained in a larger prototype, the resultant timer state change operations will be the same if the operator is contained in a larger prototype.

Since the control constraints and the output history of an operator can only depend on the input received from the data streams and timer dependency edges, and we know these to be the same, putting the operator in the context of a larger prototype can not affect its possibility function. □

It is important to note that Lemma 1 applies equally to operators which are components of larger operators, or operators which implement some operation in an abstract data type. From our perspective, there is no difference.

APPENDIX C

PROOFS OF THEOREMS

1. Φ: *Traces* $\longrightarrow$ *FunctionRepresentations* **IS WELL-DEFINED AND A BIJECTION**

THEOREM 2:

Proof

1. Show Φ is single-valued and a total function. Let r be a trace on a stream, and let Ψ_1 and Ψ_2 be two functional representations for r SUCH THAT $\Psi_1 \neq \Psi_2$.

Since $\Psi_1 \neq \Psi_2$, $\exists$ a time $t \in [0, \infty)$ SUCH THAT $\Psi_1(t) \neq \Psi_2(t)$.

But, then by the definition of Φ^{-1} $\exists n \leq min(\mathbf{length}(\Phi^{-1}(\Psi_1)), \mathbf{length}(\Phi^{-1}(\Psi_2)))$

SUCH THAT

$\Phi^{-1}(\Psi_1)[n].value \neq \Phi^{-1}(\Psi_2)[n].value$ or

$\Phi^{-1}(\Psi_1)[n].operator \neq \Phi^{-1}(\Psi_2)[n].operator$ or

$\Phi^{-1}(\Psi_1)[n].write\_time \neq \Phi^{-1}(\Psi_2)[n].write\_time$ or

$\Phi^{-1}(\Psi_1)[n].read\_time \neq \Phi^{-1}(\Psi_2)[n].read\_time.$

Thus, $\Phi^{-1}(\Psi_1) \neq \Phi^{-1}(\Psi_2)$, but we know that $\Phi^{-1}(\Psi_1) = \Phi^{-1}(\Psi_2) = r$, and

we have a contradiction.

Therefore, Φ is well defined.

2. Show Φ is onto. Let

$$M = \begin{array}{ll} [0, t_1) & \longrightarrow & [\bot, \bot, 0] \\ [t_1, t_2) & \longrightarrow & [x_1, o_1, tr_1] \\ \cdots & & \\ [t_n, t_{n+1}) & \longrightarrow & [x_n, o_n, tr_n] \end{array}$$

be a mapping in Ψ.

133

Then by definition of Φ,

$$M = \Phi(\langle [\perp, \perp, 0, 0], [x_1, o_1, tr_1, tw_1], \cdots, [x_n, o_n, tr_n, tw_n], \cdots \rangle)$$

or $t_{n+1} = \infty$.

Therefore, Φ is onto.

3. Show $r \neq s \implies \Phi(r) \neq \Phi(s)$

Let r and s be two traces on a stream, where $r \neq s$.

Then $\exists n \leq max(\mathbf{length}(r), \mathbf{length}(s))$ SUCH THAT

$r[n].value \neq s[n].value$ or $r[n].operator \neq s[n].operator$ or

$r[n].write\_time \neq s[n].write\_time$ or $r[n].read\_time \neq s[n].read\_time$.

If $r[n].write\_time \neq s[n].write\_time$

then $\exists t \leq min(r[n].write\_time, s[n].write\_time)$ SUCH THAT

$\Phi(r)(t) \neq \Phi(s)(t)$.

If $(r[n].read\_time \neq s[n].read\_time$ or $r[n].value \neq s[n].value)$

and $r[n].write\_time = s[n].write\_time$

then $\exists t = r[n].write\_time = s[n].write\_time$ SUCH THAT

$\Phi(r)(t) \neq \Phi(s)(t)$.

Therefore, $\Phi(r) \neq \Phi(s)$, and Φ is one-to-one.

By 1, 2 & 3, Φ is a Bijection. $\square$

2. SLICING THEOREM FOR PSDL PROTOTYPES

THEOREM 3: Slicing Theorem

Let $S_P(X)$ be the slice of a prototype P with respect to a set of streams X. Then $S_P(X)$ and P have the same **prototype behavior** on any subset of the streams in $S_P(X)$.

Proof

Let $S_P(X)$ be an arbitrary slice of a prototype P. We show that at any point during the execution of $S_P(X)$, both P and $S_P(X)$ have the same truncated **prototype behavior** over the data streams in $S_P(X)$. From this, we conclude that the **prototype behavior** over any subset of the data streams in $S_P(X)$ is the same in both the slice and the prototype.

Using Φ, we view each of the trace tuples in $\mathbf{B}_{E(S_P(X))} \mid k$ as a sequence of vectors, each vector containing a data tuple from each data stream in $E(S_P(X))$ and do an induction over the length of the longest sequence.

Induction Hypothesis:

If the length of the longest sequence of vectors in $\mathbf{B}_{E(S_P(X))}$ is no more than k, then $\mathbf{B}_{E(S_P(X))}$ is the same in both P and $S_P(X)$.

Basis: (Sequence of length one)

The semantics of PSDL determine an initial data tuple for each stream. The read time and write time of this initial data tuple are both 0. If the stream is declared as a state variable, then the initial data tuple contains a data value specified by the STATE declaration, and otherwise it contains the undefined data value $\perp$. The operator field of the data tuple contains either the **id** of the operator containing the state variable declaration for the stream, if one is declared, or $\perp$. Since the state variable declarations are the same in both P and $S_P(X)$, the $\mathbf{B}$ over all of the streams in $S_P(X)$ is the same in both, when the length of the longest sequence of vectors is one.

Induction Step: ($\mathbf{B}_{E(S_P(X))}$ is a sequence of length $k + 1$)

Equation 1 shows us that $\mathbf{B}_{E(S_P(X))} \mid (k+1)$ is completely determined by $\mathbf{B}_{E(S_P(X))} \mid k$.

$$\mathbf{B}_{E(S_P(X))} \mid (k+1) =$$

$$\bigcup_{T \in \mathbf{B}_{E(S_P(X))|k}} \left[T \oplus \bigcup_{S \in \mathcal{P}(V(S_P(X)))} \left(\bigoplus_{o \in S} \left(\bigcup_{\rho(T,o)<tr} \left(\bigcup_{tr<t} \Delta \left(t, \mathit{fill}\left(E(P), \mathcal{F}_o(T_{I(o)}, tr) \right) \right) \right) \right) \right) \right] \tag{1}$$

Since $\mathbf{B}_{E(S_P(X))} \mid k$ is the same in the slice and the entire prototype by the induction hypothesis, $\mathbf{B}_{E(S_P(X))} \mid (k+1)$ must also be the same in the slice and the entire prototype.

Consider the main subexpression of the right hand side of Equation 1:

$$T \oplus \bigcup_{S \in \mathcal{P}(V(S_P(X)))} \left(\bigoplus_{o \in S} \left(\bigcup_{\rho(T,o)<tr} \left(\bigcup_{tr<t} \Delta \left(t, \mathit{fill}\left(E(P), \mathcal{F}_o(T_{I(o)}, tr) \right) \right) \right) \right) \right)$$

This construction defines a set of trace tuples of length $k+1$ in terms of a trace tuple τ of length k and a set of incremental trace tuples of length one that is derived from T and the properties of the slice. This set of trace tuples is a subset of $\mathbf{B}_{E(S_P(X))} \mid (k+1)$. The $\oplus$ operation is a function, so, providing the trace tuple, T, and the set of incremental trace tuples are the same in both P and $S_P(X)$, the resultant subset of $\mathbf{B}_{E(S_P(X))} \mid (k+1)$ is the same in both P and $S_P(X)$.

The set of trace tuples, $\mathbf{B}_{E(S_P(X))} \mid k$ is the projected $\mathbf{B}$ of P over the streams in $S_P(X)$, so any trace tuple, $T \in \mathbf{B}_{E(S_P(X))} \mid k$ is certainly the same in both P and $S_P(X)$, as $S_P(X)$ is a subgraph of P.

The set of possible incremental trace tuples, D, used in the above construction is constructed using the following equation:

$$D = \bigcup_{S \in \mathcal{P}(V(S_P(X)))} \left(\bigoplus_{o \in S} \left(\bigcup_{\rho(T,o)<tr} \left(\bigcup_{tr<t} \Delta \left(t, \mathit{fill}\left(E(P), \mathcal{F}_o(T_{I(o)}, tr) \right) \right) \right) \right) \right)$$

D is constructed by looking at every possible subset of the operators in $S_P(X)$, building the set of possible incremental trace tuples for the output streams of that subset and finding the union over all subsets. Since the powerset of the the set of operators in $S_P(X)$, $\mathcal{P}(V(S_P(X)))$,

is the same in both P and $S_P(X)$, then the union over all of the subsets is the same in both P and $S_P(X)$ provided that the incremental trace tuples produced for each subset are the same in both.

Pick an arbitrary $S' \in \mathcal{P}(V(S_P(X)))$. We want to construct the set of possible incremental trace tuples over the output streams of the operators in S'. To do this, we must look at each operator, and construct the set of possible incremental trace tuples over their output streams. Then, we take each of those and combine them using the $\oplus$ function. Since an incremental trace tuple is simply a trace tuple of length one, the function $\oplus$ can be overloaded to accomplish this task as well. The operator $\oplus$ is a commutative function, so as long as the incremental trace tuples produced by each operator are the same in both P and $S_P(X)$, their combination using $\oplus$ is the same in both P and $S_P(X)$. Accordingly, we pick an arbitrary operator, v. Constructing the set of possible incremental trace tuples for o is accomplished using the following:

$$\bigcup_{\rho(T,o)<tr} \left(\bigcup_{tr<t} \Delta \left(t, fill \left(E(P), \mathcal{F}_o(T_{I(o)}, tr) \right) \right) \right)$$

By Lemma 1, we know that the set of incremental trace tuples produced by o is the same in both P and $S_P(X)$. Now since we already knew that T is the same in both P and $S_P(X)$, and we know that $\oplus$ is a function, we conclude that the resultant set of trace tuples is the same in both P and $S_P(X)$, for each $T \in \mathbf{B}_{E(S_P(X))} \mid k$. Further, we conclude that the union over all possible trace tuples in $\mathbf{B}_{E(S_P(X))} \mid k$ is the same in both P and $S_P(X)$. Therefore, $\mathbf{B}_{E(S_P(X))} \mid (k+1)$ is the same in both P and $S_P(X)$.

Since any finite **prototype behavior** over the set of streams in the slice is the same in both P and $S_P(X)$, we conclude that any finite **behavior** over a subset of the streams in the slice is the same in both P and $S_P(X)$.

Now, we want to show that any countably infinite **prototype behavior** is the same in both P and $S_P(X)$. Assume not. If any countably infinite **prototype behavior** is not

137

the same in both, then there must be a finite prefix which is not the same in both P and $S_P(X)$. However, according to our induction above, all finite subsequences of $\mathbf{B}_{E(S_P(X))}$ are the same in both, thus we have a contradiction. Therefore, any countably infinite **prototype behavior** over a subset of the streams in $S_P(X)$ is the same in both P and $S_P(X)$. $\square$

The construction shown in Equation 1 defines the behavior of a prototype in PSDL. Since PSDL is non-deterministic and can be executed in parallel, it is necessary for us to consider all possible execution circumstances. What the construction really does is lengthen the prototypes behavior one incremental_trace_tuple at a time. This incremental_trace_tuple added to the end of the behavior at some time t is the output of every operator in the prototype that is writing to its output streams at precisely time t. This can be every operator in the prototype or only one operator in the prototype.

APPENDIX D

PSDL Grammar

The following is the grammar listing for the Prototyping System Description Language (PSDL) as of 14 November 1991. This version corresponds to the implementation of our merging tool. Optional items are enclosed in [square brackets]. Items which may appear zero or more times appear in { braces }. Terminal symbols appear in **BOLDFACE**. Groupings appear in (parentheses).

```
*****************************************************************************

psdl
      = {component}

component
      = data_type
      | operator

data_type
      = type id type_spec type_impl

type_spec
      = specification [generic type_decl] [type_decl]
        {operator id operator_spec}
        [functionality] end

operator
      = operator id operator_spec operator_impl

operator_spec
      = specification {interface} [functionality] end

interface
      = attribute [reqmts_trace]
```

attribute
    = **generic** type_decl
    | **input** type_decl
    | **output** type_decl
    | **states** type_decl **initially** initial_expression_list
    | **exceptions** id_list
    | **maximum execution time** time

type_decl
    = id_list : type_name {, id_list : type_name}

type_name
    = id
    | id [type_decl]

id_list
    = id {, id}

reqmts_trace
    = **required by** id_list

functionality
    = [keywords] [informal_desc] [formal_desc]

keywords
    = **keywords** id_list

informal_desc
    = **description** { text }

formal_desc
    = **axioms** { text }

type_impl
    = **implementation ada** id **end**
    | **implementation** type_name {**operator** id operator_impl} **end**

operator_impl
    = **implementation ada** id **end**
    | **implementation** psdl_impl **end**

psdl_impl
    = data_flow_diagram [streams] [timers] [control_constraints] [informal_desc]

data_flow_diagram
    = **graph** {vertex} {edge}

vertex

        = **vertex** op_id [: time]
          – time is the maximum execution time

edge

        = **edge** id [: time] op_id → op_id
          – time is the latency

op_id

        = id [([id_list] | [id_list])]

streams

        = **data stream** type_decl

timers

        = **timer** id_list

control_constraints

        = **control constraints** constraint {constraint}

constraint

        = **operator** op_id
          [**triggered** [trigger] [**if** expression] [reqmts_trace]]
          [**period** time [reqmts_trace]]
          [**finish within** time [reqmts_trace]]
          [**minimum calling period** time [reqmts_trace]]
          [**maximum response time** time [reqmts_trace]]
          {constraint_options}

constraint_options

        = **output** id_list **if** expression [reqmts_trace]
        | **exception** id [**if** expression] [reqmts_trace]
        | timer_op id [**if** expression] [reqmts_trace]

trigger

        = **by all** id_list
        | **by some** id_list

timer_op

        = **reset timer**
        | **start timer**
        | **stop timer**

initial_expression_list

        = initial_expression , initial_expression

initial_expression
        = **true**
        | **false**
        | integer_literal
        | real_literal
        | string_literal
        | id
        | type_name . id [(initial_expression_list)]
        | (initial_expression)
        | initial_expression binary_op initial_expression
        | unary_op initial_expression

binary_op
        = **and**
        | **or**
        | **xor**
        | **<**
        | **>**
        | **=**
        | **>=**
        | **<=**
        | **/=**
        | **+**
        | **-**
        | **&**
        | *****
        | **/**
        | **mod**
        | **rem**
        | ******

unary_op
        = **not** | **abs** | **-** | **+**

time
        = integer_literal unit

unit
        = **microsec**
        | **ms**
        | **sec**
        | **min**
        | **hours**

142

```
expression_list
      = expression {, expression}

expression
      = true
      | false
      | integer_literal
      | time          | real_literal
      | string_literal
      | id
      | type_name . id [( expression_list )]
      | ( expression )
      | initial_expression binary_op initial_expression
      | unary_op initial_expression

id
      = letter {alpha_numeric}

real_literal
      = integer_literal . integer_literal

integer_literal
      = digit {digit}

string_literal
      = " {char} "

char
      = any printable character except }

digit
      = 0 .. 9

letter
      = a .. z
      | A .. Z
      | _

alpha_numeric
      = letter
      | digit

text
      = {char}
```

APPENDIX E

Ada Implementation Code

On the following pages are contained the implementation code for the current version of the Change-Merge Tool. This tool used the PSDL Abstract Data Type developed n Ada by other members of the CAPS Research Team, as well as the PSDL Expander developed and implemented by Dr. Berzins. The code for these systems are not included in this dissertation.

The code contained in this appendix is broken up into different files. Each section of this appendix will contain a different file. All code was implemented in Ada and compiled using the Sun Ada Compiler Version 1.0.

1. merge_main_pkg

```
--------------------------------------------------------------------------
-- COMPONENT NAME   : PACKAGE MERGE_MAIN_PKG   ( merge_main_pkg_s.a )
-- USAGE            : Used to perform all of the housekeeping and interface
--                    between the CAPS interface and the change-merge system
--                    developed by Dave Dampier.
-- INPUT/OUTPUT     :
-- AUTHOR           : Jim Brockett
-- DATE OF CREATION : 28 NOVEMBER 1993
-- LANGUAGE USED    : Ada
-- COMPILER USED    : Sun Ada 1.0
-- PURPOSE          : Provides three functions used by the Merge Interface;
--                    merge, find_Base, and commit_merge.
-- FILES USED       :
-- NOTES            : This is the module to which the TAE interface code for
--                    the CAPS merge tool connects. Calls are made from the
--                    merge interface to this package. It is TBD whether or
--                    not the actual merge software is integrated into this
--                    package or put separately elsewhere. Either way will
--                    work. The purpose of this packaage is integration
--                    specification.
--------------------------------------------------------------------------
-- MODIFICATIONS
--------------------------------------------------------------------------
--     DATE         : 19 APRIL 1994
--     AUTHOR       : Dave Dampier
--     PURPOSE      : Completed Integration with Change_Merge_Pkg.
-- AFFECTED MODULES : All
--------------------------------------------------------------------------
```

```
with unix_prcs; use unix_prcs;
with unix_dirs; use unix_dirs;
with text_io; use text_io;
with a_strings; use a_strings;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with expander_pkg; use expander_pkg;
with change_merge_pkg; use change_merge_pkg;

package merge_main is

  procedure merge          (BASE_VERSION,
                            VERSION_A,
                            VERSION_B : in a_string;
                            RESULT    : in out a_string;
                            CONFLICT  : in out boolean);

  procedure find_Base      (VERSION_A,
                            VERSION_B    : in a_string;
                            BASE_VERSION : in out a_string;
                            ERROR        : in out boolean);

  procedure commit_merge  (BASE_VERSION,
                            VERSION_A,
                            VERSION_B : in a_string;
                            RESULT    : in out a_string);

end merge_main;
```

```
with unix; use unix;
with system; use system;

package body merge_main is

  prototype_path_error: exception;

procedure system_call(command : in string) is
        procedure system_C(command :address);
        pragma INTERFACE(C, system_C);
        pragma INTERFACE_NAME(system_C, "_system");
        temp : constant STRING := command&ASCII.NUL;
        error: integer;
begin
    system_C(TEMP'ADDRESS);
end system_call;
```

```
------------------------------------------------------------------------
-- Local function to extract the name of the prototype from the          '
-- version string. Raises PROTOTYPE_PATH_ERROR if an a_string without
-- the substring "/.caps/" is received as P.
------------------------------------------------------------------------


function name_of_prototype(P : a_string) return a_string is

  pname  : string(1..P.len);           -- to hold prototype name.
  index1 : integer := P.len;           -- to iterate through P.
  index2 : integer := 1;
  slash_not_found : boolean := true;

begin
  for i in 1..P.len loop
    pname(i) := ascii.nul;
  end loop;
  for i in 1..2 loop
    while slash_not_found loop
      index1 := index1 - 1;
      if index1 < 1
        then raise prototype_path_error;
      end if;
      if P.s(index1) = '/'
        then
          slash_not_found := false;
      end if;
    end loop;
    slash_not_found := true;
  end loop;
  index1 := index1 + 1;
  while slash_not_found loop
    if P.s(index1) = '/'
      then
        slash_not_found := false;
      else
        pname(index2) := P.s(index1);
        index1 := index1 + 1;
        index2 := index2 + 1;
    end if;
  end loop;
  return truncate(to_a(pname),index2);
end name_of_prototype;
```

```
    ------------------------------------------------------------------------
    -- Procedure merge reads in three prototypes and change-merges them,
    -- returning a file name holding the resultant prototype from the
    -- change-merge.
    ------------------------------------------------------------------------
    procedure merge (BASE_VERSION,
                     VERSION_A,
                     VERSION_B : in a_string;
                     RESULT    : in out a_string;
                     CONFLICT  : in out boolean) is


    PROTOTYPE_NAME : a_string;
    FILE_STRING    : a_string;
    BASEFILE       : file_type;        -- Used to hold expanded file.
    AFILE          : file_type;        --  "       "         "
    BFILE          : file_type;        --  "       "         "
    MERGEFILE      : file_type;        --  "       "         "
    BASE           : psdl_program;     -- Used to hold base program.
    OPA            : psdl_program;     -- Used to hold first modification.
    OPB            : psdl_program;     -- Used to hold second modification.
    MERGE          : psdl_program;     -- Used to hold merged program.
    TEMP           : status_code;

    begin

      -- reads in Base prototype and puts in ADT.
      put_line("change-merging prototypes");
      put_line("reading base version");
      PROTOTYPE_NAME := name_of_prototype(BASE_VERSION);
      system_call("merge.script -p "&BASE_VERSION.s&" "&
              PROTOTYPE_NAME.s&"> "&"/tmp/temp_base_file.psdl");
        -- builds single file input!
      open(BASEFILE, in_file, "/tmp/temp_base_file.psdl");
      assign(BASE,empty_psdl_program);
      get(BASEFILE,BASE);
      close(BASEFILE);
      system_call("rm /tmp/temp_base_file.psdl");
      expand(BASE);
```

```
    -- reads in first modified version of prototype and puts in ADT.
    put_line("reading 1st modified version");
    system_call("merge.script -p "&VERSION_A.s&" "&
            PROTOTYPE_NAME.s&"> "&"/tmp/temp_a_file.psdl");
      -- builds single file input!
    open(AFILE, in_file, "/tmp/temp_a_file.psdl");
    assign(OPA,empty_psdl_program);
    get(AFILE,OPA);
    close(AFILE);
    system_call("rm /tmp/temp_a_file.psdl");
    expand(OPA);
    -- reads in second modified version of prototype and puts in ADT.
    put_line("reading 2nd modified version");
    system_call("merge.script -p "&VERSION_B.s&" "&
            PROTOTYPE_NAME.s&"> "&"/tmp/temp_b_file.psdl");
      -- builds single file input!
    open(BFILE, in_file, "/tmp/temp_b_file.psdl");
    assign(OPB,empty_psdl_program);
    get(BFILE,OPB);
    close(BFILE);
    system_call("rm /tmp/temp_b_file.psdl");
    expand(OPB);
    -- puts result of performing the merge into the directory result.
    change_merge(BASE, OPA, OPB, MERGE, CONFLICT);
    temp := mkdir(result.s);
    split(result,PROTOTYPE_NAME,MERGE);

exception
  when use_error =>
    put_line(standard_error,
      "error: can't create output file. permission denied.");
  when syntax_error =>
    put_line(standard_error,
      " parsing aborted due to syntax error.");
  when semantic_error =>
    put_line(standard_error,
      " semantic error, parsing aborted.");
  when expander_pkg.no_root =>
    put_line(standard_error,
      " semantic error - no top level operator, expansion aborted.");
    put_line(standard_error,
      " check for recursive use of the prototype name in an expansion.");
```

150

```
when expander_pkg.multiple_roots =>
   put_line(standard_error,
    " semantic error - more than one top level operator,
      expansion aborted.");
   put_line(standard_error,
     " check for operators that are not used or");
   put_line(standard_error,
     " add an extra top-level operator that decomposes");
   put_line(standard_error,
     " into the current set of top-level components");
   put_line(standard_error,
     " if your design has several top-level components.");
--when undefined_component =>
--   put_line(standard_error,
--     " semantic error - an operator without a PSDL definition has
--   been used.");
when prototype_path_error =>
   put_line("from merge_main_pkg.merge");
   put_line(standard_error,
     " path to merge inputs provided by top-level interface was
       incorrect.");
when constraint_error =>
   put_line(standard_error,
     " constraint_error - merger not working properly.");
when numeric_error =>
   put_line(standard_error,
     " numeric_error - merger not working properly.");
when program_error =>
   put_line(standard_error,
     " program_error - merger not working properly.");
when storage_error =>
   put_line(standard_error,
     " storage_error - merger not working properly.");
when tasking_error =>
   put_line(standard_error,
     " tasking_error - merger not working properly.");
when others =>
   put_line(standard_error,
     " unexpected exception - merger not working properly.");

end merge;
```

```
procedure find_Base (VERSION_A,
                     VERSION_B    : in a_string;
                     BASE_VERSION : in out a_string;
                     ERROR        : in out boolean) is

begin
  text_io.put_line("this procedure is not yet implemented");
  BASE_VERSION := to_a("You must select a base version manually!");
  ERROR := true;
end find_Base;

procedure commit_merge (BASE_VERSION,
                        VERSION_A,
                        VERSION_B : in a_string;
                        RESULT    : in out a_string) is

  in_result   : a_string := copy(RESULT);
  temp_string : a_string;
  temp        : status_code;

function version_num(x : a_string) return a_string is

  vnum  : a_string;
  index : integer := x.len;

begin
  while x.s(index) /= '/' loop
     index := index - 1;
  end loop;
  vnum := to_a(x.s(index+1..x.len));
  return vnum;
end version_num;

begin
  temp_string := (in_result & to_a("-") & version_num(VERSION_A)
                 & to_a("_") & version_num(BASE_VERSION)
                 & to_a("_") & version_num(VERSION_B));
  temp := mkdir(temp_string.s);
  system_call("mv " & in_result.s & "/*.psdl " & temp_string.s);
  temp := rmdir(in_result.s);
  RESULT := copy(temp_string);
end commit_merge;
```

```
end merge_main;
```

2. change_merge_pkg

```
-------------------------------------------------------------------------
-- COMPONENT NAME    : PACKAGE CHANGE_MERGE_PKG ( change_merge_pkg_s.a )
-- USAGE            :
-- INPUT/OUTPUT     : BASE, A, B: in psdl_program
--                    MERGE: in out psdl_program
--                    CONFLICT: out boolean
-- AUTHOR           : Dave Dampier
-- DATE OF CREATION : 19 April 1994
-- LANGUAGE USED    : Ada
-- COMPILER USED    : Sun Ada 1.0
-- PURPOSE          : Contains the procedure which performs the change-merge
--                    operation on PSDL programs.
-- FILES USED       : psdl_type_s.a, psdl_ct_s.a, psdl_prog_s.a,
--                    psdl_graph_s.a, prototype_dependency_graph_pkg_s.a,
--                    proto_spec_merge_pkg_s.a, proto_impl_merge_pkg_s.a,
--                    exp_s.a.
-- NOTES            :
-------------------------------------------------------------------------

with a_strings; use a_strings;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_graph_pkg; use psdl_graph_pkg;
with prototype_dependency_graph_pkg; use prototype_dependency_graph_pkg;
with proto_spec_merge_pkg; use proto_spec_merge_pkg;
with proto_impl_merge_pkg; use proto_impl_merge_pkg;
with text_io; use text_io;
with expression_pkg; use expression_pkg;

package change_merge_pkg is
```

153

```
--------------------------------------------------------------------------
-- This function performs the change_merge operation on PSDL prototypes. --
-- Given three prototypes, BASE, A and B, the function creates            --
-- prototype dependency graphs for the three prototypes, and using        --
-- prototype slicing, it identifies the preserved part of the base        --
-- in all three versions, and the parts of the changed versions which     --
-- are different from the base.  It then combines the three pieces into    --
-- a merged graph.  If the graph correctly represents the semantic merge  --
-- of the three versions, and there are no conflicts, then the merged     --
-- prototype is reconstructed from the merged graph. In the case of a     --
-- conflict, the exception "merge_conflict" is raised.                    --
--------------------------------------------------------------------------


procedure change_merge(BASE, A, B: in psdl_program;
                       MERGE: in out psdl_program;
                       CONFLICT: out boolean);

procedure build_prototype(P: in out psdl_component;
                          G: in prototype_dependency_graph);

end change_merge_pkg;
```

154

```
package body change_merge_pkg is


----------------------------------------------------------------------------
--  This function performs the change_merge operation on PSDL prototypes.
--  Given three prototypes, BASE, A and B, the function creates
--  prototype dependency graphs for the three prototypes, and using
--  prototype slicing, it identifies the preserved part of the base
--  in all three versions, and the parts of the changed versions which
--  are different from the base.  It then combines the three pieces into a
--  merged graph.  If the graph correctly represents the semantic merge
--  of the three versions, and there are no conflicts, then the merged
--  prototype is reconstructed from the merged graph.
----------------------------------------------------------------------------

procedure change_merge(BASE, A, B: in psdl_program;
                       MERGE: in out psdl_program;
                       CONFLICT: out boolean) is

  BASEHOLD, AHOLD, BHOLD: psdl_program := empty_psdl_program;
  BASETYPE, ATYPE, BTYPE: psdl_program := empty_psdl_program;
  ATOMIC_COMP: atomic_operator;
  BASECOMP, ACOMP, BCOMP, MERGECOMP: composite_operator;
  GBASE, GA, GB, GM, PP, APA, APB: prototype_dependency_graph;
  BASESTREAMS, ASTREAMS, BSTREAMS, MERGESTREAMS: type_declaration;
  MERGESTATES: type_declaration;
  MERGEINIT: init_map := empty_init_map;
  MERGEEXCEPTIONS, MERGEKEYWORDS: id_set;
  MERGEMET: millisec := 0;
  MERGE_INF_DESC, MERGE_AX: text;
  BASETRIG, ATRIG, BTRIG, MERGETRIG: trigger_map := empty_trigger_map;
  BASEEG, AEG, BEG, MERGEEG: exec_guard_map := empty_exec_guard_map;
  BASEOG, AOG, BOG, MERGEOG: out_guard_map := empty_out_guard_map;
  BASEET, AET, BET, MERGEET: excep_trigger_map := empty_excep_trigger_map;
  BASETO, ATO, BTO, MERGETO: timer_op_map := empty_timer_op_map;
  BASEPER, APER, BPER, MERGEPER: timing_map := empty_timing_map;
  BASEFW, AFW, BFW, MERGEFW: timing_map := empty_timing_map;
```

155

```
    BASEMCP, AMCP, BMCP, MERGEMCP: timing_map := empty_timing_map;
    BASEMRT, AMRT, BMRT, MERGEMRT: timing_map := empty_timing_map;
    BASEDESC, ADESC, BDESC, MERGEDESC: text;
    MERGEID: psdl_id;
    BASETIMERS, ATIMERS, BTIMERS, MERGETIMERS, V: id_set;
    tempexpression: expression;
    tempoutid: output_id;
    tempexid: excep_id;
    conflict_free_a, conflict_free_b: boolean := true;

  begin

    conflict := false;
-----------------------------------------------------------------------------
-- This section of code is used to extract the psdl components from each
-- of the three programs. It assigns the parent composite operator to its
-- own component variable, and it assigns the atomic operators to holding
-- components, so they can be retrieved later.
-----------------------------------------------------------------------------


------------------------------
-- BASE
------------------------------

    for id:psdl_id,c:psdl_component in psdl_program_map_pkg.scan(BASE) loop
      if component_category(c) = psdl_type
        then
          bind(id,c,BASETYPE);
        else
          if component_granularity(c) = composite
            then
              BASECOMP := c;
            else
              bind(id, c, BASEHOLD);
          end if;
      end if;
    end loop;
```

```
-----------------------------
-- A
-----------------------------

    for id:psdl_id,c:psdl_component in psdl_program_map_pkg.scan(A) loop
      if component_category(c) = psdl_type
        then
          bind(id,c,ATYPE);
        else
          if component_granularity(c) = composite
            then
              ACOMP := c;
            else
              bind(id, c, AHOLD);
          end if;
      end if;
    end loop;

-----------------------------
-- B
-----------------------------

    for id:psdl_id,c:psdl_component in psdl_program_map_pkg.scan(B) loop
      if component_category(c) = psdl_type
        then
          bind(id,c,BTYPE);
        else
          if component_granularity(c) = composite
            then
              BCOMP := c;
            else
              bind(id, c, BHOLD);
          end if;
      end if;
    end loop;
```

```
--------------------------------------------------------------------------
-- Create the Merged Specification
--------------------------------------------------------------------------


------------------------------------------
-- Merge the states
------------------------------------------


    merge_states(MERGESTATES,states(BASECOMP),states(ACOMP),states(BCOMP),
                 MERGEINIT, get_init_map(BASECOMP), get_init_map(ACOMP),
                 get_init_map(BCOMP));


------------------------------------------
-- Merge the Exceptions
------------------------------------------


    assign(MERGEEXCEPTIONS, merge_id_sets(exceptions(BASECOMP),
                                exceptions(ACOMP), exceptions(BCOMP)));


------------------------------------------
-- Merge the Keywords
------------------------------------------


    assign(MERGEKEYWORDS, merge_id_sets(keywords(BASECOMP),
                                keywords(ACOMP), keywords(BCOMP)));


------------------------------------------
-- Merge the Informal Description
------------------------------------------


    MERGE_INF_DESC := merge_text(informal_description(BASECOMP),
                                 informal_description(ACOMP),
                                 informal_description(BCOMP));


------------------------------------------
-- Merge the Formal Description
------------------------------------------


    MERGE_AX := merge_text(axioms(BASECOMP),
                           axioms(ACOMP),
                           axioms(BCOMP));
```

158

```
-----------------------------------------
--  Merge the Maximum Execution Times
-----------------------------------------

    MERGEMET := merge_met(specified_maximum_execution_time(BASECOMP),
                          specified_maximum_execution_time(ACOMP),
                          specified_maximum_execution_time(BCOMP));

-------------------------------------------------------------------------------
--  Merge the Implementation
-------------------------------------------------------------------------------


-----------------------------------------
--  Extract the prototype dependency
--  graphs from the psdl components.
-----------------------------------------

    assign(GBASE, build_PDG(BASECOMP));
    assign(GA, build_PDG(ACOMP));
    assign(GB, build_PDG(BCOMP));

-----------------------------------------
--  Create the Preserved Part
-----------------------------------------

    assign(PP, preserved_part(GBASE, GA, GB));

-----------------------------------------
--  Create the Affected Parts of each
--  modification graph.
-----------------------------------------

    -- put_line("Affected Part: A");
    assign(APA, affected_part(GA, GBASE)); -- First Modification
    -- put_line("Affected Part: B");
    assign(APB, affected_part(GB, GBASE)); -- Second Modification
```

```
----------------------------------------
-- Create the Merged Graph using the
-- Preserved Part of the Base and
-- the Affected Parts of both
-- modifications.
----------------------------------------

    assign(GM, graph_merge(PP, APA, APB));

----------------------------------------
-- Merge the streams
----------------------------------------

    assign(BASESTREAMS, streams(BASECOMP));
    assign(ASTREAMS, streams(ACOMP));
    assign(BSTREAMS, streams(BCOMP));
    assign(MERGESTREAMS, merge_streams(BASESTREAMS, ASTREAMS, BSTREAMS));

----------------------------------------
-- Merge the timers
----------------------------------------

    assign(BASETIMERS, timers(BASECOMP));
    assign(ATIMERS, timers(ACOMP));
    assign(BTIMERS, timers(BCOMP));
    assign(MERGETIMERS, merge_timers(BASETIMERS, ATIMERS, BTIMERS));

----------------------------------------
-- Merge the triggers
----------------------------------------

    for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
      if not eq(id, EXT)
        then
          bind(id, get_trigger(id, BASECOMP), BASETRIG);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
      if not eq(id, EXT)
        then
          bind(id, get_trigger(id, ACOMP), ATRIG);
      end if;
    end loop;
```

```
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, get_trigger(id, BCOMP), BTRIG);
      end if;
    end loop;
    assign(MERGETRIG, merge_trigger_maps(vertices(GM),
                                         BASETRIG, ATRIG, BTRIG));

-----------------------------------------
-- Merge the execution guards
-----------------------------------------

    for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
      if not eq(id, EXT)
        then
          bind(id, execution_guard(id, BASECOMP), BASEEG);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
      if not eq(id, EXT)
        then
          bind(id, execution_guard(id, ACOMP), AEG);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, execution_guard(id, BCOMP), BEG);
      end if;
    end loop;
    assign(MERGEEG, merge_exec_guard_maps(vertices(GM),
                                          BASEEG, AEG, BEG));
```

```
    for e: edge in edge_set_pkg.scan(edges(GBASE)) loop
      assign(tempexpression,output_guard(e.x,e.stream_name,BASECOMP));
      if not(tempexpression = true_expression)
        then
          tempoutid.op := copy(e.x);
          tempoutid.stream := copy(e.stream_name);
          bind(tempoutid, tempexpression, BASEOG);
        end if;
    end loop;
    for e: edge in edge_set_pkg.scan(edges(GA)) loop
      assign(tempexpression,output_guard(e.x,e.stream_name,ACOMP));
      if not(tempexpression = true_expression)
        then
          tempoutid.op := copy(e.x);
          tempoutid.stream := copy(e.stream_name);
          bind(tempoutid, tempexpression, AOG);
        end if;
    end loop;
    for e: edge in edge_set_pkg.scan(edges(GB)) loop
      assign(tempexpression,output_guard(e.x,e.stream_name,BCOMP));
      if not(tempexpression = true_expression)
        then
          tempoutid.op := copy(e.x);
          tempoutid.stream := copy(e.stream_name);
          bind(tempoutid, tempexpression, BOG);
        end if;
    end loop;
    assign(MERGEOG, merge_output_guard_maps(BASEOG,AOG,BOG));
```

-- Merge the exception triggers

```
for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
  if not eq(id, EXT)
    then
      for e: psdl_id in id_set_pkg.scan(exceptions(BASECOMP)) loop
        assign(tempexpression, exception_trigger(id,e,BASECOMP));
        if not eq(tempexpression, false_expression)
          then
            tempexid.op := copy(id);
            tempexid.excep := copy(e);
            bind(tempexid, tempexpression, BASEET);
        end if;
      end loop;
  end if;
end loop;
for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
  if not eq(id, EXT)
    then
      for e: psdl_id in id_set_pkg.scan(exceptions(ACOMP)) loop
        assign(tempexpression, exception_trigger(id,e,ACOMP));
        if not eq(tempexpression, false_expression)
          then
            tempexid.op := copy(id);
            tempexid.excep := copy(e);
            bind(tempexid, tempexpression, AET);
        end if;
      end loop;
  end if;
end loop;
for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
  if not eq(id, EXT)
    then
      for e: psdl_id in id_set_pkg.scan(exceptions(BCOMP)) loop
        assign(tempexpression, exception_trigger(id,e,BCOMP));
        if not eq(tempexpression, false_expression)
          then
            tempexid.op := copy(id);
            tempexid.excep := copy(e);
            bind(tempexid, tempexpression, BET);
        end if;
      end loop;
  end if;
end loop;
assign(MERGEET,(merge_excep_trigger_maps(BASEET, AET, BET)));
```

163

```
------------------------------------------
-- Merge the timer operations
------------------------------------------

   for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
     if not eq(id, EXT)
       then
         bind(id, timer_operations(id, BASECOMP), BASETO);
     end if;
   end loop;
   for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
     if not eq(id, EXT)
       then
         bind(id, timer_operations(id, ACOMP), ATO);
     end if;
   end loop;
   for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
     if not eq(id, EXT)
       then
         bind(id, timer_operations(id, BCOMP), BTO);
     end if;
   end loop;
   assign(MERGETO, merge_timer_op_maps(vertices(GM), BASETO, ATO, BTO));

------------------------------------------
-- Merge the periods
------------------------------------------

   for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
     if not eq(id, EXT)
       then
         bind(id, period(id, BASECOMP), BASEPER);
     end if;
   end loop;
   for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
     if not eq(id, EXT)
       then
         bind(id, period(id, ACOMP), APER);
     end if;
   end loop;
```

164

```
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, period(id, BCOMP), BPER);
      end if;
    end loop;
    assign(MERGEPER, merge_period(BASEPER, APER, BPER));

-----------------------------------------
-- Merge the finish_withins
-----------------------------------------

    for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
      if not eq(id, EXT)
        then
          bind(id, finish_within(id, BASECOMP), BASEFW);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
      if not eq(id, EXT)
        then
          bind(id, finish_within(id, ACOMP), AFW);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, finish_within(id, BCOMP), BFW);
      end if;
    end loop;
    assign(MERGEFW, merge_fw_or_mrt(BASEFW, AFW, BFW));

-----------------------------------------
-- Merge the max response times
-----------------------------------------

    for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
      if not eq(id, EXT)
        then
          bind(id, maximum_response_time(id, BASECOMP), BASEMRT);
      end if;
    end loop;
```

165

```
    for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
      if not eq(id, EXT)
        then
          bind(id, maximum_response_time(id, ACOMP), AMRT);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, maximum_response_time(id, BCOMP), BMRT);
      end if;
    end loop;
    assign(MERGEMRT, merge_fw_or_mrt(BASEMRT, AMRT, BMRT));

-----------------------------------------
-- Merge the minimum calling periods
-----------------------------------------

    for id: psdl_id in id_set_pkg.scan(vertices(GBASE)) loop
      if not eq(id, EXT)
        then
          bind(id, minimum_calling_period(id, BASECOMP), BASEMCP);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GA)) loop
      if not eq(id, EXT)
        then
          bind(id, minimum_calling_period(id, ACOMP), AMCP);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(vertices(GB)) loop
      if not eq(id, EXT)
        then
          bind(id, minimum_calling_period(id, BCOMP), BMCP);
      end if;
    end loop;
    assign(MERGEMCP, merge_min_call_per(BASEMCP, AMCP, BMCP));
```

```
-----------------------------------------
-- Merge the implementation descriptions
-----------------------------------------

    BASEDESC := implementation_description(BASECOMP);
    ADESC := implementation_description(ACOMP);
    BDESC := implementation_description(BCOMP);
    MERGEDESC := merge_implementation_description(BASEDESC, ADESC, BDESC);

-----------------------------------------------------------------------------
-- Construct the merged program.
-----------------------------------------------------------------------------

    MERGEID := copy(name(BASECOMP));
    MERGECOMP := make_composite_operator(MERGEID,
                              keywords => MERGEKEYWORDS,
                              informal_description => MERGE_INF_DESC,
                              axioms => MERGE_AX,
                              state => MERGESTATES,
                              initialization_map => MERGEINIT,
                              exceptions => MERGEEXCEPTIONS,
                              specified_met => MERGEMET,
                              streams => MERGESTREAMS,
                              timers => MERGETIMERS,
                              trigger => MERGETRIG,
                              exec_guard => MERGEEG,
                              out_guard => MERGEOG,
                              excep_trigger => MERGEET,
                              timer_op => MERGETO,
                              per => MERGEPER,
                              fw => MERGEFW,
                              mcp => MERGEMCP,
                              mrt => MERGEMRT,
                              impl_desc => MERGEDESC);
```

167

```
--------------------------------------------------------------------------
--  Compare the the Merged Graph with the Graph of each modification by
--  comparing the slices of each with respect to their affected parts.
--  If the slices are the same, then the merged graph is correct and the
--  program can be rebuilt. Otherwise, there is a conflict that must be
--  resolved.
--------------------------------------------------------------------------

    build_prototype(MERGECOMP, GM);
    conflict_free_a := compare_graphs(GM, GA, APA);
    conflict_free_b := compare_graphs(GM, GB, APB);
    if not conflict_free_a
      then
        put_line("Conflict found in Version_A");
        conflict := true;
    end if;
    if not conflict_free_b
      then
        put_line("Conflict found in Version_B");
        conflict := true;
    end if;


--------------------------------------------------------------------------
--  Return the Merged Program.
--------------------------------------------------------------------------

    bind(MERGEID,MERGECOMP,MERGE);

    assign(V, vertices(PP));
    for id: psdl_id in id_set_pkg.scan(V) loop
      bind(id,fetch(BASEHOLD,id),MERGE);
    end loop;
    assign(V, vertices(APA));
    for id: psdl_id in id_set_pkg.scan(V) loop
      if not member(id,MERGE)
        then
          bind(id,fetch(AHOLD,id),MERGE);
      end if;
    end loop;
    assign(V, vertices(APB));
```

```
   for id: psdl_id in id_set_pkg.scan(V) loop
     if not member(id,MERGE)
       then
         bind(id,fetch(BHOLD,id),MERGE);
     end if;
   end loop;

end change_merge;

----------------------------------------------------------------------------
-- This procedure is used to build the merged prototype when the change-
-- merge operation is successful.
----------------------------------------------------------------------------

procedure build_prototype(P: in out psdl_component;
                          G: in prototype_dependency_graph) is

  A: psdl_graph;

  begin
    assign(A, psdl_graph(G));
    remove_vertex(EXT, A);
    set_graph(A,P);
  end build_prototype;

end change_merge_pkg;
```

3. proto_spec_merge_pkg

```
-------------------------------------------------------------------------
-- COMPONENT NAME  : PACKAGE PROTO_SPEC_MERGE_PKG(proto_spec_merge_pkg_s.a)
-- AUTHOR          : Dave Dampier
-- DATE OF CREATION : 19 April 1994
-- LANGUAGE USED   : Ada
-- COMPILER  USED  : Sun Ada 1.0
-- PURPOSE         : This package provides specifications for the functions
--                   used to perform change-merges on psdl operator
--                   specifications.
-- FILES USED      : psdl_ct_s.a, psdl_ct_b.a, psdl_type_s.a, psdl_type_b.a,
--                   set_s.a, set_b.a, map_s.a, map_b.a, exp_s.a, exp_b.a.
-------------------------------------------------------------------------

with system;
with generic_map_pkg;
with generic_set_pkg;
with TEXT_IO;                       use TEXT_IO;
with a_strings;                     use a_strings;
with psdl_concrete_type_pkg;        use psdl_concrete_type_pkg;
with psdl_component_pkg;            use psdl_component_pkg;
with expression_pkg;               use expression_pkg;

package proto_spec_merge_pkg is

function MERGE_SEQUENCES(BASE, A, B: type_declaration)
        return type_declaration;

procedure MERGE_STATES(MERGE: in out type_declaration;
                       BASE, A, B: in type_declaration;
                       MERGEINIT: in out init_map;
                       BASEINIT, AINIT, BINIT: in init_map);

function MERGE_MET(BASE, A, B: millisec) return millisec;

function MERGE_ID_SETS(BASE, A, B: id_set) return id_set;

function MERGE_TEXT(BASE, A, B: text) return text;

end proto_spec_merge_pkg;
```

170

```
package body proto_spec_merge_pkg is

function merge_sequences(BASE, A, B: type_declaration)
        return type_declaration is

  MERGE : type_declaration;
  TOP   : constant psdl_id := to_a("TOP");

  begin
    assign(MERGE, empty_type_declaration);
    if   equal(BASE, A)
      then if equal(BASE, B)
        then assign(MERGE, BASE);
        else assign(MERGE, B);
      end if;
      else if equal(BASE, B)
        then assign(MERGE, A);
        else if equal(A, B)
          then assign(MERGE, A);
          else bind(TOP, null_type, MERGE);
        end if;
      end if;
    end if;
    return MERGE;
  end merge_sequences;

function merge_types(t_base, t_a, t_b : type_name) return type_name is

  begin
    if equal(t_base, t_a)
      then
        if equal(t_base, t_b)
          then
            return(t_base);
          else
            return(t_b);
        end if;
```

```
        else
          if equal(t_base, t_b)
            then
              return(t_a);
            else
              if equal(t_a, t_b)
                then
                  return(t_a);
                else
                  return null_type;
              end if;
          end if;
      end if;
  end merge_types;

procedure merge_states(MERGE: in out type_declaration;
                       BASE, A, B: in type_declaration;
                       MERGEINIT: in out init_map;
                       BASEINIT, AINIT, BINIT: in init_map) is

  init_value : expression;
  base_type, a_type, b_type : type_name;

  begin
    assign(init_value, empty_expression);
    assign(MERGE, empty_type_declaration);
    for id: psdl_id, t: type_name in type_declaration_pkg.scan(BASE) loop
      if member(id, A) and member(id, B)
        then
          a_type := type_declaration_pkg.fetch(A,id);
          b_type := type_declaration_pkg.fetch(B,id);
          bind(id, merge_types(t, a_type, b_type), MERGE);
          assign(init_value, init_map_pkg.fetch(BASEINIT,id));
          if eq(init_value,init_map_pkg.fetch(AINIT,id))
            then
              if eq(init_value,init_map_pkg.fetch(BINIT,id))
                then
                  bind(id,init_value,MERGEINIT);
                else
                  bind(id,init_map_pkg.fetch(BINIT,id),MERGEINIT);
              end if;
```

172

```
        else
           if eq(init_value,init_map_pkg.fetch(BINIT,id))
             then
                bind(id,init_map_pkg.fetch(AINIT,id),MERGEINIT);
             else
               if eq(init_map_pkg.fetch(AINIT,id),
                      init_map_pkg.fetch(BINIT,id))
                 then
                    bind(id,init_map_pkg.fetch(AINIT,id),MERGEINIT);
                 else
                    bind(id,conflict_expression,MERGEINIT);
               end if;
           end if;
        end if;
   end if;
end loop;
for id: psdl_id, t: type_name in type_declaration_pkg.scan(A) loop
   if not member(id, BASE) and member(id, B)
     then
        base_type := null_type;
        b_type := type_declaration_pkg.fetch(B,id);
        bind(id, merge_types(base_type, t, b_type), MERGE);
        assign(init_value, init_map_pkg.fetch(AINIT,id));
        if eq(init_value,init_map_pkg.fetch(BINIT,id))
          then
             bind(id,init_value,MERGEINIT);
          else
             bind(id,conflict_expression,MERGEINIT);
        end if;
   end if;
end loop;
```

173

```
    for id: psdl_id, t: type_name in type_declaration_pkg.scan(B) loop
      if not member(id, BASE) and member(id, A)
        then
          base_type := null_type;
          a_type := type_declaration_pkg.fetch(A,id);
          bind(id, merge_types(base_type, a_type, t), MERGE);
          assign(init_value, init_map_pkg.fetch(BINIT,id));
          if eq(init_value,init_map_pkg.fetch(AINIT,id))
            then
              bind(id,init_value,MERGEINIT);
            else
              bind(id,conflict_expression,MERGEINIT);
          end if;
      end if;
    end loop;
  end merge_states;

function merge_met(BASE, A, B: millisec) return millisec is
  A_DIFF_BASE, B_DIFF_BASE, A_INT_B: millisec;
  begin
    if A <= B
      then A_INT_B := B;
      else A_INT_B := A;
    end if;
    if BASE <= A
      then A_DIFF_BASE := system.max_int;
      else A_DIFF_BASE := A;
    end if;
    if BASE <= B
      then B_DIFF_BASE := system.max_int;
      else B_DIFF_BASE := B;
    end if;
    if A_DIFF_BASE <= A_INT_B
      then if A_DIFF_BASE <= B_DIFF_BASE
        then return A_DIFF_BASE;
        else return B_DIFF_BASE;
      end if;
      else if A_INT_B <= B_DIFF_BASE
        then return A_INT_B;
        else return B_DIFF_BASE;
      end if;
    end if;
  end merge_met;
```

174

```
function merge_id_sets(BASE, A, B: id_set) return id_set is

  A_DIFF_BASE, B_DIFF_BASE, MERGE: id_set;

  begin
    assign(A_DIFF_BASE, empty_id_set);
    assign(B_DIFF_BASE, empty_id_set);
    assign(MERGE, empty_id_set);
    difference(A, BASE, A_DIFF_BASE);
    difference(B, BASE, B_DIFF_BASE);
    for id: psdl_id in id_set_pkg.scan(A) loop
      if member(id, B)
        then
          add(id, MERGE);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(A_DIFF_BASE) loop
      if not member(id, MERGE)
        then
          add(id, MERGE);
      end if;
    end loop;
    for id: psdl_id in id_set_pkg.scan(B_DIFF_BASE) loop
      if not member(id, MERGE)
        then
          add(id, MERGE);
      end if;
    end loop;
    return MERGE;
  end merge_id_sets;
```

```
function merge_text(BASE, A, B: text) return text is

  begin
    if eq(BASE, empty_text) and eq(A, empty_text) and eq(B, empty_text)
      then return empty_text;
      else if eq(BASE, A)
        then if not eq(BASE, B)
          then return B;
          else return BASE;
        end if;
        else if eq(BASE, B)
          then return A;
          else if eq(A, B)
            then return A;
            else return to_a("**Text Conflict**");
          end if;
        end if;
      end if;
    end if;
  end merge_text;

end proto_spec_merge_pkg;
```

4. proto_impl_merge_pkg

```
-------------------------------------------------------------------------
-- COMPONENT NAME  : PACKAGE PROTO_IMPL_MERGE_PKG(proto_impl_merge_pkg_s.a)
-- USAGE           : Used to perform change-merging on PSDL program
--                   implementations.
-- AUTHOR          : David A. Dampier
-- DATE OF CREATION : 19 April 1994
-- LANGUAGE USED   : Ada
-- COMPILER USED   : Sun Ada 1.0
-- PURPOSE         : Provides specifications for the functions necessary
--                   to merge PSDL Implementations.
-------------------------------------------------------------------------
with system;
with TEXT_IO; use TEXT_IO;
with a_strings; use a_strings;
with generic_map_pkg;
with generic_set_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;
with proto_spec_merge_pkg; use proto_spec_merge_pkg;
with expression_pkg; use expression_pkg;

package proto_impl_merge_pkg is

function merge_streams(BASE, A, B: type_declaration)
                                            return type_declaration;

function merge_timers(BASE, A, B: id_set) return id_set
   renames proto_spec_merge_pkg.merge_id_sets;

function merge_trigger_maps(VERTS: id_set; BASE, A, B: trigger_map)
                                            return trigger_map;

function merge_exec_guard_maps(VERTS: id_set; BASE, A, B: exec_guard_map)
                                            return exec_guard_map;
```

177

```
function merge_output_guard_maps(BASE, A, B: out_guard_map)
                                            return out_guard_map;

function merge_excep_trigger_maps(BASE, A, B: excep_trigger_map)
                                            return excep_trigger_map;

function merge_timer_op_maps(VERTS: id_set; BASE, A, B: timer_op_map)
                                            return timer_op_map;

function merge_period(BASE, A, B: timing_map) return timing_map;

function merge_fw_or_mrt(BASE, A, B: timing_map) return timing_map;

function merge_min_call_per(BASE, A, B: timing_map) return timing_map;

function merge_implementation_description(BASE, A, B: text) return text
   renames proto_spec_merge_pkg.merge_text;

end proto_impl_merge_pkg;
```

```
package body proto_impl_merge_pkg is

function merge_types(t_base, t_a, t_b : type_name) return type_name is

  begin
    if equal(t_base, t_a)
      then
        if equal(t_base, t_b)
          then
            return(t_base);
          else
            return(t_b);
          end if;
      else
        if equal(t_base, t_b)
          then
            return(t_a);
          else
            if equal(t_a, t_b)
              then
                return(t_a);
              else
                return null_type;
            end if;
        end if;
    end if;
  end merge_types;

function merge_streams(BASE, A, B: type_declaration)
          return type_declaration is
  MERGE: type_declaration;
  base_type, a_type, b_type : type_name;

  begin
    assign(MERGE, empty_type_declaration);
    for id: psdl_id, t: type_name in type_declaration_pkg.scan(BASE) loop
      if member(id, A) and member(id, B)
        then
          a_type := type_declaration_pkg.fetch(A,id);
          b_type := type_declaration_pkg.fetch(B,id);
          bind(id, merge_types(t, a_type, b_type), MERGE);
      end if;
    end loop;
```

179

```
    for id: psdl_id, t: type_name in type_declaration_pkg.scan(A) loop
      if not member(id, BASE) and member(id, B)
        then
          base_type := null_type;
          b_type := type_declaration_pkg.fetch(B,id);
          bind(id,merge_types(base_type, t, b_type), MERGE);
      end if;
    end loop;
    for id: psdl_id, t: type_name in type_declaration_pkg.scan(B) loop
      if not member(id, BASE) and member(id, A)
        then
          base_type := null_type;
          a_type := type_declaration_pkg.fetch(A,id);
          bind(id,merge_types(base_type,a_type, t), MERGE);
      end if;
    end loop;
    return MERGE;
  end merge_streams;

function merge_triggers(BASE, A, B: trigger) return trigger is

  MERGE: trigger;
  conflict_trigger : trigger := (tt => by_all,
                                 streams => id_set_pkg.add(to_a("TOP"),
                                                           id_set_pkg.empty));

begin
  if eq(BASE, A)
    then
      if eq(BASE, B)
        then
          MERGE.tt := BASE.tt;
          assign(MERGE.streams,
                 merge_id_sets(BASE.streams, A.streams, B.streams));
          return MERGE;
        else
          return B;
      end if;
```

180

```
    else
      if eq(BASE, B)
        then
          return A;
        else
          if eq(BASE, B)
            then
              return A;
            else
              return conflict_trigger;
          end if;
      end if;
  end if;
end merge_triggers;

function merge_trigger_maps(VERTS: id_set; BASE, A, B: trigger_map)
                                                return trigger_map is

  MERGE: trigger_map;
  base_trig, a_trig, b_trig, merge_trig : trigger;

  begin
    assign(MERGE, empty_trigger_map);
    for id : psdl_id in id_set_pkg.scan(VERTS) loop
      base_trig := fetch(BASE,id);
      a_trig := fetch(A,id);
      b_trig := fetch(B,id);
      merge_trig := merge_triggers(base_trig, a_trig, b_trig);
      bind(id, merge_trig, MERGE);
    end loop;
    return MERGE;
  end merge_trigger_maps;

function merge_expressions(BASE, A, B :expression) return expression is

  local_base : expression;
  local_a    : expression;
  local_b    : expression;
  conflict_expression : constant expression :=
                          create_identifier(to_a("**CONFLICT**"));
```

```
   begin
     assign(local_base, BASE);
     assign(local_a, A);
     assign(local_b, B);
     if eq(local_BASE, local_a)
       then
         if eq(local_BASE, local_B)
           then
             return(local_BASE);
           else
             return(local_B);
         end if;
       else
         if eq(local_BASE, local_B)
           then
             return(local_A);
           else
             if eq(local_A, local_B)
               then
                 return(local_A);
               else
                 return conflict_expression;
             end if;
         end if;
     end if;
   end merge_expressions;

function merge_exec_guard_maps(VERTS: id_set; BASE, A, B: exec_guard_map)
                                              return exec_guard_map is

 MERGE: exec_guard_map;
 base_eg, a_eg, b_eg, merge_eg : expression;

 begin
   assign(MERGE, empty_exec_guard_map);
   for id : psdl_id in id_set_pkg.scan(VERTS) loop
     assign(base_eg, fetch(BASE,id));
     assign(a_eg, fetch(A,id));
     assign(b_eg, fetch(B,id));
     assign(merge_eg, merge_expressions(base_eg, a_eg, b_eg));
     bind(id, merge_eg, MERGE);
   end loop;
   return MERGE;
 end merge_exec_guard_maps;
```

```
function merge_output_guard_maps(BASE, A, B: out_guard_map)
                                          return out_guard_map is

  MERGE: out_guard_map := empty_out_guard_map;
  base_og, a_og, b_og, merge_og : expression;

  begin
    for id : output_id, e: expression in out_guard_map_pkg.scan(BASE) loop
      if member(id,A) and member(id,B)
        then
          assign(a_og, fetch(A,id));
          assign(b_og, fetch(B,id));
          assign(merge_og, merge_expressions(e, a_og, b_og));
          bind(id, merge_og, MERGE);
      end if;
    end loop;
    for id : output_id, e : expression in out_guard_map_pkg.scan(A) loop
      if not member(id, MERGE)
        then
          if member(id, B)
            then
              assign(base_og, empty_expression);
              assign(b_og, fetch(B,id));
              assign(merge_og, merge_expressions(base_og, e, b_og));
              bind(id, merge_og, MERGE);
            else
              if not member(id, BASE)
                then
                  bind(id, e, MERGE);
              end if;
          end if;
      end if;
    end loop;
    for id : output_id, e : expression in out_guard_map_pkg.scan(B) loop
      if not member(id, MERGE) and not member(id, BASE)
        then
          bind(id, e, MERGE);
      end if;
    end loop;
    return MERGE;
  end merge_output_guard_maps;
```

```
function merge_excep_trigger_maps(BASE, A, B: excep_trigger_map)
                                  return excep_trigger_map is

  MERGE: excep_trigger_map;
  base_et, a_et, b_et, merge_et : expression;

begin
  assign(MERGE, empty_excep_trigger_map);
  for id:excep_id,e:expression in excep_trigger_map_pkg.scan(BASE) loop
    if member(id,A) and member(id,B)
      then
        assign(a_et, fetch(A,id));
        assign(b_et, fetch(B,id));
        assign(merge_et, merge_expressions(e, a_et, b_et));
        bind(id, merge_et, MERGE);
      end if;
  end loop;
  for id:excep_id,e:expression in excep_trigger_map_pkg.scan(A) loop
    if not member(id, MERGE)
      then
        if member(id, B)
          then
            assign(base_et, empty_expression);
            assign(b_et, fetch(B,id));
            assign(merge_et, merge_expressions(base_et, e, b_et));
            bind(id, merge_et, MERGE);
          else
            if not member(id, BASE)
              then
                bind(id, e, MERGE);
              end if;
          end if;
      end if;
  end loop;
  for id:excep_id,e:expression in excep_trigger_map_pkg.scan(B) loop
    if not member(id, MERGE) and not member(id, BASE)
      then
        bind(id, e, MERGE);
      end if;
  end loop;
  return MERGE;
end merge_excep_trigger_maps;
```

```
function merge_timer_op_sets(BASE,A,B:timer_op_set) return timer_op_set is

  MERGE : timer_op_set;

begin
  for t_op : timer_op in timer_op_set_pkg.scan(BASE) loop
    if member(t_op,A)
      then
        if member(t_op,B)
          then
            add(t_op,MERGE);
        end if;
    end if;
  end loop;
  for t_op : timer_op in timer_op_set_pkg.scan(A) loop
    if not member(t_op,MERGE)
      then
        if member(t_op,B)
          then
            add(t_op,MERGE);
        end if;
    end if;
  end loop;
  for t_op : timer_op in timer_op_set_pkg.scan(B) loop
    if not member(t_op,MERGE)
      then
        if member(t_op,A)
          then
            add(t_op,MERGE);
        end if;
    end if;
  end loop;
  return MERGE;
end merge_timer_op_sets;
```

```
function merge_timer_op_maps(VERTS: id_set; BASE, A, B: timer_op_map)
                                              return timer_op_map is·

  MERGE: timer_op_map;
  base_set, a_set, b_set, merge_set : timer_op_set;

  begin
    assign(MERGE, empty_timer_op_map);
    for id : psdl_id in id_set_pkg.scan(VERTS) loop
      assign(base_set, fetch(BASE,id));
      assign(a_set, fetch(A,id));
      assign(b_set, fetch(B,id));
      assign(merge_set, merge_timer_op_sets(base_set, a_set, b_set));
      bind(id, merge_set, MERGE);
    end loop;
    return MERGE;
  end merge_timer_op_maps;

function merge_timing_data(BASE, A, B: millisec) return millisec is

  begin
    if BASE = A
      then if BASE = B
        then return BASE;
        else return B;
      end if;
      else if BASE = B
        then return A;
        else if A = B
          then return A;
          else return system.max_int;
        end if;
      end if;
    end if;
  end merge_timing_data;
```

```
function merge_period(BASE, A, B: timing_map) return timing_map is

  MERGE: timing_map;
  BASEVAL, AVAL, BVAL: millisec := 0;

  begin
    assign(MERGE, empty_timing_map);
    for id: psdl_id, m: millisec in timing_map_pkg.scan(BASE) loop
      if member(id, A) and member(id,B)
        then
          AVAL := fetch(A, id);
          BVAL := fetch(B, id);
          bind(id, merge_timing_data(m,AVAL,BVAL), MERGE);
      end if;
    end loop;
    for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
      if not member(id, MERGE) and not member(id,BASE)
        then
          if member(id, B)
            then
              BVAL := fetch(B, id);
              if m /= BVAL then
                bind(id, system.max_int, MERGE);
              end if;
            else
              bind(id, m, MERGE);
          end if;
      end if;
    end loop;
    for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
      if not member(id, A) and not member(id,BASE)
        then
          bind(id, m, MERGE);
      end if;
    end loop;
    return MERGE;
  end merge_period;
```

```
function merge_fw_or_mrt(BASE, A, B: timing_map) return timing_map is

  MERGE: timing_map;
  BASEVAL, AVAL, BVAL: millisec := 0;

begin
  assign(MERGE, empty_timing_map);
  for id: psdl_id, m: millisec in timing_map_pkg.scan(BASE) loop
    if member(id, A) and member(id,B)
      then
        AVAL := fetch(A, id);
        BVAL := fetch(B, id);
        bind(id, merge_met(m,AVAL,BVAL), MERGE);
      end if;
  end loop;
  for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
    if not member(id, MERGE) and not member(id,BASE)
      then
        if member(id, B)
          then
            BVAL := fetch(B, id);
            if m /= BVAL then
              bind(id, system.max_int, MERGE);
            end if;
          else
            bind(id, m, MERGE);
        end if;
    end if;
  end loop;
  for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
    if not member(id, A) and not member(id,BASE)
      then
        bind(id, m, MERGE);
    end if;
  end loop;
  return MERGE;
end merge_fw_or_mrt;
```

```
function merge_mcp(BASE, A, B: millisec) return millisec is

  A_DIFF_BASE, B_DIFF_BASE, A_INT_B: millisec;

  begin
    if A >= B
      then A_INT_B := B;
      else A_INT_B := A;
    end if;
    if BASE <= A
      then A_DIFF_BASE := A;
      else A_DIFF_BASE := system.max_int;
    end if;
    if BASE <= B
      then B_DIFF_BASE := B;
      else B_DIFF_BASE := system.max_int;
    end if;
    if A_DIFF_BASE >= A_INT_B
      then if A_DIFF_BASE >= B_DIFF_BASE
        then return A_DIFF_BASE;
        else return B_DIFF_BASE;
      end if;
      else if A_INT_B >= B_DIFF_BASE
        then return A_INT_B;
        else return B_DIFF_BASE;
      end if;
    end if;
  end merge_mcp;
```

```
function merge_min_call_per(BASE, A, B: timing_map) return timing_map is

  MERGE: timing_map;
  BASEVAL, AVAL, BVAL: millisec := 0;

  begin
    assign(MERGE, empty_timing_map);
    for id: psdl_id, m: millisec in timing_map_pkg.scan(BASE) loop
      if member(id, A) and member(id,B)
        then
          AVAL := fetch(A, id);
          BVAL := fetch(B, id);
          bind(id, merge_mcp(m,AVAL,BVAL), MERGE);
      end if;
    end loop;
    for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
      if not member(id, MERGE) and not member(id,BASE)
        then
          if member(id, B)
            then
              BVAL := fetch(B, id);
              if m /= BVAL then
                bind(id, system.max_int, MERGE);
              end if;
            else
              bind(id, m, MERGE);
          end if;
      end if;
    end loop;
    for id: psdl_id, m: millisec in timing_map_pkg.scan(A) loop
      if not member(id, A) and not member(id,BASE)
        then
          bind(id, m, MERGE);
      end if;
    end loop;
    return MERGE;
  end merge_min_call_per;

end proto_impl_merge_pkg;
```

5. prototype_dependency_graph_pkg

```
--------------------------------------------------------------------------
-- COMPONENT NAME  : PACKAGE PROTOTYPE_DEPENDENCY_GRAPH_PKG
--                   ( prototype_dependency_graph_pkg_s.a )
-- USAGE           : Used to perform change-merging on prototype
--                   dependency graphs.
-- AUTHOR          : David A. Dampier
-- DATE OF CREATION : 19 April 1994
-- LANGUAGE USED   : Ada
-- COMPILER  USED  : Sun Ada 1.0
-- PURPOSE         : Provides specifications for the functions necessary
--                   to merge PSDL prototype dependency graphs.
--------------------------------------------------------------------------
with TEXT_IO; use TEXT_IO;
with a_strings; use a_strings;
with generic_map_pkg;
with generic_set_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_graph_pkg; use psdl_graph_pkg;
with psdl_component_pkg; use psdl_component_pkg;

package prototype_dependency_graph_pkg is

procedure assign(x: in out edge_set; y: in edge_set) renames
    edge_set_pkg.assign;

type prototype_dependency_graph is new psdl_graph;

function empty_PDG return prototype_dependency_graph;

function build_PDG(P: in psdl_component)
                                    return prototype_dependency_graph;

function preserved_part(Base,A,B: in prototype_dependency_graph)
                        return prototype_dependency_graph;

function create_slice(G: in prototype_dependency_graph;E: in edge)
                        return prototype_dependency_graph;

function create_slice(G: in prototype_dependency_graph;E: in edge_set)
                        return prototype_dependency_graph;
```

191

```
function compare_slices(S1,S2: in prototype_dependency_graph)
                                              return boolean; .

function graph_union(G1,G2: in prototype_dependency_graph)
                         return prototype_dependency_graph;

function graph_merge(G1,G2,G3: in prototype_dependency_graph)
                         return prototype_dependency_graph;

function affected_part(G,B: in prototype_dependency_graph)
                         return prototype_dependency_graph;

function compare_graphs(G1,G2,S: in prototype_dependency_graph)
                                              return boolean;

end prototype_dependency_graph_pkg;
```

```
package body prototype_dependency_graph_pkg is

function empty_PDG return prototype_dependency_graph is

  begin
    return empty_psdl_graph;
  end;

--------------------------------------------------------------------------
-- This function takes a PSDL component and creates from the
-- implementation graph, a prototype dependency graph.
--------------------------------------------------------------------------
function build_PDG(P: in psdl_component)
                                    return prototype_dependency_graph is

  G: prototype_dependency_graph;
  O: psdl_id;
  VERTS: id_set;
  OUTEDGE: a_string;

  begin
    assign(G, empty_PDG);
    assign(G, prototype_dependency_graph(graph(P)));
    assign(VERTS, vertices(G));
    for id: psdl_id in id_set_pkg.scan(VERTS) loop
      if equal(successors(id, G), empty_id_set)
        then
          OUTEDGE := copy(id&EXT);
          assign(G, add_edge(id, EXT, OUTEDGE, G, O));
          if not has_vertex(EXT, G)
            then
              assign(G, add_vertex(EXT, G));
          end if;
      end if;
    end loop;
    return G;
  end build_PDG;
```

194

```
-----------------------------------------------------------------------------
--  This function calculates the part of Base,A,and B which are identical.
-----------------------------------------------------------------------------


function preserved_part(Base,A,B: in prototype_dependency_graph)
                        return prototype_dependency_graph is

  PP, S1, S2, S3: prototype_dependency_graph;
  E: edge;
  D: edge_set;

  begin
    assign(PP, empty_PDG);
    assign(S1, empty_PDG);
    assign(S2, empty_PDG);
    assign(S3, empty_PDG);
    assign(D, edges(Base));
    for E:edge in edge_set_pkg.scan(D) loop
      assign(S1, create_slice(Base, E));
      assign(S2, create_slice(A, E));
      assign(S3, create_slice(B, E));
      if compare_slices(S1, S2) and then compare_slices(S1, S3)
        then
          assign(PP, graph_union(S1, PP));
      end if;
      recycle(S1);
      recycle(S2);
      recycle(S3);
    end loop;
    return PP;
  end preserved_part;
```

```
-------------------------------------------------------------------------
-- This function creates a graph which contains only the part of G which
-- affects the output values written to the edge E.
-------------------------------------------------------------------------

function create_slice(G: in prototype_dependency_graph;E: in edge)
                          return prototype_dependency_graph is

  S1, S2: prototype_dependency_graph;
  D: edge;
  C: edge_set;

  begin
    assign(S1, empty_PDG);
    assign(S2, empty_PDG);
    if has_edge(E.x, E.y, G) then
      assign(S1, add_edge(E.x,E.y,E.stream_name,
                          S1, latency(E.x,E.y,E.stream_name,G)));
      assign(S1, add_vertex(E.x, S1, maximum_execution_time(E.x,G)));
      if eq(E.y, EXT)
        then
          assign(S1, add_vertex(E.y, S1));
      end if;
      assign(C, edges(G));
      while not compare_slices(S1, S2) loop
        assign(S2, S1);
        for D:edge in edge_set_pkg.scan(C) loop
          if (has_vertex(D.y, S1) and not eq(D.y, EXT))
            then
              assign(S1, add_edge(D.x,D.y,
                    D.stream_name,S1, latency(D.x,D.y,D.stream_name,G)));
              assign(S1, add_vertex(D.x, S1,
                    maximum_execution_time(D.x,G)));
          end if;
        end loop;
      end loop;
    end if;
    return S1;
  end create_slice;
```

```
----------------------------------------------------------------------
-- This function calculates the union of the graphs G1 and G2.      `
----------------------------------------------------------------------

function graph_union(G1,G2: in prototype_dependency_graph)
                      return prototype_dependency_graph is

  G: prototype_dependency_graph;
  V: psdl_id;
  W: id_set;
  E: edge;
  D: edge_set;

  begin
    assign(G, empty_PDG);
    assign(G,G1);
    assign(W, vertices(G2));
    assign(D, edges(G2));
    for V:psdl_id in id_set_pkg.scan(W) loop
      if not(has_vertex(V, G))
        then
          assign(G, add_vertex(V, G,maximum_execution_time(V,G2)));
      end if;
    end loop;
    for E:edge in edge_set_pkg.scan(D) loop
      if not (edge_set_pkg.member(E,edges(G)))
        then
          assign(G, add_edge(E.x,E.y,
                  E.stream_name, G,latency(E.x,E.y,E.stream_name,G2)));
      end if;
    end loop;
    return G;
  end graph_union;
```

```
-------------------------------------------------------------------------
-- This function merges three graphs using the function graph_union.
-------------------------------------------------------------------------
function graph_merge(G1,G2,G3: in prototype_dependency_graph)
                        return prototype_dependency_graph is

  G: prototype_dependency_graph;

  begin
    assign(G, empty_PDG);
    assign(G, graph_union(G1, G2));
    assign(G, graph_union(G, G3));
    return G;
  end graph_merge;


-------------------------------------------------------------------------
-- This function calculates the part of G which is not contained in P.
-------------------------------------------------------------------------
function affected_part(G,B: in prototype_dependency_graph)
                        return prototype_dependency_graph is

  A, SG, SB: prototype_dependency_graph;
  E: edge;
  D: edge_set;

  begin
    assign(A, empty_PDG);
    assign(SG, empty_PDG);
    assign(SB, empty_PDG);
    assign(D, edges(G));
    for E:edge in edge_set_pkg.scan(D) loop
      assign(SG, create_slice(G, E));
      assign(SB, create_slice(B, E));
      if not compare_slices(SG, SB)
        then
          assign(A, add_edge(E.x,E.y,
                   E.stream_name, A, latency(E.x,E.y,E.stream_name,G)));
          assign(A, add_vertex(E.x, A, maximum_execution_time(E.x, G)));
          assign(A, add_vertex(E.y, A, maximum_execution_time(E.y, G)));
      end if;
    end loop;
    return A;
  end affected_part;
```

```
---------------------------------------------------------------------------
--  This function compares the graphs G1 and G2 with respect to the
--  slice S. If each of G1 and G2 are the same with respect to S, then
--  it returns TRUE.
---------------------------------------------------------------------------
function compare_graphs(G1,G2,S: in prototype_dependency_graph)
                        return boolean is

  E: edge_set;
  T, V: prototype_dependency_graph;

  begin
    assign(T, empty_PDG);
    assign(V, empty_PDG);
    assign(E, edges(psdl_graph(S)));
    assign(T, create_slice(G1, E));
    assign(V, create_slice(G2, E));
    return(compare_slices(T, V));
  end compare_graphs;

end prototype_dependency_graph_pkg;
```

LIST OF REFERENCES

[1] Agrawal, H. and Horgan, J., "Dynamic Program Slicing", *Proceedings of the ACM SIG-PLAN '90 Conference on Programming Language Design and Implementation*, ACM, June 1991.

[2] Agrawal, H., *Towards Automatic Debugging of Computer Programs*, PhD Dissertation, Purdue University, September 1991.

[3] Altizer, C. and Berzins, V., "Role of Translation Mechanisms in Software Comprehension", *CSM-92 Program Comprehension Workshop Notes*, IEEE Computer Society, November 1992.

[4] Naval Postgraduate School Technical Report CS-92-020, *A Design Management and Job Assignment System*, Badr, S. and Berzins, V., 1992.

[5] Badr, S. and Luqi, "A Version and Configuration Model for Software Evolution", *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, KSI, June 1993, pp. 225-227.

[6] Berzins, V., "On Merging Software Extensions", *Acta Informatica*, Springer-Verlag, 1986, pp. 607-619.

[7] Naval Postgraduate School Technical Report 52-87-032, *The Semantics of Inheritance in SPEC*, Berzins, V. and Luqi, 1987.

[8] Berzins, V. and Luqi, "Semantics of a Real-Time Language", *Proceedings of the Real-Time Systems Symposium*, December 1988.

[9] Berzins, V., "Software Merge: Semantics of Combining Changes to Programs", to appear in ACM Transactions on Programming Languages and Systems, 1994.

[10] Berzins, V., "Software Merge: Models and Methods for Combining Changes to Programs", *Journal of Systems Integration*, Vol. 1, Num. 2, Kluwer, August 1991, pp. 121-141.

[11] Berzins, V. and Luqi, *Software Engineering With Abstractions*, Addison-Wesley, 1991.

[12] Berzins, V., Luqi and Yehudai, A., "Using Transformations in Specification-Based Prototyping", *IEEE Transactions on Software Engineering*, Vol. 19, Num. 5, IEEE, May 1993, pp. 436-452.

[13] Naval Postgraduate School Technical Report CS 94-009, *Software Merge: Conditional Flowgraphs and Program Slices*, Berzins, V. and Dampier, D., 1994.

[14] Birkhoff, G., *Lattice Theory*, American Mathematical Society, 1948.

[15] Birkhoff, G. and Bartee, T., *Modern Applied Algebra*, McGraw Hill, 1970.

[16] Bloch, N., *Abstract Algebra with Applications*, Prentice Hall, 1987.

[17] Borison, E., "A Model of Software Manufacture", *Proceedings of an International Workshop, Trondheim, Norway*, Springer-Verlag, June 1986, pp. 197-220.

[18] Boudriga, N., Elloumi, F., and Mili, A., "On the Lattice of Specifications: Applications to a Specification Methodology", *Formal Aspects of Computing*, Vol. 4, Springer-Verlag, 1991, pp. 544-571.

[19] Brock, J. and Ackerman, W., "Scenarios: A Model of Non-determinate Computation", "Formalization of Programming Concepts", 1981.

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

[21] Naval Postgraduate School Technical Report CS-92-014, *A Model for Merging Software Prototypes*, Dampier, D. and Luqi, 1992.

[22] Dampier, D. and Luqi, "Automated Software Maintenance Using Comprehension and Specification", *CSM-92 Program Comprehension Workshop Notes*, IEEE Computer Society, November 1992, pp. 24-26.

[23] Dampier, D., Luqi, and Berzins, V., "Automated Merging of Software Prototypes", *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, KSI, June 1993, pp. 604-611.

[24] Dampier, D. and Berzins, V., "A Slicing Method for Semantic Based Merging of Software Prototypes", *Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging and Integration*, NPS, October 1993, pp. 22-24.

[25] Dampier, D., Luqi, and Berzins, V., "Automated Merging of Software Prototypes", *Journal of Systems Integration*, Vol. 4, Num. 1, Kluwer, January 1994.

[26] Fountain, H., *Rapid Prototyping: A Survey and Evaluation of Methodologies and Models*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1990.

[27] Halmos, P., *Lectures on Boolean Algebras*, Van Nostrand, 1963.

[28] 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.

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

[30] Keller, R., "Denotational Models for Parallel Programs with Indeterminate Operators", *Formal Description of Programming Concepts*, North Holland, August 1977, pp. 337-366.

[31] Kosinski, P., "A Straightforwad Denotational Semantics for Non-Determinate Data Flow Programs", *"Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, January 1978, pp. 214-219.

[32] Kramer, B., Luqi, and Berzins, V., "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Transactions on Software Engineering*, Vol. 19, Num. 5, IEEE, May 1993, pp. 453-477.

[33] Linger, R., Mill, H., and Witt, B, *Structured Programming: Theory and Practice*, Addison Wesley, 1979.

[34] Lidl, R. and Pilz, G., *Applied Abstract Algebra*, Springer-Verlag, 1984.

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

[36] Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989.

[37] Luqi, "A Graph Model for Software Evolution", *IEEE Transaction on Software Engineering*, Vol. 16, Num. 8, August 1990.

[38] Luqi, "Computer-Aided Prototyping for a Command And Control System Using CAPS", *IEEE Software*, January 1992.

[39] McKinsey, J. and Tarski, A., "On Closed Elements in Closure Algebras", *Annals of Mathematics* 47(1), January 1946, pp. 122-162.

[40] Piersall, J., *Army Research, Development, and Acquisition Bulletin*, January-February 1994, pp. 37-40.

[41] University of Wisconsin-Madison Technical Report CS-777, "The Semantics of Program Slicing", Reps, T. and Yang, W., 1988.

[42] University of Wisconsin-Madison Technical Report CS-856, "On the Algebraic Properties of Program Integration", Reps, T., June 1989.

[43] Siverberg, I., *Source File Management with SCCS*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[44] Tanik, M. and Yeh, R., "Rapid Prototyping in Software Development", *IEEE Computer*, Vol. 22, May 1989, pp. 9-10.

[45] Tichy, W., "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982, pp. 58-67.

[46] Venkatesh, G., "The Semantic Approach to Program Slicing", *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, ACM, June 1991.

[47] Weiser, M., "Program Slicing", *IEEE Transactions on Software Engineering*, IEEE, July 1984, pp. 352-357.

[48] University of Wisconsin-Madison Technical Report CS-962, "A New Algorithm for Semantics-Based Program Integration", Yang, W., 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304–6145

2. Dudley Knox Library 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943–5002

3. Chairman, Computer Science Department 1
   Code CS
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Valdis Berzins 5
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

5. Professor Daniel Dolk 1
   Department of Systems Management
   Naval Postgraduate School
   Monterey, CA 93943

6. Professor Luqi 5
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

7. Professor Mantak Shing 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

8. Professor Craig Rasmussen 1
   Department of Mathematics
   Naval Postgraduate School
   Monterey, CA 93943

9.   Dr. Hiralal Agrawal   1
        Bellcore, MRE2D-388
        445 South Street
        Morristown, NJ 07960

10.   Dr. Sergio Antoy   1
        University of Portland
        Computer Science Department
        Portland, OR 97207

11.   Colonel Salah El-Din M. Badr   1
        101 El-Tyaran Street
        Nasser City, Cairo
        EGYPT

12.   Dr. Alfs Berztiss   1
        University of Pittsburgh
        Computer Science Department
        Room 321, Alumni Hall, University Drive
        Pittsburgh, Pennsylvania 15260

13.   Colonel James T. Blake, Ph.D.   1
        U.S. Army Research Laboratory
        ATTN: AMSRL-CI
        Aberdeen Proving Ground, MD 21005-5067

14.   Dr. Dan Cooke   1
        Department of Computer Science
        University of Texas at El Paso
        El Paso, Texas 79968-0518

15.   Dr. Bill Griswold   1
        Department of Computer Science and Eng.
        University of California San Diego
        9500 Gilman Drive
        La Jolla, CA 92093

16.   Dr. David Hislop   1
        U.S. Army Research Office Elec Div
        4300 S. Miami Blvd
        Research Triangle Park, North Carolina 27709-2211

17. Dr. Susan Horwitz   1
     Department of Computer Science
     University of Wisconsin-Madison
     1210 W. Dayton Street
     Madison, Wisconsin 53706

18. Dr. J.C. Huang   1
     Department of Computer Science
     University of Houston
     Houston, TX 77204-3475

19. CPT Kevin Jones   1
     Department of Electrical and Computer Eng.
     Royal Military College of Canada
     Kingston, Ontario
     CANADA
     K7K-3L0

20. Dr. Deepak Kapur   1
     Department of Computer Science
     State University of New York
     Albany, New York 12222

21. Dr. Arun Lakhotia   1
     Center for Advanced Computer Studies
     University of Southwestern Louisiana
     Lafayette, LA 70504

22. Dr. Karl Levitt   1
     Department of Computer Science
     University of California at Davis
     Davis, California

23. Dr. David Luckham   1
     Department of Computer Science
     Stanford University
     Palo Alto, California 94305

24. Dr. A. Mili   1
     Computer Science
     Faculty of Science
     University of Ottawa
     150 Louis Pasteur/Priv.
     OTTAWA, Ontario K1N 6N5
     CANADA

25.   Dr. Roland Mittermeir   1
      University Klagenfurt, Inst F Informatik
      Universitaetsstr 63, A-9022
      Klagenfurt
      Austria

26.   G. Ramalingam   1
      IBM Hawthorne Research Center
      P.O. Box 704
      Yorktown Heights, NY 10598

27.   Dr. Thomas Reps   1
      Department of Computer Science
      University of Wisconsin-Madison
      1210 W. Dayton Street
      Madison, Wisconsin 53706

28.   Dr. John Salasin   1
      SEI
      801 N. Randolh St. Suite 405
      Arlington, Virginia 22203

29.   Dr. Alan Shaw   1
      Department of Computer Science
      University of Washington
      Seattle, Washington 98195

30.   Dr. Douglas Smith   1
      Kestrel Institute
      3260 Hillview Avenue
      Palo Alto, California 94304

31.   Dr. M. K. Srinivas   1
      Artificial Intelligence Center,
      SRI International
      Menlo Park, California 94025

32.   Dr. David Stemple   1
      Department of Computer Science
      University of Massachusetts at Amherst
      Amherst, MA 01003

33.   Dr. Leon Sterling   1
      Case Western Reserve University
      Cleveland, OH

34. Prof. Murat Tanik   1
     Department of Computer Science and Eng.
     Southern Methodist University
     Dallas, Texas 75275-0122

35. Dr. Douglas Waugh   1
     SEI
     801 N. Randolh St. Suite 405
     Arlington, Virginia 22203

36. Dr. Mark Wegman   1
     IBM Hawthorne Research Center
     P.O. Box 704
     Yorktown Heights, NY 10598

37. Captain David Anthony Dampier   10
     8508 Hopewell
     El Paso, TX 79925