AFRL-IF-WP-TR-2000-1502

ADVANCED AVIONICS VERIFICATION AND VALIDATION PHASE II (AAV&V-II)

ROBERT E. COOK, JR.

TASC/LITTON
55 WALKERS BROOK DRIVE
READING, MASSACHUSETTS 01867

JANUARY 1999

FINAL REPORT FOR 12/01/1994 – 01/01/1999

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE OH 45433-7334

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Charles P. Satterthwaite

CHARLES P. SATTERTHWAITE, Project Engineer
Embedded Information System Engineering Branch
AFRL/IFTA

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Information System Engineering Branch
AFRL/IFTA

Eugene C Blackburn

EUGENE C. BLACKBURN, Chief
Information Technology Division
AFRL/IFT

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JANUARY 1999	FINAL REPORT FOR 12/01/1994 - 01/01/1999

4. TITLE AND SUBTITLE
ADVANCED AVIONICS VERIFICATION AND VALIDATION PHASE II (AAV&V-II)

5. FUNDING NUMBERS
C F33615-92-D-1052
PE 78611
PR 3090
TA 01
WU 14

6. AUTHOR(S)
ROBERT E. COOK, JR.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TASC/LITTON
55 WALKERS BROOK DRIVE
READING, MASSACHUSETTS 01867

8. PERFORMING ORGANIZATION REPORT NUMBER
TR-06664-3

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7334
POC: CHARLES P. SATTERTHWAITE, AFRL/IFTA, 937-255-6548 EXT 3584

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-WP-TR-2000-1502

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

PROBLEM STATEMENT: As the complexity of software projects for embedded avionics applications increases, it becomes increasingly obvious that new innovative techniques will have to be used to help test those projects. Current methods of testing often require man-in-the-loop as well as extensive set up times. Often, the verification and validation of moderate software changes requires several weeks of man hours to accomplish.

ADVANCED AVIONICS VERIFICATION AND VALIDATION (AAV&V) AS AN INNOVATIVE TECHNIQUE: The AAV&V project addresses the above problem by providing software developers and testers access to current testing technologies while investigating and proposing new techniques for validation and verification. Current software developers and testers can take advantage of an AAV&V Tool which sets on a powerful engineering workstation with open system architecture and gives them coverage and static analysis capability as well as documentation access and generation. Future software developers and testers will enjoy expansion of language options on the AAV&V tool's front end, as well as access to Formal Methods and Statistical techniques.

14. SUBJECT TERMS
Verification and Validation, Static and Coverage Analysis

15. NUMBER OF PAGES
138

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

LIST OF TABLES vii

EXECUTIVE SUMMARY ES-1

1. INTRODUCTION 1-1
   - 1.1 Background 1-1
   - 1.2 AAV&V Program Overview 1-3
     - 1.2.1 Goal of the AAV&V-II Program 1-3
     - 1.2.2 Structure and Tasks of the AAV&V-II Program 1-3
   - 1.3 Summary of Results 1-7
     - 1.3.1 Results of Interprocedural Analysis Algorithm Development 1-7
     - 1.3.2 Results of Advanced Prototype Development for Ada Programs 1-8
     - 1.3.3 Results of Prototype Development for Programs Written in a Subset of JOVIAL 1-9
     - 1.3.4 Results of the Survey and Evaluation of Automated Testing Support 1-11
     - 1.3.5 Results of Formulation of Approach Supporting Automated Testing of Implemented Software Modifications 1-11
   - 1.4 Recommendations 1-12
   - 1.5 Report Organization 1-13

2. THE INTERPROCEDURAL ANALYSIS ALGORITHMS 2-1
   - 2.1 The Interprocedural Control Flow Graph 2-2
   - 2.2 The Reachability Algorithm 2-3
   - 2.3 The Immediate Forward Dominators Algorithm 2-5
   - 2.4 The Forward Dominators Algorithm 2-6
   - 2.5 The Data Dependence Algorithm 2-7
   - 2.6 The Strong Control Dependence Algorithm 2-8
   - 2.7 The Weak Control Dependence Algorithm 2-8
   - 2.8 The Indirect Dependence Algorithms 2-9
   - 2.9 Improvements to the Pleiades Object Management System 2-9

3. THE IMPACT PROTOTYPE FOR ADA 3-1
   - 3.1 Design of the Advanced Prototype for Ada 3-2
     - 3.1.1 The Front-End Component 3-3
     - 3.1.2 The Analysis Component 3-4
     - 3.1.3 The Persistent Database Component 3-4
     - 3.1.4 The ImpACT User Interface 3-4

3.2 Implementation of the Advanced Prototype for Ada 3-6
3.2.1 The Front-End Component 3-6
3.2.2 The Analysis Component 3-7
3.2.3 The Persistent Database Component 3-7
3.2.4 The ImpACT User Interface 3-9
3.3 Demonstration of the Advanced Prototype for Ada 3-21
3.3.1 The Binary Search Demonstration 3-21
3.3.2 The F-16 FCC Navigation Support Component Demonstration 3-22
3.4 Recommendations for the Prototype for Ada 3-22

4. THE IMPACT PROTOTYPE FOR JOVIAL 4-1
4.1 Design of the Prototype for JOVIAL 4-1
4.2 Implementation of the Prototype for JOVIAL 4-2
4.2.1 The Syntactic and Semantic Analyzer for JOVIAL 4-3
4.2.2 The Control Graph Generator for JOVIAL 4-4
4.3 Demonstration of the Prototype for JOVIAL 4-5
4.3.1 Demonstration on the Avionics-like Code 4-5
4.3.2 Demonstration of B-1B CITS subsystems 4-6
4.4 Recommendations for the Prototype for JOVIAL 4-8

5. AUTOMATED TESTING SURVEY AND EVALUATION 5-1
5.1 Research Resources Utilized 5-1
5.1.1 The World Wide Web 5-1
5.1.2 Journals and Proceedings 5-2
5.1.3 Technical Report Databases 5-2
5.1.4 Following References 5-2
5.2 Summary of Methods 5-2
5.2.1 Test Coverage Criteria 5-3
5.2.2 Code Annotations and Assertion Checkers 5-7
5.2.3 Automated Test Generation 5-10
5.2.4 Automated Selection of Regression Tests 5-17
5.2.5 Formal Methods 5-21
5.3 Evaluation of Methods 5-29
5.3.1 Technical Effectiveness 5-30
5.3.2 Automation Requirements and Restrictions 5-36
5.3.3 Effort Required to Performed Verification and Validation 5-38
5.4 Recommendation 5-39

6. THE AUTOMATED TESTING PROTOTYPE 6-1
6.1 The Top-Level Design 6-2
6.1.1 The Generate Coverage Requirements Process 6-3
6.1.2 The Create Instrumented Program Process 6-3
6.1.3 The Select Existing Tests Process 6-5
6.1.4 The Generate New Tests Process 6-5
6.1.5 The Software Change V&V Control Process 6-5

6.2 The Detailed Design 6-6
6.3 Investigation and Implementation 6-6

7. CONCLUSIONS AND RECOMMENDATIONS 7-1
7.1 Summary of Tasks of the AAV&V-II Program 7-2
7.1.1 A Production-quality AAV&V System for Ada 7-2
7.1.2 A Prototype AAV&V System for JOVIAL 7-3
7.1.3 Survey on the Science of Testing 7-3
7.1.4 Prototype for Automated Testing Support 7-3
7.2 Conclusions 7-4
7.3 Recommendations 7-6
7.3.1 Improvements to the Interprocedural Dependence Analysis Algorithms 7-7
7.3.2 Enhanced Support for the Display and Manipulation of Large Graphs 7-7
7.3.3 Complete Support for MIL-STD-1589C JOVIAL 7-7
7.3.4 Support for Additional Languages 7-8
7.3.5 Combined Software and Hardware Analysis 7-8
7.3.6 Development of the Prototype for Automated Testing 7-8

APPENDIX A EVALUATION CRITERIA A-1

APPENDIX B SURVEY OF AUTOMATED TESTING TECHNIQUES B-1

APPENDIX C PORTABILITY ISSUES C-1

APPENDIX D DEVELOPMENT ENVIRONMENT D-1

REFERENCES R-1

LIST OF FIGURES

Figure 1-1 Overview of AAV&V-II Tasks 1-5
Figure 3-1 ImpACT Design 3-2
Figure 3-2 The Main ImpACT Window, with Database Pulldown Menu 3-8
Figure 3-3 Interface for Creating and Initializing a New Database 3-9
Figure 3-4 Interface for Creating Analysis Information for Ada Program 3-10
Figure 3-5 Source Code Listing View 3-11
Figure 3-6 Fully Expanded Declaration Diagram for a Package Body 3-12
Figure 3-7 Control Flow Graph View 3-13
Figure 3-8 Data Flow Information Window 3-14
Figure 3-9 Interprocedural Control Flow Graph with Nodes Depicting Calls 3-15
Figure 3-10 Interprocedural Control Flow Graph Depicted as a Call Hierarchy Chart 3-16
Figure 3-11 Interprocedural Control Flow Graph with Lines Depicting Calls 3-16
Figure 3-12 Program Dependence Graph View 3-17
Figure 3-13 Zoomed Portion of Program Dependence Graph View, with Highlighting 3-17
Figure 3-14 Statement Dependence Graph View 3-18
Figure 3-15 Complexity Metrics View 3-19
Figure 5-1 Example Where Statement Coverage May Not Expose Fault 5-3
Figure 5-2 Specification of Binary Search in Anna 5-8
Figure 5-3 Example of Annotated Sort Function 5-9
Figure 5-4 Example of an EVES s-Verdi Specification 5-22
Figure 5-5 Example of a Larch/Ada Specification 5-23
Figure 6-1 Context Diagram for Software Change Verification and Validation System 6-2
Figure 6-2 Level 1 Data Flow Diagram for the Software Change Verification and Validation Prototype 6-4

LIST OF TABLES

Table 2-1 Time to Perform Selected Kinds of Analysis 2-2
Table 4-1 JOVIAL Language Features Implemented and Not Implemented 4-4
Table 4-2 CITS JOVIAL Nuances 4-7
Table 5-1 Evaluation Criteria Summary 5-31

EXECUTIVE SUMMARY

The Advanced Avionics Verification and Validation Phase II (AAV&V-II) program began December 1994 and ran through November 1998. The objective of the AAV&V-II program was to advance the state-of-the art in automated verification and validation for real-time, mission-critical avionics software. During the AAV&V-II program, TASC:

- designed and implemented algorithms for interprocedural dependence analysis at the statement level
- developed a near-commercial-quality system for V&V of Ada programs, partially based on the proof-of-concept prototype developed in the original AAV&V program
- developed a prototype for automated V&V supporting software written in the JOVIAL programming language
- improved upon the implementation of its state-of-the-art V&V methodology and process
- returned to avionics maintainers to demonstrate how our V&V support addresses the needs they expressed during the original AAV&V program
- performed an extensive study and evaluation of existing and emerging techniques for V&V of software changes

TASC achieved the following significant results during the AAV&V-II program:

- *Novel algorithms for interprocedural analysis.* Algorithms were designed and implemented, based on pioneering research performed by TASC, for interprocedural analysis of programs at the statement level. These algorithms were implemented as extensions to the ProDAG system, developed by the University of California, Irvine, for procedural dependence analysis at the statement level.
- *A production-quality AAV&V system implementing portions of the AAV&V methodology for Ada programs.* This system addressed the needs of avionics software maintainers in the area of program understanding, and interprocedural dependence analysis in particular. The system was rehosted to a popular platform and enhanced from its original concept. The system made use of common standards in graphical user interface development and implementation, and continued to reuse third-party and COTS software.
- *An advanced proof-of-concept AAV&V prototype that implemented portions of the AAV&V methodology for JOVIAL programs.* The prototype exhibited the same capabilities as the system for Ada programs, and supported a sizeable subset

of the MIL-STD-1589C JOVIAL language definition. In developing the prototype, TASC demonstrated the benefits of its use of a language-independent representation of source code as input to the analysis portions of the system.

- *Enthusiastic reception of the implemented portions of the AAV&V system by avionics software maintainers.* In returning to the avionics maintainers that were surveyed in the original AAV&V program, they responded by enthusiastically stating that they needed capabilities such as those supplied by the AAV&V-II program. They also stated that this program was rare in that it actively addressed issues they faced in their everyday duties.
- *A comprehensive evaluation of the state-of-the-practice and state-of-the-art in V&V of implemented software changes.* TASC performed a thorough literature search to identify existing and emerging tools and techniques for V&V of implemented software changes. These tools and methods were evaluated using a set of criteria developed to address the important issues in the V&V of real-time, mission-critical software. Using the literature survey and the evaluation results, TASC suggested a novel approach to selective regression testing that supports the special needs involved in testing avionics software.

Research in advanced V&V, such as the AAV&V-II program, becomes increasingly important as more crucial parts of systems, such as advanced weapons systems, national security information, transportation, and health care systems, are implemented in software, and whose failure could have catastrophic consequences. Software maintenance and testing is becoming more difficult, time-consuming, and expensive because of the exponential growth in the size, complexity, and importance of software systems. Improved automated V&V techniques, tools, and processes are necessary if current and future software systems are to be manageable.

The automated AAV&V system that is based upon the results of the AAV&V-II program reduces the V&V, maintenance, and test burden by providing the following benefits:

- *Reduced manual effort,* through automation of V&V techniques.
- *Reduced number of tests,* through improved analysis and test selection techniques.
- *Improved software quality,* through earlier detection of errors and reduction of the number of errors introduced during maintenance, as well as through controlled growth of program complexity.
- *Improved software understanding,* through improved capture and visualization of software requirements, design, and implementation decisions.
- *Improved software reusability,* through dependence analysis of potentially reusable software segments.

Improvements in the V&V process and its incorporation in the software development process based upon the AAV&V-II program will result in reduced software development and maintenance cost and effort and result in more robust, reliable, maintainable, and reusable software.

The AAV&V advanced prototype developed under the AAV&V-II program implements portions of the AAV&V methodology that support the current needs of avionics software maintainers, as identified by the survey in the original AAV&V program. The advanced prototype, referred to as ImpACT (IMPact Analysis Capability Tool) provides the following automated support for software maintenance:

- *Aids the maintainer in understanding the software.*
- *Aids the maintainer in determining the risk associated with the contemplated modification.*
- *Aids the maintainer in evaluating the suitability of a modification.*
- *Aids the maintainer in choosing the implementation of a modification that least increases the complexity of the software.*
- *Aids the maintainer in determining what portions of the system must be retested, as well as the particular tests that must be rerun.*

The AAV&V-II program consisted of the following four tasks:

- Task I consisted of implementing a near-commercial-quality V&V system for software written in the Ada programming language, based upon the proof-of-concept prototype developed in the original AAV&V program. The system included hyperlinked views for a listing of the source code, a control flow graph, an interprocedural control flow graph, a program dependence graph, and a statement dependence graph. Views were also provided for data flow information, declaration information, and program complexity metrics. This refinement of the original AAV&V prototype was hosted on a Sun workstation platform, and used The X Window System and OSF/Motif libraries in the implementation of its graphical user interface. This task also involved the design and development of algorithms for interprocedural dependence analysis at the statement level. The ability to translate Ada programs into their language-independent IRIS representation, as well as to generate control flow graphs from that IRIS representation, was tested using over 1000 lines of Flight Control Computer Code for the F-16. The entire system was tested with, among other small examples, a sample binary search program.
- Task II consisted of implementing a prototype V&V system, with capabilities similar to the ones developed under Task I, for software written in the JOVIAL programming language. We developed an LR(1) grammar for

JOVIAL for the MIL-STD-1589C language definition. JOVIAL language elements were compared to those in the Ada programming language, and a specification for representing JOVIAL language constructs in IRIS was created. This specification was reviewed by the developers of IRIS at The University of Massachusetts, and a majority of the language features were supported through addition of actions to the grammar. Several minor modifications were necessary to the portion of the system responsible for generation of control flow graphs. This prototype was used to analyze source code from the B-1B CITS system, modified from the original to compensate for JOVIAL language coverage limitations. The ICE and LDGR systems were analyzed, where ICE was approximately 600 lines of procedure code and LDGR was approximately 1750 lines of procedure code.

- Task III consisted of a comprehensive literature search and evaluation involving present and emerging systems and methods for the V&V of implemented software changes. The evaluation used a set of criteria that TASC developed to measure important aspects of the V&V of real-time, mission-critical software. TASC formulated a recommended approach to the V&V of software changes based on combining selective regression testing, test coverage analysis, and automated test data generation.
- Task IV began a design of the approach recommended in Task III, and started to explore implementation issues. This task, as well as the technical report that was to result from Task III, was aborted in favor of additional work on Task II. This additional work on Task II increased the functionality of the prototype support for JOVIAL programs, and was triggered as a result in the interest shown by B-1B avionics software maintainers in ImpACT.

The current AAV&V prototypes provide support for avionics software maintenance efforts, particularly in the area of software change impact analysis at an interprocedural level. As the complexity and importance of software systems increases, automated V&V provided by a complete AAV&V system, of which ImpACT is a component, will be increasingly important for the complete software life cycle.

1. INTRODUCTION

This report documents the final results of research and development for the Advanced Avionics Verification and Validation Phase II (AAV&V-II) program. The objective of the AAV&V-II program was to advance the state-of-the-art in automated verification and validation for real-time, mission-critical avionics software. The AAV&V-II program included development of algorithms for interprocedural analysis of software, development of an advanced prototype for V&V support of programs written in Ada, development of a prototype for V&V support of programs written in a subset of JOVIAL, and a study and evaluation of existing and emerging approaches for automated support of software testing. The most significant results of the AAV&V-II program are as follows:

- Design and implementation of algorithms for interprocedural analysis.
- An advanced prototype for Ada software that combines hyperlinked views and interprocedural analysis that addresses program comprehension issues experienced by avionics software maintainers.
- A prototype for JOVIAL software that reuses the analysis generation capabilities and user-interface of the advanced prototype for Ada.
- A comprehensive examination and evaluation of methods and tools that support automated testing of implemented changes to software.
- Formulation of an approach supporting automated testing of implemented software modifications.

This chapter describes the background and motivation for the AAV&V-II program, the goals and structure of the AAV&V program, the results of the effort, and the conclusions and recommendations drawn from the program.

1.1 BACKGROUND

Verification and Validation (V&V) is a crucial part of the development and maintenance of avionics software for several reasons, including the following:

- Computers have become crucial parts of systems, such as advanced weapons, national security information, transportation, and health care systems, in which software failure could have catastrophic results.
- Software maintenance has become the most effort-intensive, time-consuming, and expensive part of the software process.

- Improvements in computer hardware and software technology have facilitated an exponential growth in the size, complexity, and importance of software systems, leading to an ever-increasing testing and maintenance burden.

V&V is especially important for mission-critical avionics software. Software is continually becoming involved in more aspects of aircraft operations, resulting in larger OFPs (Operational Flight Programs), more complex interactions between OFPs and between components within an OFP, and more requests for modifications of ever-increasing difficulty. These characteristics lead to the following challenges for the avionics software maintainers:

- *Greater difficulty in understanding and modifying OFPs* - as the size of OFPs increases (to hundreds of thousands or millions of lines of code), complete OFPs become more difficult to understand as a whole. As the number of components of an OFP and the number of interactions between the components increase, it becomes more difficult to understand the individual components. In addition, testing and debugging of an OFP or component and determining the location and type of modification that should be made to correct a problem or add functionality becomes more difficult. Automated tools that facilitate software understanding and the modification of large, complex avionics systems would greatly improve the amount and quality of OFP maintenance that an avionics software maintainer could perform.
- *Increased possibility of adverse effects of modifications* - The increased number and greater complexity of interactions between components and OFPs makes it more difficult to make an "isolated modification" to a component to correct an error or add functionality. Each modification has the possibility of affecting the behavior of other components and thereby introducing unexpected errors. Automated tools that can detect the hidden effects of software modifications and aid in test selection would reduce the time spent debugging modified code and improve the amount and quality of the modifications performed by an avionics software maintainer.
- *Increased burden of verifying and validating modified software* - The test suite for thoroughly testing an OFP upgrade constantly increases throughout the software's life. The burden of manually testing the OFP using an ever-increasing test suite results in longer delays between upgrades or fewer modifications per maintainer during each upgrade. In addition, the tedium associated with the manual testing and the ever-increasing complexity of the software increase the danger of mistakes during testing or selection of inadequate test cases. Automated support of testing would reduce the amount of time, effort, and tedium associated with manual testing. Tools that evaluate the quality of the test suite against the code structure would improve

the quality of V&V and reduce the need for later modifications to correct problems introduced during the maintenance phase.

1.2 AAV&V PROGRAM OVERVIEW

1.2.1 Goal of the AAV&V-II Program

The goal of the AAV&V-II program was to further advance the state-of-the-art in automated V&V techniques and tools for real-time, mission-critical software. The main focus was on improvements to software testing and to software understanding and maintenance. The AAV&V-II program included development of algorithms for interprocedural analysis, an advanced prototype V&V system for software written in Ada, a prototype V&V system for software written in JOVIAL, and a survey and evaluation of techniques providing automated support of software testing.

The V&V system resulting from the AAV&V-II program reduces the V&V, maintenance, and test burden by providing the following benefits:

- *A reduced amount of manual effort* achieved by automating V&V techniques.
- *A reduced number of necessary tests* achieved through improved analysis.
- *Improved software quality* by detecting errors earlier in the software development process and reducing the number of errors introduced during maintenance.
- *Improved software understanding* achieved by improving the capture and dissemination of requirements, design, and implementation decisions.

Improvements in the V&V process and its incorporation in the software development process based upon the results of the AAV&V-II program will result in reduced software development and maintenance cost and effort and result in more robust and reliable software.

1.2.2 Structure and Tasks of the AAV&V-II Program

The AAV&V-II program began with the design and development of an advanced prototype for the V&V of Ada software. This prototype, which extended the proof-of-concept prototype developed in the original AAV&V program, required the design and implementation of algorithms for interprocedural analysis of software. Once the advanced prototype for software written in Ada was complete, the prototype was extended to additionally support the analysis of software written in JOVIAL. Besides developing these prototypes, known as ImpACT (**Imp**act **A**nalysis **C**apability **T**ool), a survey and evaluation of methods involving automated support for software testing

was conducted. Based on that survey and evaluation, an approach for automated testing support was developed, and the design and implementation of a prototype that was based on that approach was begun.

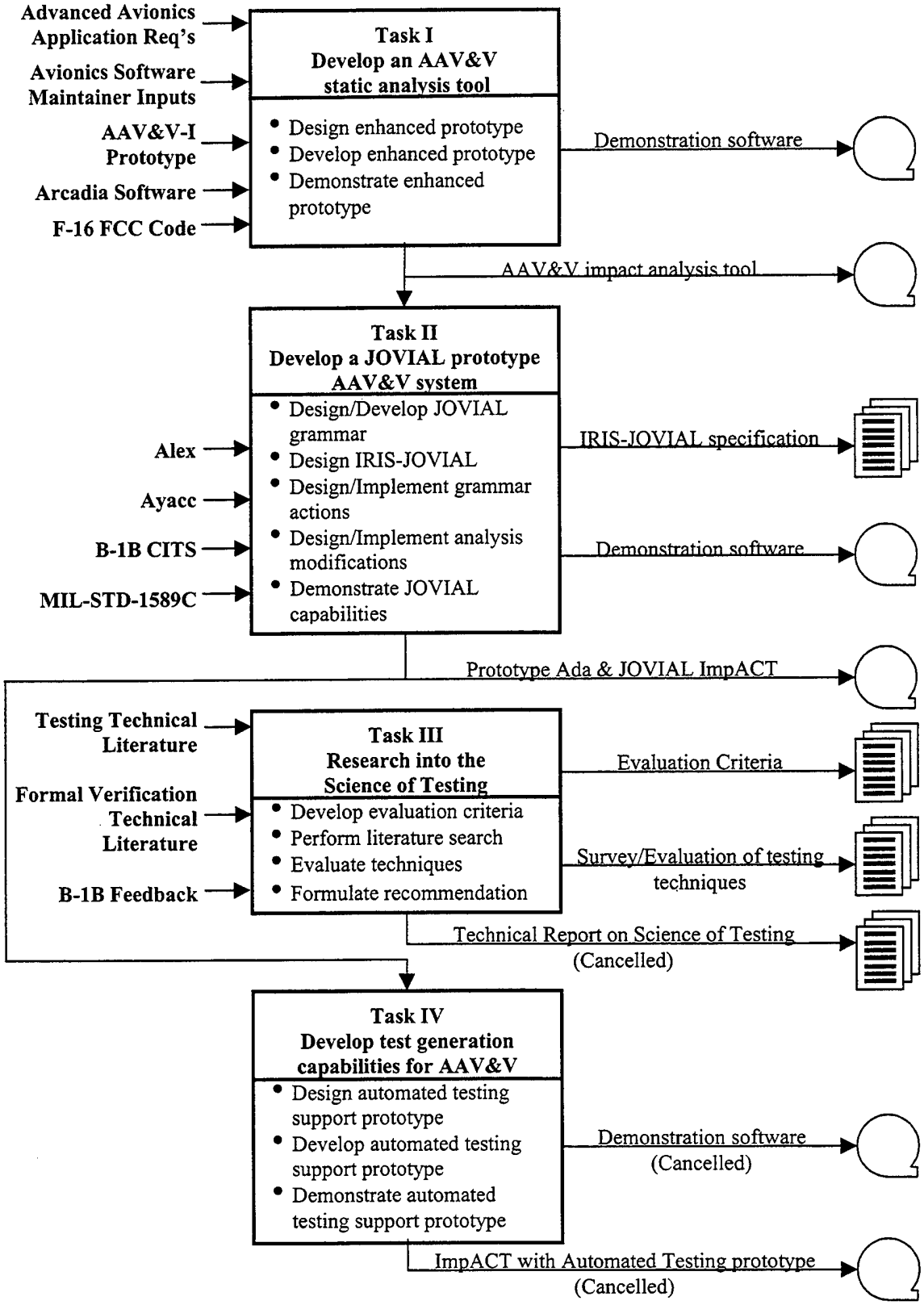

Figure 1-1 Overview of AAV&V-II Tasks

The AAV&V-II program consisted of the following four tasks, illustrated in Figure 1-1:

- In Task I, the proof-of-concept V&V prototype that was developed under the original AAV&V program was used as a starting point. That proof-of-concept prototype provided some limited capabilities for procedural-based analysis of Ada source code. The prototype was extended by providing additional views of the software, most of which were interrelated using hyperlinking. In addition, TASC designed and implemented algorithms for performing interprocedural analysis of source code. The resulting prototype was tested using some simple code fragments and a small binary search program. In addition, the language processor and control flow graph generator were tested on over 1000 lines of source code from the Flight Control Computer of the F-16.
- In Task II, the prototype that was developed in Task I was modified to support software written in JOVIAL. First, an LR(1) grammar was developed for the MIL-STD-1589C definition of JOVIAL by referencing the BNF (Backus-Naur Form - a language for expressing language syntax) in that document. Once the grammar was developed and a majority of the conflicts were eliminated, JOVIAL language features were compared and contrasted with Ada language features. Next, sample IRIS graphs were sketched for the JOVIAL language features. After review by the developers of IRIS at The University of Massachusetts, grammar actions were added to a majority of the grammar to generate the sketched IRIS graphs. Because some JOVIAL language features have no closely corresponding Ada language feature, it was necessary to make some minor modifications to some of the other components developed for Ada programs. The resulting JOVIAL capability was demonstrated on some small sample programs, as well as some code from the B-1 CITS JOVIAL software (the ICE and LDGR modules that were 600 and 1750 lines of procedure code, respectively). The CITS software required some modification to address the prototype limitation that source code be contained in one file, as well as to address some features of the dialect of JOVIAL that was used that are not in strict MIL-STD-1589C JOVIAL. In analyzing this code, it was necessary to make some improvements to the interprocedural dependence analysis algorithms.
- In Task III, TASC located and reviewed literature involving automated support of testing and of formal verification. Then, a set of criteria were developed to evaluate the methods discussed in the literature. Based on the literature survey and the evaluation results, an approach to automated support for testing implemented software modifications was formulated. This approach combined methods from the areas of selective regression testing, test coverage analysis, and automated test data generation. A technical report detailing the methods, the evaluation criteria and

evaluations, and the recommended approach was drafted. Before the draft could be refined and delivered, effort on this task was redirected to Task II - TASC is therefore including the contents of that draft in this report.

- In Task IV, TASC commenced design and experimental implementation of a prototype that would implement the recommended approach to automated support of software testing. While investigating using IRIS graphs to determine changes made to the source code, effort on this task was redirected to Task II. Therefore, the result of Task IV is some refinement of the approach that was recommended in Task III.

1.3 SUMMARY OF RESULTS

This section provides a summary of five major results of the AAV&V-II program: the interprocedural analysis algorithms, the prototypes for both Ada and JOVIAL software, the survey and evaluation of automated testing methods, and the recommended approach for automated support of testing implemented software changes.

1.3.1 Results of Interprocedural Analysis Algorithm Development

The prototype developed in Task I provides an interprocedural analysis of a software system. The algorithms for this analysis were originally to be designed and implemented by developers of ProDAG at The University of California, Irvine. When it became apparent that they would not have the work completed when needed by the AAV&V-II program, TASC undertook the task of designing and implementing algorithms to perform the interprocedural analysis. It is our belief that algorithms for the interprocedural analysis performed by ImpACT did not previously exist. The original design of the algorithms was based on the algorithms for procedural analysis, and improvements in both speed and precision were made to some of those algorithms as improvements became necessary while performing Task II. Significant improvements in speed and precision can still be made to several of the algorithms. The algorithms provided the following kinds of interprocedural analysis:

- *Reachability* - what statements may be executed following execution of a selected statement.
- *Immediate Forward Domination* - what is the next statement that must be executed following execution of a selected statement.
- *Forward Domination* - what statements must be executed following execution of a selected statement.
- *Data Dependence* - what statements use the variables whose values may be changed by a selected statement.

- *Control Dependence* - what statements will have their execution determine by the condition in the selected control statement.
- *Syntactic Dependence* - what statements, through a combination of data and control dependence, will be impacted by a change to the selected statement.

The algorithms enable ImpACT to have the following key capabilities:

- Determining where values assigned to a variable in a chosen statement are used.
- Determining the increase in dependence complexity metrics that results from implementing a specific modification.
- Determining the portions of the system that can be affected by a change, and which therefore must be retested.

The algorithms satisfy the following properties:

- *Extensibility.* The algorithms work using language-independent representations of programs, so additional languages may be supported with no effort.
- *Support for ALC software maintainer needs.* The original AAV&V proof-of-concept prototype provided only procedural analysis. Analyzing the program on a procedural basis is insufficient because it is the interprocedural relationships that tend to become more complex as the size of the software system increases. The interprocedural analysis provides the maintainer with a complete view of what could be affected by a modification and must therefore be retested.
- *Advancement of the state-of-the-art in V&V techniques.* The algorithms significantly contributed to the improvement of V&V techniques, particularly in the use of program dependence analysis to detect the effects of software modifications throughout a program.
- *Reuse of third-party software.* The algorithms were originally designed and developed by adapting the procedural dependence algorithms found in ProDAG. While many of the algorithms have been significantly modified from the ProDAG algorithms on which they were based, there is still a strong resemblance to the original procedural analysis algorithms.

1.3.2 Results of Advanced Prototype Development for Ada Programs

The advanced prototype for the V&V of Ada programs, developed in Task I, implemented the recommendation in the final technical report for the original AAV&V program involving development of a production-quality AAV&V system. This was done by:

- Expanding the language processing front-end support of the Ada language.

- Enhancing the analysis components to support additional Ada language features.
- Improving the dependence analysis capabilities so that they are less conservative through development of new interprocedural analysis algorithms.
- Improving the display and navigation of large programs through the hyperlinking capability of ImpACT and the organization of the information presented in the views.

The ImpACT prototype supporting Ada programs satisfies the following properties:

- *Basis for a complete AAV&V system.* Although the advanced prototype still implements only a subset of the AAV&V methodology as developed during the original AAV&V program, it is extensible. It uses a language-independent representation of programs, therefore requiring minimal effort to add additional techniques or to support additional languages.
- *Demonstration of AAV&V concepts.* The ImpACT prototype demonstrates a significant component of a completed V&V system, and has been applied, in part, to an actual avionics OFP that is implemented in Ada.
- *Support for ALC software maintainer needs.* ImpACT supports the identification of the effects of a software modification, which was one of the common areas mentioned by avionics software maintainers. This information can be used in many ways by the maintainer, including the selection of regression tests to be rerun. In demonstrating this prototype to software maintainers, they were impressed with how it could support them in a sizeable portion of their activities.
- *Advancement of the state-of-the-art in V&V techniques.* Significant contributions were made to the improvement of V&V techniques, most notably in the area of interprocedural analysis algorithms and the use of the generated information in the software maintenance process. Also worthy of mention is the hyperlinking between the many views of the software, some textual and some graphical, available to the maintainer.
- *Reuse of third-party software.* As with the original proof-of-concept prototype, we continued to incorporate existing software into ImpACT where appropriate. This continued to minimize our duplication of other researchers' efforts, and simplified the development of a sophisticated user interface.

1.3.3 Results of Prototype Development for Programs Written in a Subset of JOVIAL

While ImpACT is useful for Ada programs, even greater need existed for JOVIAL programs. This was true because of the lack of tools to support JOVIAL

developers and maintainers. Based on this need, the advanced prototype for Ada programs was modified in Task II to add support for a sizeable subset of the MIL-STD-1589C JOVIAL language definition. Other than not supporting the entire language definition, the version for JOVIAL possessed the same capabilities as the version for Ada. The ImpACT prototype for JOVIAL satisfies the following properties:

- *Basis for a complete AAV&V system.* Although the prototype implements only a subset of the AAV&V methodology as developed during the original AAV&V program and only supports a subset of one JOVIAL dialect, it is extensible. It uses a language-independent representation of programs, therefore requiring minimal effort to add additional techniques, to supplement language coverage, and to support additional languages. In fact, this effort demonstrated that no modification of the analysis and user interface components may be necessary to support additional languages.
- *Demonstration of AAV&V concepts.* The ImpACT prototype demonstrates a significant component of a completed V&V system, and has been applied to subsystems of an actual avionics OFP that is implemented in JOVIAL - the B-1B CITS ICE and LDGR subsystems.
- *Support for ALC software maintainer needs.* ImpACT supports the identification of the effects of a software modification, which was one of the common areas mentioned by avionics software maintainers. This information can be used in many ways by the maintainer, including the selection of regression tests to be rerun. Software maintainers were impressed when the capabilities were demonstrated on some code that they provided. This version of ImpACT also meets maintainer needs by providing them with a desperately needed tool, given the lack of tools to support development and maintenance of JOVIAL software.
- *Advancement of the state-of-the-art in V&V techniques.* Significant contributions were made to the improvement of V&V techniques, most notably in the area of interprocedural analysis algorithms and the use of the generated information in the software maintenance process. It was during development of this version of the prototype that the original interprocedural analysis algorithms were improved. Also worthy of mention is the hyperlinking between the many views of the software, some textual and some graphical, available to the maintainer.
- *Reuse of third-party software.* As with the original proof-of-concept prototype, we continued to incorporate existing software into ImpACT where appropriate. This continued to minimize our duplication of other researchers' efforts, and simplified the development of a sophisticated user interface.

1.3.4 Results of the Survey and Evaluation of Automated Testing Support

In Task III, TASC conducted a thorough search of the technical literature to identify promising techniques for automated support of software testing. This search was expanded to also include literature in the area of automated support for formal verification. In order to compare, contrast, and categorize the techniques, a set of evaluation criteria, developed in the original AAV&V program, was refined to address the automated testing and formal verification of software with emphasis on real-time, mission-critical software. These evaluation criteria were then applied to the techniques uncovered during the survey. The results of this survey and evaluation allowed TASC to formulate an approach to automated support of V&V of implemented software changes. The major results of the survey, evaluation, and recommendation are as follows:

- A listing of literature references related to automated software testing and formal verification (included in the references in section R).
- A detailed summary of each relevant method reviewed in course of the literature search.
- A set of criteria (included in Appendix A) for the evaluation of the surveyed techniques, refined from the criteria used in evaluation V&V techniques in the original AAV&V program.
- A detailed evaluation of the techniques based upon the criteria (included in Appendix B).

The evaluation of the surveyed techniques indicated that no one technique is sufficient to address the needs of avionics software testers. Therefore, TASC recommends a novel approach that combines methods involving selective regression testing, test coverage criteria, and automated test data generation.

1.3.5 Results of Formulation of Approach Supporting Automated Testing of Implemented Software Modifications

As discussed above, a combination of selective regression testing, test coverage criteria, and automated test data generation was formulated as the recommended approach to the V&V of implemented software changes. The test coverage criteria component exists to ensure that some acceptable level of testing has been performed based on the software modifications that have been implemented. This will detect situations in which the modification requires that new test cases be developed. When new tests need to be developed, the automated test data generation component can be invoked by the tester to attempt to automatically generate the required new test cases. The selective regression testing component will cause tests to be executed that cause code that is potentially impacted by the modification to be executed. To determine what code is being executed by the tests, the code must be instrumented to track the

execution path. In situations where instrumenting the code would interfere with the real-time aspects of the software or would result in the executable no longer fitting in memory, selective regression testing is not attempted. Instead, the automated test data generation component is used to generate all test cases that will be involved in the testing of the modification. In this way, TASC has formulated an approach that will meet the avionics software maintainers needs by:

- Ensuring that some minimum level of testing is performed
- Running only tests that have the ability of exposing a fault that was introduced in the modifications.
- Detecting when new tests must be generated.
- Attempting to automatically generate any required new tests.
- Providing the ability to perform a variation of selective regression testing that takes real-time aspects and resource limits, common in avionics software, into consideration.
- Providing tracking of the entire selective regression testing process.

Through this novel approach to software testing, the needs of avionics software maintainers could be met efficiently with extensive automation.

1.4 RECOMMENDATIONS

The prototypes for ImpACT, one each for Ada and JOVIAL, progress the AAV&V system that was envisioned in the original AAV&V program. As software systems get more complex and the importance of software in critical systems increases, automated V&V and a completed AAV&V system is going to be increasingly important in the avionics software life cycle. Additional investment in continued development of the AAV&V system in recommended.

The current state of ImpACT requires some additional work before it can provide the support of software maintainers and testers that is consistent with its potential. Specifically, the following areas need to be addressed:

- Complete language coverage of MIL-STD-1589C JOVIAL, as well as any other desired dialects.
- Improve the interprocedural analysis algorithms to be more efficient, in both time and resources, and to be more precise.
- Enhance graphical support for large programs.

In addition, TASC recommends the investigation of the following areas to further develop and exploit the results of the AAV&V and AAV&V-II programs:

- Support for additional languages, such as C/C++, ATLAS, and VHDL.
- Formal verification for newly developed software
- Design and implementation of the recommended approach to the V&V of implemented software changes.

TASC also recommends returning to avionics maintainers with the AAV&V system and unimplemented concepts, and discussing other advanced tools that could be developed to support them in their difficult efforts.

1.5 REPORT ORGANIZATION

This document, the AAV&V-II final report, documents the results of the AAV&V-II program effort. The document is structured as follows:

- Chapter 1 provides an introduction to the program goals and a summary of the key results.
- Chapter 2 describes the design and implementation of the interprocedural analysis algorithms. First, the chapter provides a brief overview of the history of the development of the algorithms. This overview is accompanied by a brief description of the code used to test and demonstrate the algorithms, as well as by some timing information that was captured. Then, each of the interprocedural analysis algorithms is discussed in detail, including the meaning of the particular kind of dependence, the original algorithm that was adapted from ProDAG, and any improvements that were made to the original algorithm.
- Chapter 3 describes the design and implementation of an advanced prototype supporting the V&V of programs written in Ada. This prototype was an extension of the prototype developed in the original AAV&V program. First, the chapter describes the design of the prototype. Next, the chapter describes the implementation of the prototype. Finally, the chapter describes the software upon which the prototype was tested and demonstrated.
- Chapter 4 describes the design and implementation of a prototype supporting the V&V of programs written in a subset of JOVIAL. This prototype required no modification of the analysis components and the user interface. First, the chapter describes the design of the prototype. Next, the chapter describes the implementation of the prototype. Finally, the chapter describes the software upon which the prototype was tested and demonstrated.
- Chapter 5 describes the survey and evaluation of automated testing methods. First, the chapter discusses the reference sources used during the literature search phase of the survey. Next, the chapter presents an overview of various methods surveyed, grouped by specific categories. The chapter then

describes the set of evaluation criteria used in evaluating the various methods. The chapter closes by providing a recommendation for an approach to the verification and validation of implemented software changes.

- Chapter 6 describes the design effort and the implementation investigation that was performed with respect to the prototype for the verification and validation of implemented software changes.
- Chapter 7 provides a summary of the work performed during all phases of the AAV&V-II program, conclusions drawn from the work, and a set of recommendations for future work based upon the AAV&V-II program.
- Appendix A contains the list of criteria used to evaluate the automated testing methods.
- Appendix B contains the database of surveyed automated testing approaches along with their detailed evaluations based on the criteria listed in Appendix B.
- Appendix C presents issues that must be addressed when porting the AAV&V-II prototypes to other platforms.
- Appendix D describes the environment in which the AAV&V-II prototypes were developed and lessons learned during development.

2. THE INTERPROCEDURAL ANALYSIS ALGORITHMS

Task I of the AAV&V-II program involved development of an advanced AAV&V prototype that was based on the proof-of-concept prototype developed in the original AAV&V program. The advanced prototype was designed to provide more sophisticated support for avionics software maintainers than the proof-of-concept prototype. In the proof-of-concept prototype, interprocedural dependence was handled at a procedural level. By this, we mean that procedures that would be affected by a change would be identified, but the analysis would not descend to the level of individual statements affected within the procedures. For the advanced prototype envisioned for the AAV&V-II program, TASC wanted to provide this fine level of detail to the maintainer and tester.

ProDAG, developed by researchers at the University of California, Irvine, provided this level of detailed analysis within a single procedure, but did not cross procedure boundaries. These researchers had plans to extend the ProDAG analysis so that it would give the level of detail TASC desired at an interprocedural level, crossing procedure boundaries. However, as the AAV&V-II program needed this analysis technology and it became apparent that the ProDAG enhancements would not be available in a suitable time, TASC undertook the task of designing and developing the algorithms. These algorithms were initially straightforward modifications to the procedural algorithms that were present in ProDAG. During Task II, however, it became necessary to improve the efficiency and precision of the algorithms. The improvements currently exist in the JOVIAL version of the prototype, but have not been incorporated yet into the Ada version of the prototype. This chapter provides a description of the interprocedural analysis algorithms that are presently used in ImpACT, with some details of the progress from the original algorithms to the present ones.

ImpACT provides a number of different kinds of interprocedural analysis. These kinds of interprocedural analysis are:

- Interprocedural Control Flow Graph
- Reachability
- Immediate forward dominators
- Forward dominators
- Data dependence
- Strong control dependence
- Weak control dependence

Table 2-1 Time to Perform Selected Kinds of Analysis

Interprocedural Dependence Type	ICE (min.)	LDGR (min.)
Interprocedural Control Flow Graph	0.05	0.08
Reachability	3.12	100.93
Immediate Forward Dominators	0.37	1.58
Forward Dominators	0.47	2.03
Direct Data Dependence	37.22	*
Direct Weak Control Dependence	44.28	*
Weak Syntactic Dependence	2.67	*

* This analysis could not be completed.

- Indirect dependences (including syntactic dependencies)

Table 2-1 presents the time involved in generating the analysis information for two subsystems of the B-1B CITS, which is written in JOVIAL. One subsystem, ICE, is about 600 lines of procedure code in length. The other subsystem, LDGR, is approximately 1750 lines of procedure code in length. Note that we have not been able to complete the data and control dependence generation for LDGR because of sizeable memory leaks in the original ProDAG algorithms that have not yet been located and fixed.

The remainder of this chapter details each of the above analysis algorithms. It also describes some modifications made to the Pleiades object management system to increase the speed of the analysis.

2.1 THE INTERPROCEDURAL CONTROL FLOW GRAPH

The Interprocedural Control Flow Graph (ICFG) is the set of Control Flow Graphs (CFGs), one for each function and procedure, connected with call-return edges. A call edge links a node that calls a procedure or function with the start node of the CFG for that procedure or function. A return edge links the end node of each CFG with a node that calls the procedure or function represented in the CFG. Any CFGs for the program that are not reachable from the main routine of the program by a sequence of procedure and function calls are pruned from the ICFG.

In adding the call-return edges, two pieces of information are used. The first is how many times the CFG has been called in other CFGs that have been processed. The second is whether this CFG has been processed. Initially, the main routine is considered to have been called once. The algorithm chooses a CFG that is called at least once and has not yet been processed. This graph is then traversed, with call-return edges added for functions and procedures that are called and the number of times those

functions and procedures have been called is updated. When there are no more CFGs that are called at least once but have not been processed, all the CFGs that have not been called are removed from the ICFG.

One potential problem with this approach is that the call is matched with the subprogram being called by the name of the function or procedure. We expect that this may cause problems in the presence of overloaded subprograms within the same package, although we have not attempted any such experiments. Therefore, TASC recommends experimenting with code that has such overloading and, if a problem is exposed, linking call nodes with CFGs using not only the subprogram name but also the types of the actual and formal parameters.

2.2 THE REACHABILITY ALGORITHM

One statement S_1 interprocedurally reaches another statement S_2 if and only if, during execution of the program, statement S_2 can be executed after statement S_1 has already been executed. Initial attempts at this algorithm involved performing different variations of a transitive closure on a copy of the ICFG. These variations included:

- Performing a forward closure from each node, where a forward closure is calculated by traversing the subgraph in a forward direction starting from the chosen node.
- Perform a transitive closure that centers around the start node and end node of each CFG in the ICFG.
- Perform a transitive closure on a matrix representation of the graph, handling closures through an end node of a CFG as a special case.

This same transitive closure algorithm, applied to different graphs, was used in computing all of the indirect dependence graphs as well.

The first of the algorithms above is conceptually fairly simple. It connects parents of the selected node to children of the selected node, places each child onto a list of nodes, and then selects a new node from that list. Some special processing may be performed to attempt to prevent edges that indicate that a procedure `A` can call a procedure `B`, and have procedure `B` return to some other procedure, such as procedure `C`. However, there are problems with this algorithm that cause some edges to be placed in the graph in situations where reachability is not possible. The algorithm can also be very slow because of the time involved in performing queries as the size of the graph increases.

The second of these algorithms has three steps. The first step is to determine reachability within the procedure, using the same algorithms that were used in the procedural reachability for ProDAG. The second step has two parts:

- Any nodes that reach a call node reach the start node of the CFG for the function or procedure being called.
- Any nodes that reach the start node of a CFG reach any nodes that can be reached from that start node.

The third step also has two parts:

- Any nodes that can be reached from the statement that is returned to can be reached by the exit node of the associated CFG.
- Any nodes that reach the exit node of the CFG reach any nodes that the exit node reaches.

The second and third steps must be repeated until one iteration results in no further additions to the graph being generated. This algorithm has many of the same problems as the first algorithm - that it is conservative, and that the queries can become very slow as the size of the graph increases.

The third algorithm involves populating an n-by-n matrix using the contents of the ICFG, where n is the number of nodes in that graph. An algorithm similar to Warshall's algorithm is used to compute a full transitive closure. However, because the algorithm was structured differently and treated the end nodes of CFGs as a special case, the algorithm had to be repeated until an iteration resulted in the addition of no new dependence edges. This graph was still imprecise, and the handling of end nodes as a special case was removed to speed the algorithm. Besides the imprecision, this algorithm requires an amount of memory that is quadratic in the size of the program, and we found that the addition of dependence edges became slow as the edges in the graph increased. The time to insert edges was addressed by enhancements to the Pleiades object management system, which are discussed in section 2.9.

The current algorithm was designed near the end of the AAV&V-II program, and we believe it to result in a precise reachability graph. The algorithm was actually based on our interprocedural forward dominators graph, and the observation that the interprocedural forward dominator graph is simply the interprocedural reachability graph with the restriction that the statement must occur in every path as opposed to just one. The algorithm traverses the ICFG starting from the exit node of the graph, modifying the state for each node visited, placing nodes on a pending node list, and choosing the next node to be visited.

The state for each visited node is essentially a list of nodes that, to this point in the algorithm, are believed to be reachable from that node. If the node for which reachability is being determined is not the end node of a CFG, then the states of all the

children nodes are unioned together in a way such that the call hierarchy is taken into consideration. This is done by including in the union only nodes from CFGs that are lower in the call hierarchy than the called function or procedure if the child whose state is being unioned is the start node of a CFG. The merging of the state of a start node is handled in this way to avoid the problem of dependences that would require a function or procedure to return to a location other than from where it was called. If the node for which reachability is being determined is an end node of a CFG, then the state contains the nodes to which control is returned, unioned with the states of the statements that follow the statement to which control is returned.

Parents of a node are added to a list of pending nodes in two situations. The first situation is when there is more than one parent - both nodes cannot possibly be handled at the same time, so they are placed on the list and a node to processes next is selected from the list. The other situation is when a node has one parent but multiple grandparents. This is done because the node could already be on the list, in which case it would not need to be processed again at this time.

There are two situations where a node is chosen from the list. One is when nodes were placed on the list, as described above. The other is when the state that is built for the present node did not change from the state that it previously had. This means that none of its predecessors will have their state changed based on the new state calculation, so a node that still needs to be processed should be chosen from the list. The algorithm completes when a node is to be chosen from the list, but the list is empty. At that time, the state of each node contains precisely the nodes that are reachable from the node (plus the node itself), and the information for this dependence graph can be written to the active repository.

TASC recommends further investigation of improvements to the interprocedural reachability algorithm. As can be seen in Table 2-1, the time it takes to perform this particular analysis grows at an undesirable rate. We believe that some additional refinement of the algorithm, combined with improvement of the Pleiades object management system, would improve this performance.

2.3 THE IMMEDIATE FORWARD DOMINATORS ALGORITHM

The immediate forward dominator of a statement S is the next statement that **must** be executed after executing S. Each statement, with the exception of the last statement of the program, has exactly one immediate forward dominator.

Edges appear in the immediate forward dominators graph only for decision nodes. A decision node is any node that has more than one child in the same CFG in which it occurs. Again, a state is maintained, nodes are occasionally placed on a list of

pending nodes, and nodes are chosen from the list of pending nodes. The situations in which nodes are placed on the list and chosen from the list are identical to those for the reachability graph. However, edges are added to the graph as nodes are identified as decision nodes instead of waiting until the end of the algorithm as was done for the reachability graph.

If the current node is a decision node, then any existing immediate forward dominator graph edges leading from it are removed from the graph. The state, which is calculated using the intersection of all the children of the node, is a list, and the first element of that list will be the first statement following the merging of control branches. Therefore, that node is the immediate forward dominator of the decision node.

This algorithm could potentially be more efficient if the edges were added only at the conclusion of the algorithm. However, the algorithm is already relatively fast for the source code analyzed thus far.

2.4 THE FORWARD DOMINATORS ALGORITHM

The forward dominators of a statement S is the set of **all** statements that **must** be executed after executing S. Every statement, except the last one in the program, will have at least one forward dominator.

Just as with the reachability and immediate forward dominator algorithms, a state is maintained, nodes are occasionally added to a list of pending nodes, and nodes may need to be selected from the list of pending nodes. The situations in which nodes are placed on the list and chosen from the list are identical to those for the reachability graph. In analyzing the original algorithm during Task II, it was noticed that some interprocedural forward dominator dependences were not being generated. The algorithm was revised to correct this problem, and we will describe the modifications that were made. At the same time, some modifications were made that resulted in a dramatic speedup of the algorithm.

The original algorithm used the intersection of the states of the children of a node, plus the node itself, as the state of the node. Any forward dominator edges from the node were removed, and edges from the node to each node in the intersection of the states of the children were added to the forward dominator graph.

The problem with this algorithm is that a node that called a procedure or function would not have any of the nodes from within the calls in its state. That is because the node could be reached when returning from the call, and the nodes inside the called procedure or function therefore would not necessarily forward dominate this node. However, those nodes preceding one that calls a procedure or function is likely

to have nodes within that procedure or function that forward dominate it. Edges to these nodes were missing, since only the states of the children were being considered.

To correct this problem, the algorithm was modified. This new code is executed only if the current node is not the end node of a CFG. First, all the children of the current node are located. Then, for each child that is not the end node of a CFG, the call-return edges from that node are determined. For each of the call-return edges, the state of the node that the edge leads to is unioned into the state for the current node (which is initially created as described for the original algorithm). The effect is that if the current node has children that call procedures or functions, then the state of the start nodes of each of CFGs for each of those procedures and functions is added to the state of the current node.

One other modification was necessary, which would force forward dominators of nodes immediately preceding calls to a procedure or function to be recalculated whenever the forward dominators of the start node for the corresponding CFG were modified. This was done by placing the grandchildren of the start node of a CFG on the list of pending nodes, unless that grandchild was the end node of a CFG. If the grandchild is the end node of a CFG, then it is a node inside a function or procedure that the child calls and therefore does not need to be recalculated.

The speed of the algorithm was improved by noting that the state contained all the information necessary to build the graph, and that this state existed at the end of the algorithm. By simply maintaining the state during the algorithm and adding edges to the graph only when no more modifications to the state were possible, it eliminated the need to repeatedly remove edges and add edges to the graph. This simple improvement of the algorithm resulted in a reduction of sixty times in the time necessary to generate the graph.

2.5 THE DATA DEPENDENCE ALGORITHM

Before worrying about the algorithm to generate the dependences, how variable definition and use information involved in parameter passing needed to be addressed. With respect to statements that involve calls, the definitions include the formal parameters that receive values in the calling process and actual parameters whose values may be changed as a result of the call. In the case of a function call, the variable in which the return value is stored is also considered defined. With respect to statements that involve calls, the uses include the actual parameters that are used to initialize the formal parameters, and the formal parameters to which a modification of the value may also modify the value of the corresponding actual parameter.

There is one problem with this approach. Inside the procedure or function that is being called, parameters that are initialized using the corresponding actual parameter are considered to be defined in the declaration region of the procedure or function. This means that a change to a variable that is an actual parameter would not show that the calculations within the called function or procedure are impacted. The solution appears to be that formal parameters should not be considered to be defined within the function or procedure, but this solution will also mean that a new approach to the indirect graphs will need to be implemented. Note that this solution has not yet been implemented, but TASC recommends that it be examined more closely and, assuming it does solve the problem, implemented.

The actual data dependence algorithm is simply an adaptation of the procedural data dependence algorithm of ProDAG. It was modified to work on an ICFG as opposed to a CFG. This resulted in a small change in some of the utility functions, mainly to handle the correct type of parameter. Changes also had to be made to determine the CFG that a node is in, where the graph was known in the original algorithm. This algorithm to generate the data dependence is actually very complicated and appears to perform poorly for larger programs. TASC recommends examination and refinement of this algorithm to increase its efficiency, both in the time it takes and the resources it uses.

2.6 THE STRONG CONTROL DEPENDENCE ALGORITHM

The strong control dependence algorithm is simply an adaptation of the procedural strong control dependence algorithm of ProDAG. It was modified to work on an ICFG as opposed to a CFG. This resulted in a small change in some of the utility functions, mainly to handle the correct type of parameter. This algorithm to generate the strong control dependence is actually very complicated and appears to perform poorly for larger programs. TASC recommends examination and refinement of the algorithm to increase its efficiency, both in the time it takes and the resources it uses.

2.7 THE WEAK CONTROL DEPENDENCE ALGORITHM

The weak control dependence algorithm is simply an adaptation of the procedural weak control dependence algorithm of ProDAG. It was modified to work on an ICFG as opposed to a CFG. This resulted in a small change in some of the utility functions, mainly to handle the correct type of parameter. This algorithm to generate the weak control dependence is actually very complicated and appears to perform poorly for larger programs. TASC recommends examination and refinement of the algorithm to increase its efficiency, both in the time it takes and the resources it uses.

2.8 THE INDIRECT DEPENDENCE ALGORITHMS

Generating the indirect graphs presently involves making a copy of the direct graph (for example, the data dependence graph) and performing the closure algorithm on it. The present closure algorithm involves using a matrix of nodes and dependence values, and is described in section 2.2.

To generate the syntactic dependence graphs, the data dependence graph and the desired control graph have their dependence edges unioned into a new graph. Then, that graph has the same closure algorithm that is used for the indirect graphs applied to it.

The resulting indirect or syntactic dependence graph is conservative, and contains dependencies that cannot possibly exist. The precision of the graph can be improved, we believe to the point of being precise, by considering the reachability graph when performing the transitive closure. Presently, the algorithm says that if S_1 is dependent on S_2 and S_2 is dependent on S_3, then S_1 is dependent on S_3. The problem is that S_1 may not be reachable from S_3, so how could it be dependent on it? The solution, since the reachability graph has been made precise, is to add the condition that S_1 must be reachable from S_3. This enhancement, however, has not been implemented, and would likely be very slow without performance enhancement to the code used to query dependence relations. TASC recommends implementing this modification to increase the precision, followed by increasing the efficiency of the code used to query dependence relations.

2.9 IMPROVEMENTS TO THE PLEIADES OBJECT MANAGEMENT SYSTEM

As we began to work with larger programs, some problems were experienced with the time involved in performing an analysis. We attempted to make improvements by providing a better data structure for accessing dependence information. Some improvements were also made to the code used to insert edges into the graphs. Finally, the method for accessing contents of lists was improved.

Although the original ProDAG software did not make use of it, the Pleiades object management system provides the ability to have keys for relations. Accessing data based on a field for which no key exists results in a linear search. Pleiades provides the ability to specify several types of key data structures, such as binary search trees and hash tables. Since hash tables provide constant time access, we chose these. We decided to keep both the source node of a dependence edge and the target node of a dependence edge as keys. This was expected to significantly decrease the time to access edges, used in the routine to create a new edge, but it appeared to have far less effect than was expected.

Not seeing the speed increase that was expected, we then examined the routine used to create an edge. Before creating an edge, it checked to see if an edge with the same source and target nodes already existed. If it did, then that edge would be reused, and no new edge would need to be created. If it did not, then the edge was created. Through inserting timers into this code, we discovered that most of the time was being spent in creating a subgraph of edges that had the source node and in destroying that subgraph when it was no longer needed. To improve the performance, a function was created that, given a graph, a source node, and a target node, would return the edge from the graph with the indicated source and target nodes if such an edge existed. This function first gets a list of edges that have the desired source - since a key for the source nodes was being maintained, this was a constant time operation. Next, the list was searched to see if there was an edge with the desired source - the entire list was searched to make sure that there were not duplicate edges, with an exception raised if there were. The located edge was returned, with `null` being returned if no such edge was found. This improvement resulted in an eleven times increase in speed in the generation of the interprocedural forward dominators graph for a relatively large program.

Some inefficiency was also noted in the way that lists were being accessed. The Reusable Component Library used in Pleiades provided a list package, but accessing elements of the list required traversing the list from the beginning. However, the Pleiades code was basically iterating through the elements of the list which, with the poor capabilities provided in the list package, resulted in quadratic time performance. The ability was provided to maintain iterators into the list, resulting in this iteration through the elements of the list performing in linear time.

TASC recommends further enhancements involving the storage of, and access to, dependence information. We believe that these enhancements will become necessary as the size of the programs to be analyzed increases.

3. THE IMPACT PROTOTYPE FOR ADA

Task I of the AAV&V-II program involved the development of an advanced prototype, based on the proof-of-concept prototype developed in the original AAV&V program, for analysis of programs written in Ada. Our development of this prototype pursued the following goals:

- *Address the needs of the target users* - Design the advanced prototype, based on the proof-of-concept prototype, to address the needs and concerns of avionics software testers and maintainers.
- *Provide the AAV&V system on a popular platform* - Develop the advanced prototype on a commonly-used combination of hardware and operating system.
- *Enhance the basis for AAV&V* - Design the advanced prototype with extension of the prototype in mind.
- *Extend the state-of-the-art* - Through the design and implementation of the prototype, advance the state of V&V techniques, practices, and concepts.

We used the following approach to design and implement the advanced prototype for analyzing programs written in Ada to achieve these goals:

- We based both the proof-of-concept prototype and the advanced prototype on needs expressed in a survey of avionics software maintainers at three Air Logistics Centers (ALCs) that was conducted in the original AAV&V program.
- We targeted the advanced prototype at a platform of a Sun workstation running the Solaris operating system (although the operating system was originally SunOS). We also used The X Window System libraries and Motif libraries, both commonly used, in the design and development of the ImpACT graphical user interface.
- We designed the prototype so that it is extensible to support not only other portions of the methodology that were not implemented, but also to support additional programming languages.
- We utilized Commercial-Off-The-Shelf (COTS) and academic software wherever possible, extending its capabilities where necessary, so that we could concentrate on improving current techniques rather than reimplementing existing capabilities.

The following sections describe the design, implementation, and enhancement of this advanced prototype. First, we describe the design of the advanced prototype, including all features in the final implementation. Second, we describe the

implementation of the final advanced prototype. Third, we describe the software upon which the prototype was demonstrated and the demonstration itself. Finally, we make recommendations for further efforts with this advanced prototype.

3.1 DESIGN OF THE ADVANCED PROTOTYPE FOR ADA

The advanced prototype, known as ImpACT, for the analysis of programs written in Ada has been designed to provide a basis on which a complete AAV&V system can be built and to illustrate how AAV&V is used to support the avionics software maintenance process. Portions of the AAV&V methodology, for which proof-of-concept prototypes were developed in the original AAV&V program, were enhanced. The advanced prototype addressed determination of the effects of software modification, which can be used to assist in deciding software tests to be performed and in evaluating the thoroughness of the testing performed.

As depicted in Figure 3-1, the ImpACT advanced prototype for Ada includes the following components:

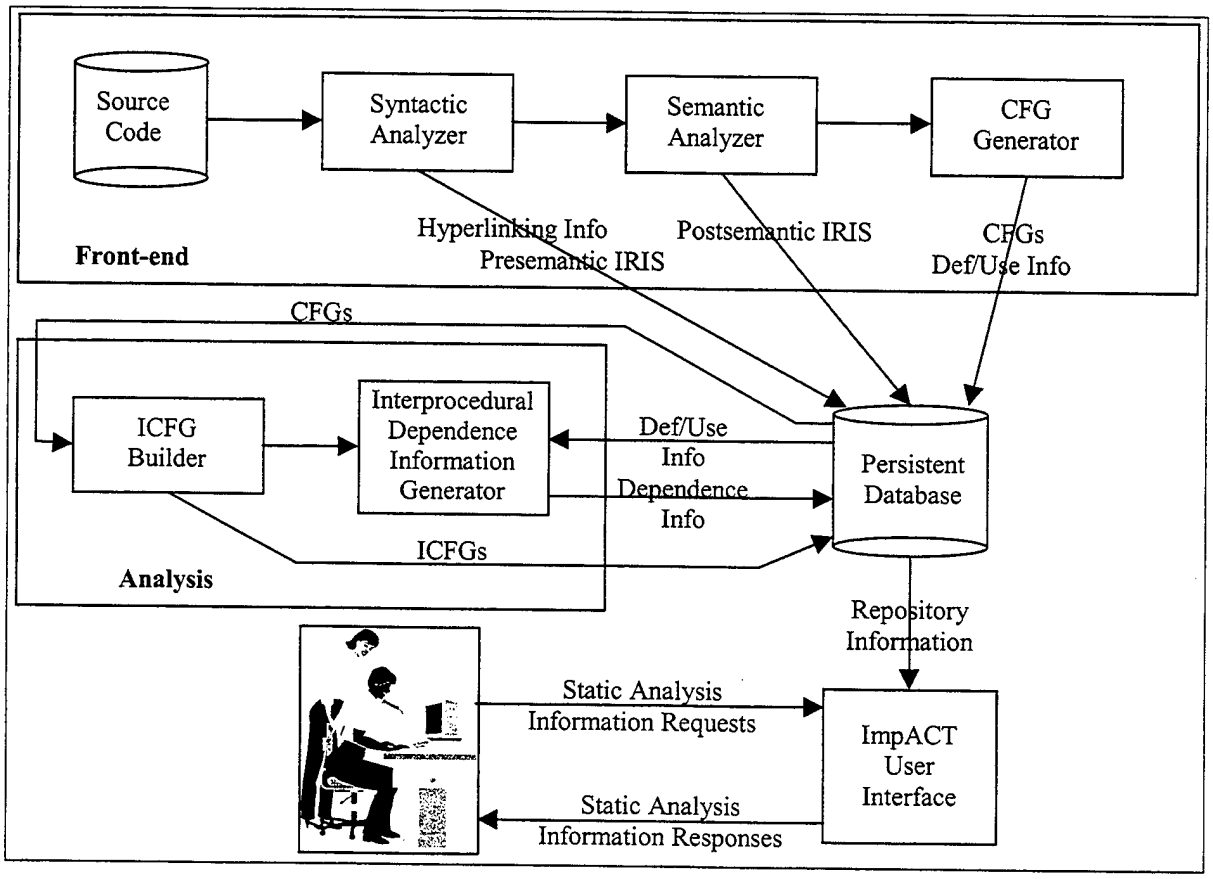

Figure 3-1 ImpACT Design

- A front-end component that converts the source code into an intermediate representation. Included as part of this front-end, for purposes of the advanced prototype, is the generation of CFGs. This generation is included since it must be designed to process the resulting intermediate representation of the source code.
- An analysis component that generates the CFG for the program in its entirety (an ICFG), as well as the interprocedural dependence information.
- A persistent database that stores the information created by both the front-end and analysis components, so that they may be retrieved at a later time without needing to be regenerated.
- The ImpACT user interface, where the user can request and receive information that was stored in the persistent database.

3.1.1 The Front-End Component

The purpose of the Front-End component is to receive the source code of an Ada program as input and translate it into internal representations that are suitable for analysis by the Dependence Analysis component. The information generated by this component is stored by the Persistent Database component. As indicated in Figure 3-1, the Front-End component consists of the following subcomponents: Syntactic Analyzer, Semantic Analyzer, and Control Flow Graph Generator.

The Syntactic Analyzer receives Ada source code, parses it, and generates an internal representation that represents the structure of the program. The internal representation generated by the Syntactic Analyzer is a presemantic IRIS graph, a language-independent internal representation. An IRIS graph is created for each Ada component (i.e., package, procedure, etc.) in the source code. In addition, information is maintained to connect each node in an IRIS graph with the text that resulted in the generation of that node. This information is necessary in implementing the hyperlinking capabilities of ImpACT.

The Semantic Analyzer converts the presemantic IRIS graphs created by the Syntactic Analyzer into postsemantic IRIS graphs that contain not only information about the structure of the source code, but also information about its meaning. This is accomplished by *resolving* identifiers in the IRIS graph. An identifier is resolved by replacing its node in the presemantic graph (which contains only the identifier name) with a node that points to the declaration of the type, package, function, procedure, etc., to which the identifier refers.

The CFG Generator receives as input a postsemantic IRIS graph representing an Ada module and generates a CFG for the module. A CFG is a graph-based representation of the order in which statements are control-dependent on others, i.e.,

their execution depends on whether other statements are executed. The CFG representation is described in Section 3.2.

3.1.2 The Analysis Component

The Analysis component generates the control information for the entire program, as well as dependence information that can be used to determine parts of a program that could be affected by a change to the code. It consists of the following subcomponents: ICFG Generator and Interprocedural Dependence Information Generator.

The ICFG Generator uses the CFGs for the program, generated by the Front-End component, and builds an ICFG for the program. An ICFG contains all the CFGs of the program that can be reached by some combination of procedure and function calls. These CFGs are connected together using a special dependence edge, referred to as a *call-return edge*. The ICFG is described in more detail in Section 2.1.

The Interprocedural Dependence Information Generator uses the ICFG and generates program dependence information, at an interprocedural level, for the program. This program dependence information relates individual statements of the program based on the types of dependences. This program dependence information is described in more detail in Sections 2.2 through 2.8.

3.1.3 The Persistent Database Component

The Persistent Database component stores the analysis information that is generated for analyzed programs. This information includes information about the modification dates and times of the source files, the IRIS graphs for the source files, the hyperlinking information for the various views, the CFGs, def/use information, the ICFGs, and interprocedural dependence information. Storing this information in a persistent database eliminates the need to generate it each time the maintainer wishes to use ImpACT to analyze the same version of the source code for a system.

3.1.4 The ImpACT User Interface

The ImpACT User Interface is used to create the analysis information for an Ada program. Once the source code has been analyzed, the user may request a number of views of the source code. These views are the Source Code Listing, the Declaration Diagram, the CFG, the Data Flow Information, the ICFG, the Program Dependence Graph (PDG), the Statement Dependence Graph, and the Dependence Metrics. Most of these views are connected through a system of hyperlinks - therefore, clicking on something in one view will cause related elements of the hyperlinked views to be highlighted.

The Source Code Listing view lists the source code contained in a file just as it appears in that file. The user can click on a line of code to highlight the node corresponding to that line of code in most of the graphical views, such as the CFG, the ICFG, and the PDG.

The Declaration Diagram View presents a list of declarations that are contained within a chosen file. This view can be used to get progressively detailed information about the contents of a source code file as the contents of that file are expanded.

The CFG view shows a CFG for a function or procedure within the program. This CFG shows the possible execution paths through the function or procedure, giving the maintainer a graphical view of the control structure of that function or procedure. This view is hyperlinked to the Source Code Listing, the ICFG, the PDG, and the Data Flow Information. If the Data Flow Information view is open, clicking on a node in the CFG will cause the Data Flow View to display the defined and used identifiers that occur in the associated statement or statements.

The Data Flow Information view lists information about identifiers that are defined and used in a particular statement or group of statements. Clicking on a CFG node for a statement chooses the statement for which data flow information is displayed.

The ICFG view shows an ICFG for a program. The ICFG shows the possible execution paths through the entire program, giving the maintainer a graphical view of the control structure of that program. The ICFG is built by adding special edges, called *call-return* edges, to the set of CFGs for the program, connecting a statement that makes a call to the CFG for the procedure or function that is being called. This view is hyperlinked to the Source Code Listing, the CFG, and the PDG.

The PDG view shows the dependence relations between statements within the program. The maintainer can choose the type of dependence that is displayed before ImpACT is started. Clicking on a node for a statement in this graph will result in identifying the nodes for statements that are dependent on the selected statement. These nodes are indicated using highlighting, not only of nodes but also of edges between the nodes. This view is hyperlinked to the Source Code Listing, the CFG, and the ICFG.

The Statement Dependence Graph view shows a selected statement and the nodes that are dependent on that selected statement. This view can be opened from the CFG view, the ICFG view, or the PDG view. The nodes of this view hyperlink to most of the other views. In particular, the hyperlinking back to the Source Code Listing can be used to determine the exact lines in the source code that will be impacted by the modification under consideration.

The Dependence Metrics view can be used to determine metrics about dependences in the entire program and in selected procedures and functions. This information can be used to determine which implementation of a modification will result in the least increase, or perhaps the greatest decrease, in the dependences that exist in the program.

3.2 IMPLEMENTATION OF THE ADVANCED PROTOTYPE FOR ADA

The ImpACT prototype for Ada programs consists of four components (Front-End, Analysis, Persistent Database, and User Interface) described in Section 3.1. Each of these components is implemented in Ada, and consists of a combination of third-party software and software developed by TASC. The reuse of existing commercial off-the-shelf (COTS) and third-party software allowed TASC to efficiently develop the prototype and utilize existing technology. The following sections describe the implementation of each of the four AAV&V components including the use of any third-party software in that component. The ImpACT prototype for Ada programs is hosted on a Sun/Solaris environment.

3.2.1 The Front-End Component

The Front End component translates Ada source code into internal representations that are used by the Analysis component to build dependence graphs. The Front End component consists of three components: a Syntactic Analyzer, a Semantic Analyzer, and a CFG Generator. All three of these components are modifications to the Arcadia language processing tools. The remainder of this section provides additional detail on the implementation of each component and the corresponding internal representations that they generate.

The Syntactic and Semantic Analyzers generate an IRIS (Internal Representation Including Semantics) representation of the code for each Ada module. IRIS, developed by the Arcadia consortium, is a language-independent abstract syntax graph representation of a program, and is generated in two steps. First, the Syntactic Analyzer parses the Ada source code and generates a presemantic IRIS graph. Then, the Semantic Analyzer converts the presemantic IRIS graph created by the Syntactic Analyzer to a postsemantic graph by resolving identifiers to point to their declarations. TASC augmented the Syntactic and Semantic Analyzers to also generate information that links nodes in the generated graphs to the portions of the source code which those nodes represent. This information is needed to implement ImpACT's hyperlinking capabilities.

The IRIS internal representation was selected for several reasons. First, since IRIS is language-independent, any analysis tools that operate on IRIS graphs are also language-independent. These analysis tools could be applied to any programming language as long as there is a tool that translates that language into IRIS. Another advantage is that the Arcadia tools support persistence of IRIS graphs (i.e., the graphs can be stored on disk and retrieved for future analysis). Furthermore, the Arcadia software includes several tools that operate on IRIS graphs that we were able to incorporate, with some enhancements, into our system (e.g., the CFG Generator and ProDAG). Moreover, the tools are public domain and easily accessible. However, the Arcadia tools do have some significant weaknesses. In particular, ProDAG was unable to handle the IRIS representation of several Ada constructs, such as those associated with tasking and certain qualified references. Also, while the ProDAG algorithms may perform acceptably for intraprocedural dependence analysis, the performance of several of the algorithms, simply extended to work interprocedurally, was unacceptable for larger programs.

The CFG Generator is an Arcadia tool that receives as input a postsemantic IRIS graph representing an Ada module and generates a CFG for the module. A CFG represents the flow of control through the program code for the module. Each node in the graph represents a statement in the program, and the edges connecting pairs of nodes represent the flow of control from one statement to the next.

3.2.2 The Analysis Component

The Analysis component constructs the interprocedural analysis information for the program being analyzed. Using the graphical views of this information that are presented by the User Interface component, this information may be used to determine the parts of a program that could be affected by a modification. The Analysis component consists of two components: the ICFG Builder and the Interprocedural Dependence Information Generator. Both of these Analysis components are discussed in detail in Chapter 2, and are based on software developed by the Arcadia consortium. TASC, however, has made some substantial improvements to some of the interprocedural analysis algorithms and recommends additional examination of the algorithms. This examination would be likely to result in additional improvements to the algorithms in the areas of time, resource utilization, and precision.

3.2.3 The Persistent Database Component

The Persistent Database component stores the analysis information so that it does not need to be regenerated each time the system is used in the maintenance of the same body of source code. The Persistent Database is implemented using the Pleiades Object Management System, developed by the Arcadia consortium. Pleiades provides tools to develop the object schema in a high-level specification and then generate Ada

source code from the schema. The resulting source code manages the persistence and consistency of the data, and provides subprograms for adding, changing, and finding database information.

Although we have not found it necessary to modify Pleiades directly, it has been found to have significant inefficiencies and some modifications were necessary to some of the underlying reusable software components. Some of these inefficiencies, as well as related modifications that were made, are described in section 2.9. TASC recommends further examination of improvements that could be made that are related to Pleiades and some of its limitations. In particular, some of the graphical views need to determine whether pairs of nodes have some specific relationship. The code that determines this is similar to that used to determine if there is already an edges between two nodes before inserting an edge, indicating that there is a potential for a substantial speedup in the display of the interprocedural dependence graph.

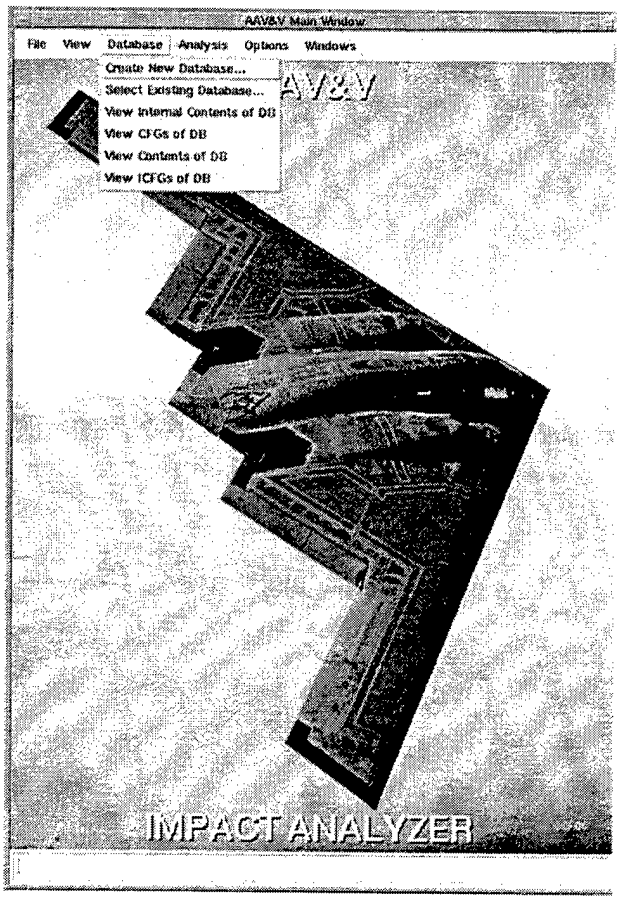

Figure 3-2 The Main ImpACT Window, with Database Pulldown Menu

3.2.4 The ImpACT User Interface

The ImpACT User Interface component is used to request that the analysis information for a program be generated and is used to present that information to the software maintainer using a graphical, hyperlinking interface. The information is presented to the maintainer using various views of the analyzed software. These views are the Source Code Listing, the Declaration Diagram, the CFG, the Data Flow Information, the ICFG, the PDG, the Statement Dependence Graph, and the Dependence Metrics.

When ImpACT is started, the main ImpACT window, shown in Figure 3-2, is presented. On the window are a number of pulldown menus that may be used to create analysis databases, add analysis information to databases, and open or activate the many views of the analysis information that are available.

Requesting Generation of Analysis Information

The ImpACT User Interface component is used to request that analysis information for a program be generated. Before the analysis information can be generated, a persistent database (or repository) must first be created and initialized. This is done using the interface shown in Figure 3-3, which appears when the `Create New Database...` option is chosen from the `Database` pulldown menu of the main ImpACT window. This interface is used to specify the directory path in which the

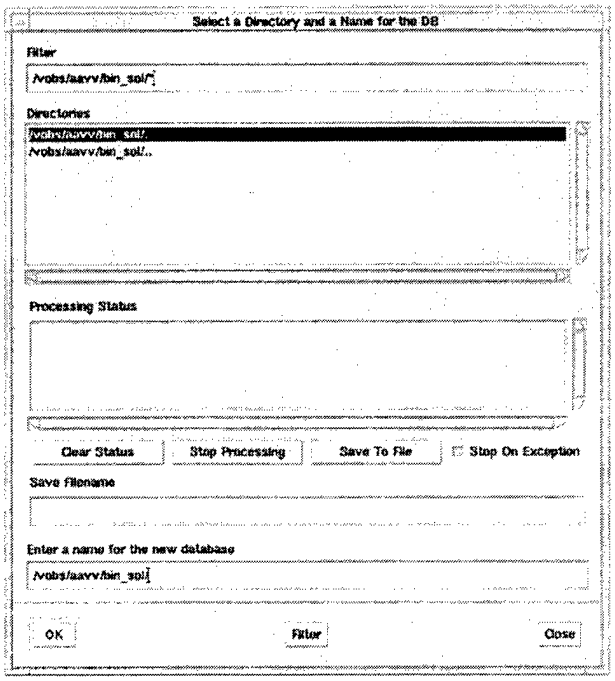

Figure 3-3 Interface for Creating and Initializing a New Database

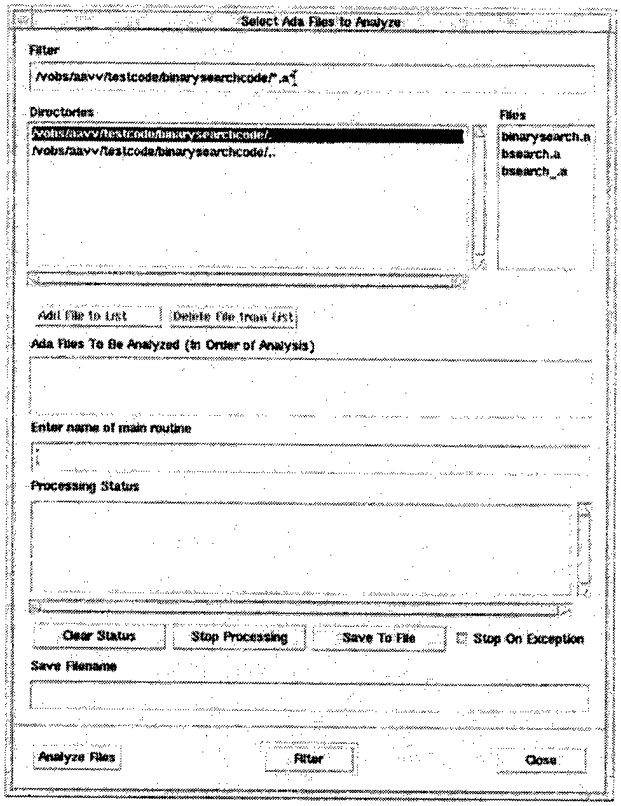

Figure 3-4 Interface for Creating Analysis Information for Ada Program

database should be created as well as the name of the database. When the `OK` button is clicked upon, the database is initialized. Analysis information for some standard Ada packages is then added to the database. During the initialization process, messages are displayed in the `Processing Status` panel of the window, and the user has the ability to save these messages to a file.

To add the analysis information for a program to a database, the user must first select the database to which the analysis information will be added. This is done using a small window that lists directory paths and the databases within those directory paths, which is presented when the `Select Existing Database...` option is chosen from the `Database` pulldown menu of the main ImpACT window. Once the database is selected, the interface shown in Figure 3-4 is displayed by choosing the `Analyze Ada Program...` option from the `File` pulldown menu of the main ImpACT window. This interface is used to select the files for which analysis information is to be created and to indicate the order in which the files are to be processed. In particular, the files must be processed in their compilation order. For example, if file *A* `withs` some compilation unit from file *B*, then file *B* must be processed before file *A* may be processed. Also, since any procedure may be the main procedure of an Ada program, the user must specify the name of the main procedure of the program being analyzed. When the required information has been provided and the `OK` button is clicked upon,

the analysis information is created. During the creation of the analysis information, messages are displayed in the `Processing Status` panel of the window, and the user has the ability to save these messages to a file.

The analysis information that is created is persistent, i.e., is saved to disk. This means that the generated analysis information can be reused, without needing to be regenerated, until the actual source code to be analyzed has been changed. This information includes the IRIS representation of the source code, the CFGs, the ICFGs, the Interprocedural Dependence information, and the hyperlinking information.

Source Code Listing

A window listing the source code of an analyzed file is one of the views of the software available through the ImpACT user interface. This window may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. An example of the Source Code Listing window is shown in Figure 3-5.

The Source Code Listing view is hyperlinked to all of the graphical views. This means that clicking on the main part of a statement, such as the assignment operator (:=), will activate the hyperlinking with respect to each of the above views if they are

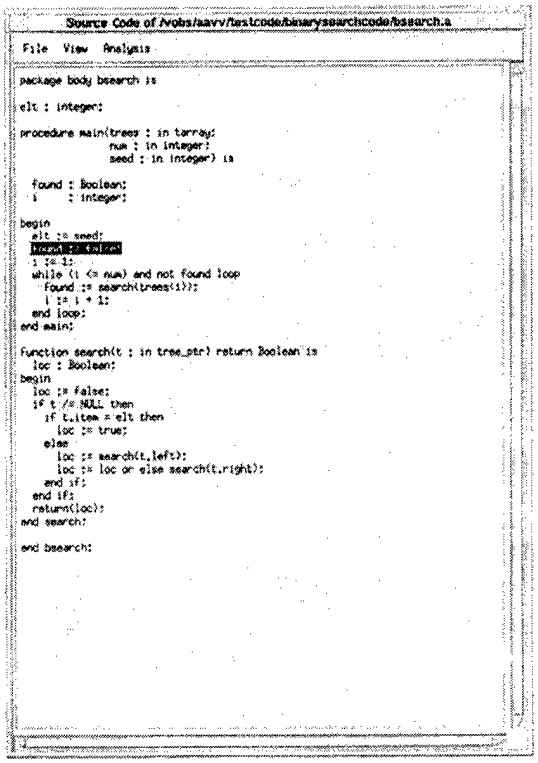

Figure 3-5 Source Code Listing View

presently open. Note, however, that the node for the selected statement must be displayed in a graphical view for the hyperlinking to that view to be activated.

This view is essential in determining the statements that would be effected by a change to the software. Assume that a certain line of code needs to be changed. Clicking on that line will cause the node for the line to be highlighted in the PDG. Clicking on that node will cause the nodes for all dependent statements to be highlighted. These dependent statements are the ones that may be affected by changing the line of code. This information helps the maintainer in determining which parts of the software may have changed functionality and, just as importantly, which parts of the software are unaffected. This will permit the maintainer to focus testing where the modification could have resulted in some undesired side-effect and to avoid testing portions of the software where no faults could have been introduced. This information on dependences could also be used to examine multiple alternatives to modifying the code, permitting the maintainer to choose the alternative that requires the least testing or, perhaps, results in the least potential impact on selected critical functionality of the software.

Declaration Diagram

Another view of the analyzed software that is provided by ImpACT is the Declaration Diagram view. This view may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. The Declaration Diagram permits the user to explore the declarations contained within a specific file.

Initially, the window for this view shows only the top-level declarations contained in the file (such as packages and subprograms not contained within packages). To obtain more detailed information associated with a certain declaration,

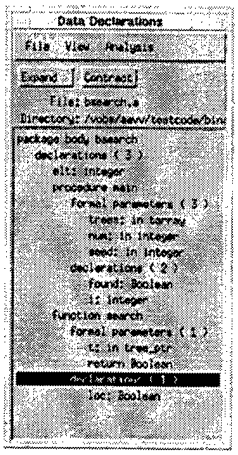

Figure 3-6 Fully Expanded Declaration Diagram for a Package Body

the maintainer clicks on that declaration and then clicks on the `Expand` button. For example, if a package is expanded, the number of declarations will be displayed and, if that number is expanded, the actual declarations contained within the package (such as subprograms and variables) will be presented. An example of a fully expanded Declaration Diagram for a package body is presented in Figure 3-6. Detail for a declaration can be removed from the window by clicking on the declaration for which detail is no longer desired and then clicking on the `Contract` button.

CFG

The CFG view is one of the graphical views of the analyzed software that is presented by ImpACT. This view may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. An example of a CFG is shown in Figure 3-7.

The CFG, as well as other graphs displayed by ImpACT, consist of nodes and edges. For these graphs, the nodes represent statements in the analyzed program, and the edges represent some relation between the nodes. For CFGs, these relations represent potential transfer of control between the two statements. In the dependence graphs, the edges represent a dependence between the two statements. Therefore, the CFG is a graphical representation of the statement control structure of a subprogram.

The CFG is hyperlinked to the Source Code Listing, Dataflow Information, and the other graphical views. Note that hyperlinking from any graphical view to the Source Code Listing is only activated if the Source Code Listing view is displaying the code associated with the selected graph node. Similarly, hyperlinking from any graphical view to either the CFG view or the Statement Dependence Graph view is activated only if the selected graph node is displayed in the corresponding graphical

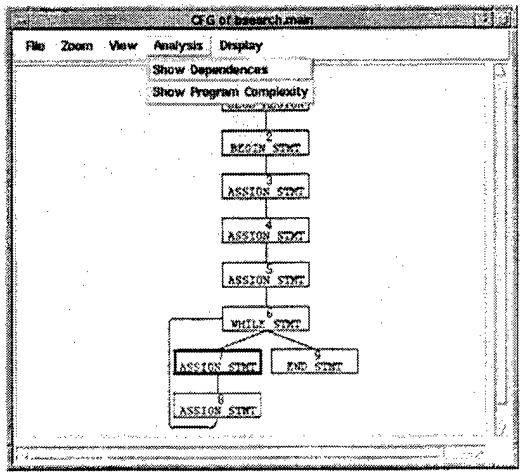

Figure 3-7 Control Flow Graph View

view.

The CFG view has many additional capabilities that are shared with the other graphical views. Graph nodes may be moved by dragging them with the mouse, permitting the maintainer to alter the appearance of the graph in such a way that he or she may interpret it more easily. Graphs may be either saved as postscript to a file or printed to a postscript printer. Saving the graph as a postscript file permits its inclusion in documents. Because some of the graphs can be quite large and therefore difficult to read when the entire graph is depicted in the window, ImpACT graphical windows provide functionality to zoom in, zoom out, and zoom by a user-specified percentage. Each graphical view, with the exception of the Statement Dependence Graph view, has an `Analysis` menu that can be used to open the Statement Dependence Graph and the Dependence Metrics window. The Statement Dependence Graph requires that a node in the original graph (such as the CFG) be selected - it is the dependences of this selected node that are displayed in the Statement Dependence Graph.

Data Flow Information

The Data Flow Information view is closely associated with the CFG view. This window may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. It displays the defined and used variables for a statement selected in the CFG. It also indicates whether these variables are parameters, local variables, or global variables. An example of the Data Flow Information window is shown in Figure 3-8.

Variables that are defined by a statement are those whose values may be changed by execution of the statement. Similarly, variables that are used by a statement are those whose values may be used in the statement. Clearly, a variable can be both used and defined in the same statement (e.g., `x := x + 1;`). To address arrays and records, it is possible to have variables that are partially defined or partially used. This happens when some particular element of an array or some particular field of a record may have its value modified or used in the statement, respectively. An example, assuming that `a` is an array of integers, would be `a(i) := a(i) + 1;`. In this

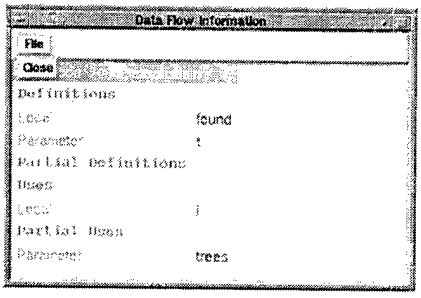

Figure 3-8 Data Flow Information Window

example, `a` is both partially used (the `a(i)` on the right hand side of the assignment operator) and partially defined (the `a(i)` on the left hand side of the assignment operator).

Special considerations needed to be given when defining the Data Flow Information for statements that call subprograms and for the subprograms themselves. All actual parameters are considered used (or partially used) in the statement that calls a subprogram. If an actual parameter is passed by reference, then the actual parameter is also considered as defined (or partially defined) by the call statement. All formal parameters of a subprogram are considered defined in the declaration region of the subprogram as well as at each call of the subprogram. If the a formal parameter is a reference parameter, then the formal parameter is also considered as used at each call of the subprogram.

ICFG

The ICFG view is another of the graphical views of the analyzed software that is presented by ImpACT. This view may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. The information presented in this graph is created using the CFGs for the entire program with edges added between the nodes containing the subprogram calls and the CFGs of the subprograms being called. This graph shows the flow of control and program structure for the entire program.

The edges between the calling nodes and the CFGs being called can be depicted in one of two ways. By default, the graph is depicted as a set of named CFGs with special, oval nodes attached to statements that make subprogram calls. These special nodes contain the name of the CFG that is being called. An example of this depiction of an ICFG is shown in Figure 3-9. A second way of depicting an ICFG is by drawing the

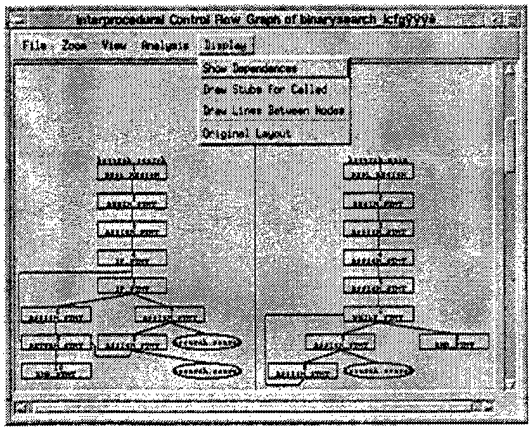

Figure 3-9 Interprocedural Control Flow Graph with Nodes Depicting Calls

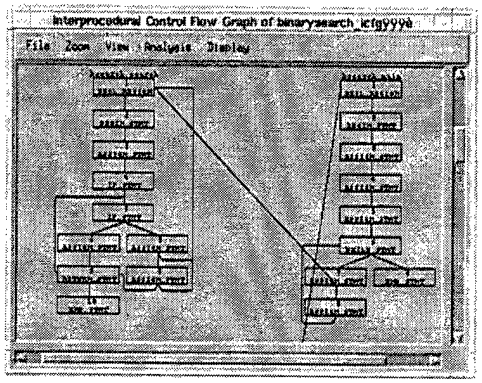

Figure 3-11 Interprocedural Control Flow Graph with Lines Depicting Calls

edges from the nodes that call subprograms to the first node (typically a declaration region node) of the CFGs being called. An example of this depiction of an ICFG is shown in Figure 3-11.

The ICFG can also be depicted as a call hierarchy chart. In this depiction, a node represents an entire subprogram, and an edge represents a call from one subprogram to another. This representation is useful for understanding the overall structure of the program without the low-level details. An example of an ICFG depicted in this manner is given in Figure 3-10.

Program Dependence Graph (PDG)

The PDG view is another of the graphical views of the analyzed software that is presented by ImpACT. This view may be displayed using the `View` pulldown menu option of either the main ImpACT window or of the window for some other view. The information presented by this view is generated by the algorithms described in Chapter 2.

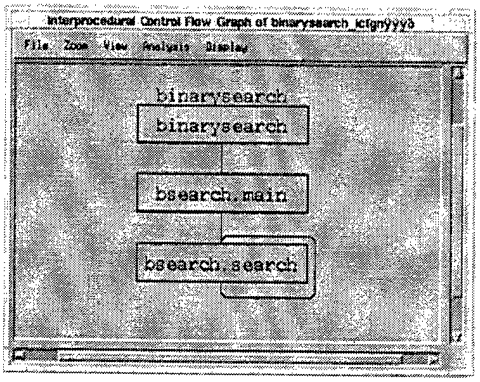

Figure 3-10 Interprocedural Control Flow Graph Depicted as a Call Hierarchy Chart

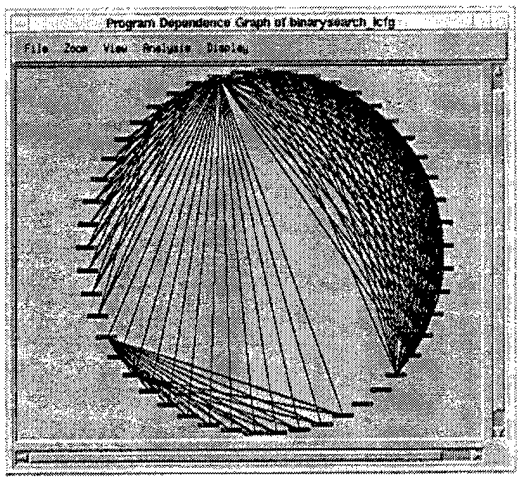

Figure 3-12 Program Dependence Graph View

An example of a PDG is shown in Figure 3-12, and a zoomed portion of that graph is shown in Figure 3-13. The nodes in the graph represent statements in the program, and the edges between nodes represent dependences between the statements represented by the nodes. The type of dependence that is displayed by the view can be set by the user by modifying the PROGRAM_DEPENDENCE_GRAPH_KIND environment variable, before invoking ImpACT, to indicate the type of dependence desired. By default, the type of dependence that is displayed is *weak interprocedural syntactic*

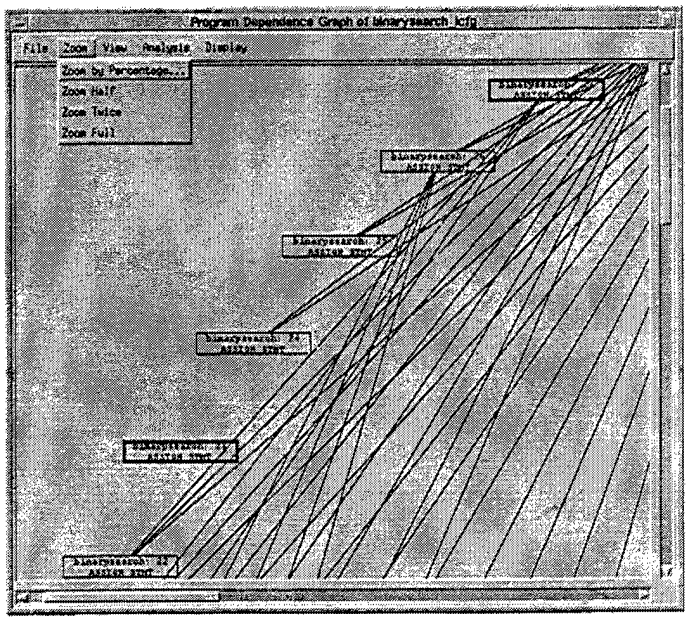

Figure 3-13 Zoomed Portion of Program Dependence Graph View, with Highlighting

dependence.

Clicking on a node in the graph will cause all nodes with the type of dependence being displayed, with respect to the selected node, to be highlighted. The edges from the selected node to the dependent nodes will also be highlighted. The highlighted nodes correspond to the statements that may be impacted if the statement corresponding to the selected node is modified.

Statement Dependence Graph

The Statement Dependence Graph is the last of the graphical views of the software. This view may be displayed using the `Analysis` pulldown menu of the other graphical views. An example of a Statement Dependence Graph is shown in Figure 3-14.

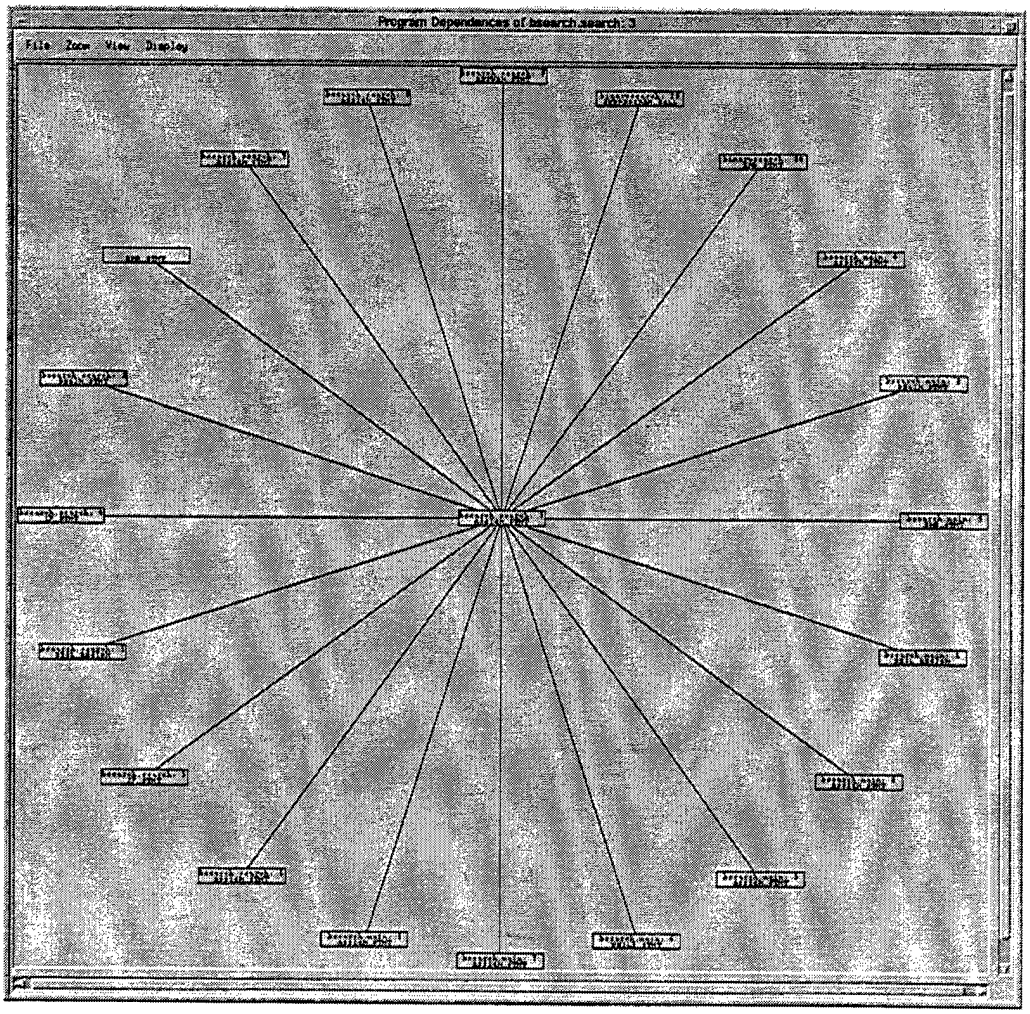

Figure 3-14 Statement Dependence Graph View

This view shows one node at the center of the graph, with nodes encircling it. The node in the center represents a statement, and the nodes along the circle represent statements that depend on the statement in the center (given the selected type of dependence). The lines from the center node to the nodes on the circle represent the dependences between the nodes. Only dependences of the node in the center are displayed.

One use of this window is to view the dependences and hyperlink to the actual impacted statements in the source code. This is done by having the source code file containing the impacted statement displayed in the Source Code Listing window, and then clicking on the dependent node in the Statement Dependence Graph.

Dependence Metrics

The Dependence Metrics view is displayed using any `Analysis` pulldown menu, and displays information on the number of dependences present in the analyzed software. These metrics are displayed for data dependences, control dependences, and all dependences (conceptually, a transitive closure on the dependencies). These metrics are displayed for the entire program as well as for a specific module of the program. Both the program and the module may be selected by the maintainer. An example of the Complexity Metrics view is shown in Figure 3-15.

The metrics information may be used to control the rate at which the program complexity grows. Typically, as the software is maintained, the software becomes more complex and, therefore, less maintainable. The result is that maintenance activities take longer to perform as time passes. The complexity information that is presented in the Complexity Metrics view can be used to evaluate the impact that maintenance alternatives have on code complexity, permitting the maintainer to implement the alternative that least increases complexity.

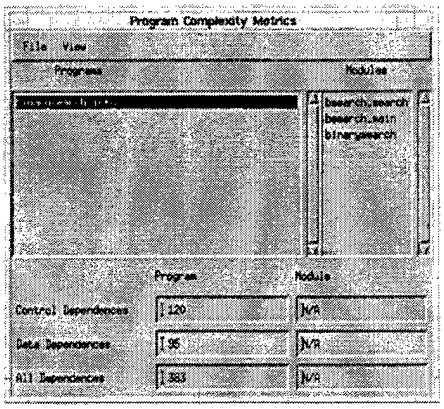

Figure 3-15 Complexity Metrics View

Recommendations for the User Interface

The ImpACT user interface is a production-quality system. There are some modifications that can be recommended, however, to increase the ease of use for the software maintainer. These recommendations center around enhancing support for the displaying of large graphs.

The size of a CFG is related to the size of a module, whereas the size of an ICFG is related to the size of the entire program. Module size is typically fairly small, usually dictated by good programming practices. Therefore, display of large CFGs is not a concern. The size of a program, however, is unbounded, is much larger, and may consist of numerous program modules. Therefore, these graphs may become very large, and presenting the information to the user can become difficult. Presently, ImpACT presents the entire ICFG, and the maintainer must zoom into a portion of interest and use the scrollbars to view the area of interest in the graph. In doing this, the maintainer no longer is presented the context of the area of interest within the program. TASC recommends detailed design and implementation of a reference window scheme to address the display of large ICFGs. In this design, one window would display the entire ICFG of the program, and the other window would present a detailed portion of the ICFG. The detailed portion would be enclosed in a box in the window containing the entire ICFG. The portion of the ICFG for which details are being displayed could be changed by either scrolling in the detail window or by dragging the box to the desired area of the ICFG in the window that displays the entire ICFG.

As with the ICFG, the size of the PDG is related to the size of the program. Unlike the ICFG, the nodes typically have a sizeable number of edges between them, and the nodes that are connected by the edges are often distant within the graph. Therefore, an approach other than the reference window approach will be necessary to improve the displaying of PDGs. TASC recommends an approach that partitions the graph into hierarchies of subgraphs, where each subgraph can be expanded or contracted. The subgraphs would be represented as a special node, and dependence edges would run between subgraphs that have dependencies between each other. Expanding on the subgraph node would cause detail of the subgraph to be displayed, with dependence edges leading to and from the detailed contents of the subgraph. Note that the contents of a subgraph can be a set of subgraphs, a set of nodes for program statements, or a combination of subgraph and statement nodes. This approach would allow the maintainer to closely examine the details that he or she is interested without being provided with extraneous detail in which he or she is not interested.

An additional recommendation is that views automatically load information that is necessary for hyperlinking when such information is not already loaded. For example, clicking on a graph node when the file containing the corresponding

statement is not presently displayed in the Source Code Listing window does not activate the hyperlinking to that window. A suggested enhancement would be to allow the maintainer to optionally decide, for the Source Code Listing and CFG, whether the necessary information should be loaded to permit the hyperlinking to be activated.

3.3 DEMONSTRATION OF THE ADVANCED PROTOTYPE FOR ADA

Two different pieces of software were used to demonstrate ImpACT support for sequential Ada programs. One piece of demonstration software was a relatively small binary search program, and the other piece of software was an implementation of the Navigation Support Component of the F-16 Fire Control Computer (FCC) Operational Flight Program (OFP). In addition, some small Ada programs were developed for testing different capabilities of ImpACT.

3.3.1 The Binary Search Demonstration

The binary search program consists of two implementation files and one specification file. One of the implementation files contains the main procedure of the program. This procedure creates a small binary search tree containing integers, and then calls the binary search procedure. The other implementation file contains a package consisting of the main binary search procedure and a recursive binary search function. The procedure calls the function multiple time to perform a set of searches. The specification file contains the specification of the package. The files contain a total of 67 lines of code, determined by counting lines containing semicolons (;).

This code was used to demonstrate and test the capabilities of ImpACT for Ada. The code was successfully processed and an analysis database was created. Using this generated analysis information, all of the capabilities of the views were exercised. In demonstrating ImpACT to potential users, TASC displayed the Source Code Listing, CFG, Data Flow Information, ICFG, and PDG views. With these views open, the hyperlinking capabilities were demonstrated by clicking on program statements and on graph nodes. Also shown was how the user can select a statement to be modified and then click on the highlighted node in the PDG to determine the nodes for statements that would be impacted by changing that statement. This would be followed by opening the Statement Dependence Graph view and showing that clicking the nodes for the statements in that view would hyperlink back to the impacted statements in the Source Code Listing window. We also demonstrated the Complexity Metrics view and explained how this view could be used to control the rate of growth of program complexity and the impact this would have on future maintenance activities.

3.3.2 The F-16 FCC Navigation Support Component Demonstration

The main purpose of this demonstration was to show that ImpACT is capable of handling avionics-based source code of a significant size. The source code was contained in sixteen source files, two of which were Ada package specification files. The demonstration code consisted of a total of 1074 lines of source code, determined by counting the number of lines containing a semicolon (;). This code was modified as described in Section 4.5.1 of [22]. These modifications were to make substitutions for RECORD and FLOAT types such that dependence analysis would not be affected and to remove the Ada tasking. Note that it appears that the code demonstrated did not have a modified main routine to compensate for the removal of the tasking.

In this demonstration, the code was processed and an analysis database was created. However, since the main routine only called the initialization routine (the functionality of the component was within the Ada task), the resulting ICFG and PDG were trivial. However, the interprocedural dependence analysis capability was also demonstrated using the JOVIAL prototype support for ImpACT on some substantial avionics software (the Front-End component is the only part of the system that is specific to the source code language). The F-16 demonstration showed that substantial quantities of Ada avionics code could be processed by the Front-End Component of ImpACT.

3.4 RECOMMENDATIONS FOR THE PROTOTYPE FOR ADA

At this time, the improvements to the Analysis component that are described in Chapter 2 have not been integrated into the released version of ImpACT for Ada. To integrate the improvements, portions of the Front-End component would need to be rebuilt with the file that describes the IRIS graphs that was used for the JOVIAL version of ImpACT. Using that file would work since it was created by adding IRIS graphs to the IRIS graphs that were already described for Ada.

Another area for improvement would be to provide support for the few Ada language features that are currently unsupported. Of particular interest would be adding support for Ada tasking. This would require a search for any published approaches to dependence analysis in the presence of tasking, development of the theory for such dependence analysis if it does not exist, and implementing support for such analysis in ImpACT.

Additional areas for improvement that are associated with ImpACT for Ada are included in the discussion of the Analysis component (see Chapter 2) and in the above discussion of the User Interface component. In particular, time and resource improvements are needed for the interprocedural dependence analysis algorithms, and

the ICFG and PDG graphical views need to be enhanced to provide better support for large programs.

4. THE IMPACT PROTOTYPE FOR JOVIAL

Task II of the AAV&V-II program involved the development of a prototype, based on the advanced prototype developed in Task I, for analysis of programs written in JOVIAL. Our development of this prototype pursued the following goals:

- *Address the needs of the target users* - Design the prototype, based on the prototype from Task I, to address the needs and concerns of avionics software testers and maintainers.
- *Provide the AAV&V system on a popular platform* - Develop the prototype on a commonly-used combination of hardware and operating system.
- *Enhance the basis for AAV&V* - Design the prototype with extension of the prototype in mind.
- *Extend the state-of-the-art* - Through the design and implementation of the prototype, advance the state of V&V techniques, practices, and concepts.

We used the following approach to design and implement the prototype for analyzing programs written in JOVIAL to achieve these goals:

- We based the prototype on the advanced prototype for Ada that was developed in Task I, which used a popular platform, was extensible, and leveraged existing software where possible.
- We designed and implemented the prototype by providing language processing capabilities for JOVIAL and reusing most of the prototype from Task I without modification.

The following sections describe the design, implementation, and enhancement of this prototype. First, we describe the design of the prototype, particularly the design of the language processing capability for JOVIAL. Second, we describe the implementation of the prototype. Third, we describe the software upon which the prototype was demonstrated and the demonstration itself. Finally, we make recommendations for further efforts with this prototype.

4.1 DESIGN OF THE PROTOTYPE FOR JOVIAL

The architecture and design for the ImpACT support of JOVIAL are identical to that for the support of Ada, discussed in Section 3.1. However, the Syntactic Analyzer and Semantic Analyzer for the front-end had to be designed by TASC, whereas they were leveraged from code from the Arcadia Consortium in the ImpACT support for Ada. Also, the CFG Generator, also leveraged from the Arcadia Consortium, needed to

be modified to support some language constructs that were present in JOVIAL but not in Ada.

The Syntactic Analyzer and Semantic Analyzer were designed and implemented as one program. In the design of this program, TASC developed an AYACC grammar that was based on JOVIAL as defined in MIL-STD-1589C. TASC also had to define what the IRIS graphs for JOVIAL would look like. This was done by first developing a correspondence between language features in Ada and JOVIAL. Next, for language features that were identical or nearly identical in both languages, the IRIS graph that represented the Ada construct was typically acceptable for JOVIAL, although some minor modifications were sometimes necessary. For language features present in JOVIAL for which there were no similar constructs in Ada, new IRIS graphs were designed. A draft of the IRIS graphs for JOVIAL, along with samples of code that showed a use of the construct, were provided to the researchers at The University of Massachusetts that developed IRIS and the IRIS representation of Ada. These researchers provided some valuable suggestions, many of which were incorporated into the design.

Because the prototype was to support only a subset of JOVIAL, the JOVIAL constructs were prioritized based on the perceived necessity in supporting demonstrations. High priority constructs were those necessary to provide a reasonable, non-avionics demonstration. The medium priority constructs were those expected to be used with some frequency in avionics software. The low priority constructs were those that TASC felt we could either work around or were not likely to be used in avionics software. These priorities were slightly modified after reviewing some JOVIAL avionics software provided by the B-1B avionics software maintainers. TASC designed and implemented all of the high priority constructs and a majority of the medium priority constructs.

4.2 IMPLEMENTATION OF THE PROTOTYPE FOR JOVIAL

In implementing the prototype for JOVIAL, the implementations of the Analysis component, the Persistent Database, and the User Interface were used without modification. However, a Syntactic and Semantic Analyzer had to be developed for JOVIAL and the CFG Generator required some minor modification. It should be noted that the Declaration Diagram has not yet been incorporated into the version of ImpACT supporting JOVIAL. Additionally, the generation of analysis information, which is perform via the ImpACT graphical user interface for Ada, has not yet been incorporated into the ImpACT graphical user interface for JOVIAL. However, the analysis information can be created using a series of commands issued at a unix prompt, as detailed in the ImpACT User's Manual [6].

4.2.1 The Syntactic and Semantic Analyzer for JOVIAL

The syntactic and semantic analyzer was designed and implemented to process JOVIAL source files complying with the MIL-STD-1589C definition of the JOVIAL programming language. This was done by developing a JOVIAL parser, and then adding actions to the parser to build the IRIS representation corresponding to the JOVIAL language constructs.

The parser was developed using the Arcadia tools ALEX and AYACC. ALEX is a lexical analyzer, reading input and returning tokens when called. AYACC is a parser generator, calling the lexical analyzer to obtain tokens and matching the tokens against a set of grammar rules. Once a grammar rule has been matched, the action defined for that rule is executed. Both the definition of the tokens and the set of grammar rules were derived from the description of JOVIAL contained in the LRM (MIL-STD-1589C). Once the parser was implemented, the grammar rules needed to be modified to remove parsing conflicts that existed. Once the parsing conflicts were minimized to a point where they did not impact the ability to parse JOVIAL code, the parser was tested against a set of small samples of code generated for the sections of the JOVIAL LRM. At conclusion of the implementation of the parser, a few JOVIAL constructs were not supported by the parser. These constructs are:

- Text manipulation directives (`!SKIP` and `!COPY` directives, and `DEFINE` calls)
- Reserved single-letter tokens used as variables

Before adding actions to the grammar, the IRIS graphs for JOVIAL were designed and reviewed as described in Section 4.1. The JOVIAL constructs were then prioritized for implementation to ensure that the subset of JOVIAL supported by ImpACT would be sufficient for providing substantial demonstrations. Actions were then added to the grammar, based on priority, by designing and implementing code that would create the IRIS representation of the corresponding JOVIAL language construct. The supported language features were incrementally tested, using small fragments of code, as their actions were implemented. When a significant number of language constructs were implemented, we also began testing against larger sections of code, entire modules, and small sample programs.

The resulting JOVIAL processing capability does have some limitations, although the existing capability was complete enough to give substantial demonstrations of ImpACT's capabilities on JOVIAL OFPs. One limitation is that all source code must be present in one file. This limitation allowed the details of resolving external references to be postponed, an area addressed by compiler and linker/loader theory, and language construct coverage to be concentrated upon. Although a majority of the language constructs were covered, there are some constructs that are not supported. These constructs either tend to occur rarely in OFPs or can be modified in

Table 4-1 JOVIAL Language Features Implemented and Not Implemented

Language Constructs	Implemented Elements	Not Yet Implemented Elements
Modules	Compool, Procedure, Main	
Data Type Descriptions	Signed/Unsigned Integers Floats and Fixed Characters Bit Literals and Booleans STATIC	Status Pointer Type matching and conversion LIKE-Option POS Attribute
Operators	Arithmetic (+, -, *, /, **, MOD) Logical (AND, OR, XOR, EQV, NOT) Relational (=, <>, <, >, <=, >=)	
Declarations	ITEM TABLE TYPE CONSTANT External(REF & DEF)	DEFINE BLOCK Statement Name OVERLAY Null
Statements	Assignment WHILE/FOR Loops IF/CASE Procedure/Function Calls GOTO RETURN EXIT	ABORT STOP
Procedures & Functions	Procedure Declaration & Definition Function Declaration & Definition	INLINE Procedures & Functions Machine-Specific Procedures & Functions
Data References	Simple Subscripted	Pointers
Built-in Functions	NENT ABS LBOUND and UBOUND	NEXT LOC BIT, BYTE, SHIFT SIGN, SIZE, NWDSEN Status Inverse
Directives	External Declarations (REF instead of compool directive) used to access variables and procedures declared in compool modules.	All Directives have not yet been implemented.

the source code being analyzed without impacting the quality of the resulting analysis. Table 4-1 details the JOVIAL language constructs that have and have not been implemented.

4.2.2 The Control Graph Generator for JOVIAL

The CFG Generator for JOVIAL programs was implemented by making some minor modifications to the CFG Generator for Ada programs. One minor modification

was necessary because the IRIS generated for JOVIAL programs had a different root node than the IRIS generated for Ada programs. A second minor modification was necessary because of the differences between function definitions in JOVIAL and function definitions in Ada. A third minor modification was necessary to support FOR loops in JOVIAL, which have more operands than FOR loops in Ada. The final minor modification that was made to the CFG generator was necessary to support CASE statements in JOVIAL, and to support the FALLTHROUGH capability of JOVIAL CASE statements in particular. Other than these modifications, which were less than one person-month of effort, no other changes were necessary to support the IRIS representation of JOVIAL, as presently implemented.

4.3 DEMONSTRATION OF THE PROTOTYPE FOR JOVIAL

The prototype ImpACT support for JOVIAL was demonstrated on a small piece of avionics-like code and on two subsystems of the B-1B CITS OFP. The software was demonstrated to avionics software maintainers for the B-1B and for the AC-130. The B-1B maintainers were provided with a training class on ImpACT, and the software was installed on one of their systems for their use and evaluation.

4.3.1 Demonstration on the Avionics-like Code

The small, avionics-like, JOVIAL source file that was used for demonstrations consisted of approximately 55 lines of code, determined by counting semicolons (;). Of those 55 lines, 17 involved data declarations such as those that would typically appear in a COMPOOL module. The sample program consisted on six JOVIAL procedures, all relatively small.

This code was used to demonstrate the capabilities and uses of ImpACT. All of the implemented views would be demonstrated, as was done in the demonstration of the ImpACT support for Ada. During the demonstration, we would also click on a line of code to see it highlighted in the PDG view, would click on the PDG node to see the nodes dependent on the statement, and would open the Statement Dependence Graph view to see only the nodes dependent on the selected line of code. We would then click on the nodes in the Statement Dependence Graph to hyperlink back to the Source Code Listing view and would show how these lines of code did depend on the line of code that was originally selected.

4.3.2 Demonstration of B-1B CITS subsystems

Management of the maintainers of the B-1B CITS OFP provided TASC with their source code to analyze using ImpACT. Two subsystems of CITS were analyzed - ICE and LDGR.

ICE is the de-icing subsystem for the B-1B. It consisted of approximately 600 lines of JOVIAL procedure source code. This source code also required the analysis of thousands of lines of declarations from COMPOOL modules. ICE was chosen because it

Table 4-2 CITS JOVIAL Nuances

Non-Parseble CITS Code	MIL-STD-1589C Rules
END;	No semicolons after END required.
MY.VARIABLE	No periods allowed in variable names.
FOR X (1 BY ...);	For loops syntax don't use parenthesis: Ex. FOR X: 1 BY...;
DEFINE BTOF (BTOFPARM) "…….";	A Define-Formal (the portion in parenthesis) must be only a single letter! Ex. DEFINE BTOF(Q) "…….";
CONSTANT X B16 = ...;	CONSTANT must always be followed by the keyword ITEM Ex. CONSTANT ITEM X B16 = ...;
ARRAY X(3) B 8;	No keyword for type ARRAY
DEF BLOCK ICE$CON; BEGIN X = 2.6;	Not allowed to do statements in block bodies.
COMPOOL (file-name);	This appears to be a COMPOOL directive, which should be preceded by a "!".
TABLE FOO ( ) ...	Table dimension must be specified. Cannot be empty. Ex. TABLE FOO (*) ..
TABLE FOO ( ) 1;	Use of a single digit for table-description is not legal.
OVERLAY FOO = FOO2, FOO3;	The equals sign should be a colon. Ex. OVERLAY FOO : FOO2, FOO3;
TABLE FOO2 (1:3) LIKE FOO;	The like-option can only be used in a type-declaration. Ex. TYPE TABLE FOO2 LIKE FOO;
COPY FOO;	Compiler directives need a "!" in front of them. Ex. !COPY FOO;
ITEM FOO (00, 00) B 1;	Need to use POS, and shift order. Ex. ITEM FOO B 1 POS (00,00);
SWITCH FOO = (CASE1,CASE2);	No SWITCH keyword.
GOTO FOO(<index>);	GOTO's are followed only by a label name
X' for hex numbers	Not defined. Need to use a 4B' instead.

was relatively small and did not involve the use of any assembly code. The files for ICE had to be combined into one source file since the ImpACT support for JOVIAL does not presently support multiple source files. Also, the source code had to be modified to conform with the MIL-STD-1589C definition of JOVIAL - it was later discovered that the dialect of JOVIAL that CITS is written in is J3B2. Table 4-2 provides details on the features of J3B2 that were used in CITS but that are not compliant with MIL-STD-1589C. These features were converted to their closest MIL-STD-1589C equivalent. After conversion, the ICE subsystem was completely analyzed and all of the views of the analyzed software have been demonstrated.

LDGR is the landing gear subsystem for the B-1B. It consisted of approximately 1750 lines of JOVIAL procedure source code. This source code also required the analysis of thousands of lines of declarations from COMPOOL modules. LDGR was chosen because it was slighly larger than the typical CITS subsystem and because it did not involve the use of any assembly code. As with ICE, the files had to be combined into one source file and syntax specific to J3B2 had to be converted into the closest MIL-STD-1589C equivalent. After conversion, the LDGR subsystem had its IRIS representation generated, its CFGs and ICFG generated, and had some of the dependence analysis performed on it. The dependence analysis that was performed consisted of immediate forward dominators, forward dominators, and reachability analysis. Because of memory leaks in the leveraged ProDAG software, it was not possible to complete either data or control dependence analysis before running out of memory. All analysis information that was able to be generated was demonstrated within the ImpACT user interface.

4.4 RECOMMENDATIONS FOR THE PROTOTYPE FOR JOVIAL

As mentioned previously, the ImpACT support for JOVIAL does not support software in multiple source files. Therefore, TASC recommends examining how multiple source files are supported for Ada and implementing a similar approach for JOVIAL.

Besides handling multiple files, the B-1B CITS maintainers pointed to JOVIAL language coverage as an issue in adopting ImpACT. Therefore, the entire MIL-STD-1589C language definition should be supported. Because JOVIAL dialects are a significant issue, they may also need to be addressed. Since it can be assumed that the source code being analyzed is syntactically correct, this can be accomplished by relaxing the existing grammar to also recognize syntax consistent with other specific dialects of JOVIAL.

Presently, the analysis information is created by entering commands at the Unix prompt. TASC recommends a small effort to integrate the creation of the analysis

information into the ImpACT graphical user interface. This can be implemented in a way similar to the implementation for Ada. Similarly, the Declaration Information view should be supported in the version of ImpACT for JOVIAL.

Finally, ImpACT support for JOVIAL has the same issues concerning program size as the support for Ada. Improvements are needed in both the time and resources required for generating the interprocedural dependence information. Also, the ICFG and PDG views should be enhanced to provide improved support for analyzed large programs.

5. AUTOMATED TESTING SURVEY AND EVALUATION

Task III involved surveying and evaluating a number of different methods and systems in the areas of automated support of software testing and automated formal verification. The associated literature search was conducted using a number of different resources. This chapter first discusses the research resources utilized, and then presents a detailed summary of the methods and systems located during the literature search. After the summary of the methods, the criteria used to evaluate each of the methods and systems are discussed. The chapter concludes with an overview of the recommended approach to the verification and validation of implemented software changes.

5.1 RESEARCH RESOURCES UTILIZED

In conducting the literature search, a number of different resources were used. These resources included the World Wide Web, journals and proceedings, computer science technical reports databases, and references from located papers. In this section, we describe how each of these resources contributed to provide a comprehensive literature search.

5.1.1 The World Wide Web

The World Wide Web was used in several capacities in conducting the literature search. One example is keyword searches performed using AltaVista (located at http://www.altavista.com). Examples of keywords used in the searches are "formal verification", "automated test generation", and "test coverage". This method was effective because it is now common for Computer Science departments at universities to place copies of technical reports and expanded versions of journal papers on their web sites. This gives access to the most recent activity in the areas being researched, and often presents more detailed information than is typically available in journals and proceedings. The searches, however, also returned information that was not related to the survey, so only literature that appeared to be potentially relevant was downloaded for further examination.

In addition to web searches, the World Wide Web was used to go directly to the homepages of universities, or individual researchers, that appeared to be performing a significant amount of research in the areas of interest. One such example is Clemson, where several researchers have written papers about automated test generation and automated selection of regression tests. Once at the site, we examined the titles of

papers (as well as abstracts, when available) and downloaded the literature that appeared to be potentially related to the areas involved in the survey.

5.1.2 Journals and Proceedings

We also performed a literature search in a more conventional manner by searching publications that might contain information in the areas of interest. Representative journals are *IEEE Transactions on Software Engineering* and *ACM Transactions on Software Engineering and Methodology*. Proceedings of interest included the series from *International Symposium on Software Testing and Analysis* and the series from *Proceedings of the International Conference on Software Engineering*. We examined the tables of contents of the publications, and proceeded to the abstracts and papers of articles that appeared to be related to the areas of interest in the survey.

5.1.3 Technical Report Databases

Our literature search also used Computer Science technical report databases, which are also available via the World Wide Web. Two of the databases that were used were NCSTRL (Networked Computer Science Technical Reports Library, pronounced "ancestral") and United Computer Science Technical Reports Index (UCSTRI). Both of these databases provide search engines, which may be provided with desired keywords such as those that were used for the AltaVista searches. The titles were then scanned to see if the technical report appeared to potentially contain information related to our survey, and reports appearing to be related to the survey were downloaded for further examination.

5.1.4 Following References

One additional method used to locate literature was to use the references in papers that we had located using the previously discussed means. We followed a similar approach to the other search methods, examining the title and pursuing further the literature that appeared to be related to the areas of interest in our survey. The key to this approach was to locate recent literature so as to obtain references that contained other recent literature. In addition to using references to located papers, we also reviewed the references cited in the final report for the original AAV&V program (Ref. 22) for literature of interest to this literature search.

5.2 SUMMARY OF METHODS

In this section, we detail the methods and systems discovered in the literature search. This literature falls into five categories:

- Test coverage criteria
- Code annotations and assertion checkers
- Automated test generation
- Automated selection of regression tests
- Formal verification

The remainder of this section gives a summary, in some detail, of the methods and systems discovered in the literature search in each of these five areas.

5.2.1 Test Coverage Criteria

Test coverage criteria can be used to assist in determining whether software has been sufficiently tested by a set of test data. Typically, the program is recompiled in such a way that the execution of the program is tracked for later analysis. Some compilers, such as the C compiler supplied on Sun workstations, have a compiler option that instruments the executable program so that the execution is tracked. Utilities also exist that will instrument a program independent of the compiler. One clear disadvantage of instrumenting programs to track the execution is that this instrumentation interferes with the execution profile of the program being tested. This is because the program executes additional instructions to track its execution, which will have a negative impact on the real-time performance of the program. This can be overcome by removing the instrumentation once the desired test coverage has been reached, and then rerunning the test suite, although at a cost of having to run each test in the test suite twice. One additional problem is that, except for the most trivial of coverage measures, it may be very costly, or even impossible, to achieve full coverage of the program. A number of coverage measures are discussed in *Software Test Coverage Analysis* (Ref. 7), and are summarized in the remainder of this subsection.

Statement Coverage

The statement coverage measure determines whether each executable statement was executed by some member of the test suite. This coverage is easy to measure, but has several faults. One fault is that an "if" statement without an "else" part may only be executed with the condition of the statement true, but the statement coverage criteria can be met. Figure 5-1 (Ref. 7) shows an example in which statement coverage can be achieved without considering an execution path that results in a fault. Another

```
int *p = NULL;
if (condition)
    p = &variable;
*p = 123;
```

Figure 5-1 Example Where Statement Coverage May Not Expose Fault

problem is that a "do-while" loop is treated the same as a linear sequence of code without a control structure. Similarly, "for" and "while" loops result in statement coverage even without considering the case in which the loop is never entered.

Branch Coverage

The branch coverage measure determines whether each boolean expression appearing in control structures has been evaluated as both true and false. For languages that use short-circuit evaluation of boolean expressions, this coverage measure is not sufficient if subexpressions of a boolean condition have side-effects. Short-circuit evaluation is when only enough of the expression is evaluated to determine its truth value. For example, given "A and B" and the fact that "A" is false, it can be determined that "A and B" is false without needing to know the truth value of "B". Similarly, given "A or B" and the fact that "A" is true, it can be determined that "A or B" is true without needing to know the truth value of "B".

Condition Coverage

The condition coverage method was developed to overcome the limitations of the branch coverage method. In condition coverage, each of the boolean subexpressions of a boolean condition must be evaluated to both true and false. Note, however, that this coverage does not necessarily give branch coverage. For example, if a condition is "A and B", condition coverage can be met with two tests - one in which "A" is true and "B" is false, and one in which "A" is false and "B" is true. In this example, the boolean condition is never evaluated to be true, and the body of the control structure for which it is a condition is never executed.

Multiple Condition Coverage

The multiple condition coverage method was developed to overcome the limitations of the condition coverage method. In multiple condition coverage, the boolean subexpressions of a boolean condition must be evaluated with every possible combination of truth values. This can result in a large number of test cases if complicated boolean expressions are used. It also is inefficient for languages that do not have short-circuit evaluation of boolean expressions.

Condition/Decision Coverage

Condition/decision coverage is when both the condition coverage measurement and the branch coverage measurement must indicate that the test suite has resulted in an acceptable level of testing. This coverage basically is a strengthening of condition coverage by requiring that branch coverage also be attained (recall that the flaw in condition coverage was that it did not guarantee branch coverage).

Modified Condition/Decision Coverage

Modified condition/decision coverage is used to determine that each boolean subexpression in a boolean condition can affect the outcome of the boolean condition. This measure is required for aviation software by RCTA/DO-178B.

Path Coverage

Path coverage is used to determine how many of the possible paths through each function have been executed by the test suite. A path is defined as the sequence of branches from the entry point of the function to its exit point. Since loops introduce an unlimited number of paths, a strategy is typically developed that requires only a limited number of iterations of the loop. This method has the same problem with short-circuit evaluation as branch coverage had. Another disadvantage of this method is that growth in the minimum number of test cases is exponential in the number of branching statements. Finally, not all paths through a function are feasible - there may be numerous paths for which it is impossible to develop test data. This is one coverage measurement for which it may be impossible to reach full coverage, because of the infeasible paths.

Function Coverage

In function coverage, the goal is to require that each function or procedure is called. This assures that initial attempts at testing provide some coverage of the entire program.

Call Coverage

In call coverage, the goal is to execute every function and procedure call. This measure is useful to detect errors in interfaces between functions and procedures.

Linear Code Sequence and Jump (LCSAJ) Coverage

In LCSAJ coverage, the goal is to execute each subpath that consists of a linear sequence of statements. This subpath can contain statements that determine control flow as long as the statement on the line following the boolean condition is the next statement executed. A major difficulty with this method is the presence of infeasible paths.

Data Flow Coverage

In data flow coverage, only the sub-paths from definitions to uses of variables are considered. This method, however, does not require that branch coverage be attained. The computational complexity of the method is also a disadvantage. Finally, the use of pointers also presents problems for this method.

Object Code Branch Coverage

In object code branch coverage, the goal is for each condition branch to be tested with cases in which the branch is taken and in which the branch is not taken. This method is more related to the implementation of the compiler than the implementation in the source code, making it difficult to generate a suitable set of test cases from the source code. Problems also result from various compiler optimizations.

Loop Coverage

In loop coverage, the goal is to avoid executing the body of the loop (except for "do-while" loops), to execute the body of the loop exactly once, and to execute the body of the loop more than once. The advantage of this measure is that it is the only one that requires more than one iteration of the loop.

Race Coverage

In race coverage, the goal is for multiple threads to execute the same code at the same time. This measure is useful in detecting the failure to properly synchronize access to shared resources.

Relational Operator Coverage

In relational operator coverage, the goal is to test subexpressions involving relational operators with the boundary cases. The purpose is to test for off-by-one errors and incorrect inclusion or exclusion (the difference between using, for example, "<" and "<=").

Weak Mutation Coverage

In weak mutation coverage, programs that are identical to the program being tested, with one minor change, are generated. The test suite is then run against the "mutant" programs and the program to be tested. The goal is to have each mutant program distinguished from the program to be tested by at least one test. When a mutant has been distinguished from the program to be tested, the mutant is referred to as *killed*; all mutants that have not been killed are referred to as *live*. The proposition is that most incorrect programs are almost correct, with an occasional small mistake in the code. The problem with this method is that a large number of mutants can be

generated, and each test must be run not only against the program being tested but also against each live mutant program.

Table Coverage

In table coverage, the goal is to reference each element of each array. This method is particularly useful for programs that are controlled by finite state machines, where the finite state machines are represented using arrays.

5.2.2 Code Annotations and Assertion Checkers

One way of checking that a program performs as desired is to embed assertions into the code. These assertions can either be written in a construct that is part of the programming language, directly in the program using conditional statements, or by using formal comments that specify the assertions. Most programming languages, however, either do not provide an assertion mechanism in the programming language, or the information that is produced when an assertion fails in not adequate. The problem with using conditional statements to implement the assertions is that the assertions tend to blend in with the code and do not stand out. Assertions represented with formal comments require that there be a preprocessor that translates the assertions into conditional statements, which can then be compiled. The advantage is that the assertions do stand out in the code, and the assertions can be stated in a more formal notation.

Anna

Anna (Ref. 24) is an assertion-based specification language for Ada. Assertions are added to Ada as formal comments, either as virtual Ada text by preceding them with the symbols --: or as annotations by preceding them with the symbols --|. Virtual Ada text is written using Ada syntax, and is used to define programming concepts or to compute values that are helpful in explaining what the actual Ada program does. Annotations, on the other hand, do not have to use Ada syntax, but do have their own formal syntax. There are many kinds of annotations - object annotations, type and subtype annotations, statement annotations, subprogram annotations, exception propagation annotations, and context annotations. Object annotations are used to constrain the values that may be associated with a program variable. Similarly, type and subtype annotations are used to constrain the values that may be associated with a type or subtype. Statement annotations are used to state the intended effects of a statement, and are often used as assertions and loop invariants. Subprogram annotations are used to state preconditions and postconditions about a subprogram, as well as the return value of functions. Exception propagation annotations may be used to state in what conditions a call of a subprogram must cause the associated exception to be raised, or to state predicates about the execution state that will be true when the

```
procedure BINARY_SEARCH(A        : in ARRAY_OF_INTEGER;
                        KEY      : in INTEGER;
                        POSITION : out INTEGER );
--| where ORDERED (A),
--|       out (A(POSITION) = KEY),
--|       raise NOT_FOUND =>
--|             for all I in A'RANGE => KEY ≠ A(I);
```

Figure 5-2 Specification of Binary Search in Anna

associated exception occurs. The first of these annotations is referred to as a strong propagation annotation, and the second is referred to as a weak propagation annotation. Context annotations are used to state what external variables can and cannot be used within a program unit. An example of using annotations for a subprogram specification, taken from an overview of Anna (Ref. 24), is shown in Figure 5-2. Whenever any assertion in a formal comment is violated, the exception ANNA_ERROR is raised.

The Annalyzer (Ref. 23) is a tool with a graphical user interface that works with Anna specifications. The Annalyzer allows a user to suppress selected annotations. The user then runs the program under control of the Annalyzer, and is notified if any assertions fail and, if one does fail, is provided with the assertion that failed as well as the region of code in which the failure occurred.

The disadvantage of Anna is that the user is still left with the task of developing a set of test cases that will provide an increased sense of reliability. Basically, using assertions, such as those provided in the formal comments in Anna, requires that the user develop formal specifications for the program, but does not give the strong results that formal verification does. One other factor to consider about Anna, and other methods that use annotations and assertions, is that the checking of the assertions requires processing which may interfere with the timing involved in real-time systems.

APP

APP (Ref. 37) is an assertion processing system for programs written in the C programming language. Assertions are stated using formal comments that begin with /*@ and end with @*/. APP is a preprocessor that can be substituted in place of the C preprocessor, resulting in no change to the way that programs are compiled. There are four kinds of assertions provided - assume, promise, return, and assert. Preconditions of functions are stated using assume assertions, and postconditions of functions are

```
int* sort(int* x, int size)
/*@
    assume x && size > 0;  // x is non-null and size positive
    return S where S       // S is non-null
                && all(int i=0; i < in size-1; i=i+1)
                     S[i] <= S[i+1]  // S nondecending order
                && all(int i=0; i < in size; i=i+1)
                     some(int j=0; j < in size; j=j+1)
                       x[i] == S[j];  // S permutation of x
@*/
{
    ...
}
```

Figure 5-3 Example of Annotated Sort Function

stated using promise assertions.[1] The return assertion is used to constrain the value returned by a function. The assert assertion is used to state a requirement about the program state at some location in the body of the function. The assertions can be individually assigned a severity level, and the user can specify, using an environment variable, the minimum severity level of the assertions that are to be checked during execution.

Besides stating assertions that are being made, APP provides the ability to state what action should be taken when a particular assertion fails. This action is written as C code, and can be used to provide detailed information that may help in pinpointing the location of the fault. If an action is not specified for an assertion, a default action is used that indicates the kind of assertion that failed, the file it was in, the line the assertion was on, and the function that was being executed.

One goal is to develop assertions that describe in a somewhat formal notation what the result of pieces of code should be. APP is designed such that specifications are not based on algorithms, but are based on first-order logic. Unfortunately, this can limit the expressive power of the assertions. An example, shown in Figure 5-3[2], was provided by Rosenblum (Ref. 37) in which this lack of expressiveness is evident. This example specifies a precondition of a sorting routine that takes an array of integers and a size, and returns a pointer to the sorted contents of the array. There are two problems with the example. One problem is that the contents of the input array could be changed

[1] A precondition is a statement that must be true in order for it to be valid to call a function or procedure. A postcondition is a statement of what is true when execution of a function or procedure terminates, assuming that the precondition was satisfied.

[2] The syntax of the example was modified from that used by Rosenblum to comply with ANSI C conventions.

between entering the function and checking the return assertion (in fact, the pointer may point to another memory location!). A copy of the array needs to be made on entrance to the function, but there is no C library function that does this. The other problem is that the check that the returned array is a permutation of the input array is not correct if either array contains duplicate elements. All the check does is make sure that the numbers in the input array also appear in the returned array, but not that the same number of occurrences appear in both arrays. The reason that permutation is not properly checked is that this is an operation that can only be done with an algorithm - the number of occurrences must be counted, or another array must be kept to indicate which return array elements have already been matched to an input array element.

5.2.3 Automated Test Generation

Automated test generation is the automated creation of test data, typically by analyzing either formalized requirements, a formal model of the system, or the source code that is to be tested. Testing against the requirements alone is not enough, especially for safety-critical systems such as avionics software. Often, in the process of designing software, the software engineer must address scenarios that were not considered in developing the requirements. The requirements usually address overall capabilities of the system, without addressing hardware limitations. One example of a hardware limitation is that memory is finite, and the resulting design decision that must be made is what to do if memory is required but none is available. Using a formal model of the system to derive test data assumes that a suitable model exists. Since the existence of such a model is unlikely and constructing such a model would require substantial effort, this subsection mainly considers approaches that derive test data from the source code. It is important to note that all of these methods essentially generate random data that meets some criteria, but that criteria is not always one that directly correlates to the probability of discovering a software fault.

The Genetic Algorithms Approach

One approach to automatically generate test data used an approach from the research area of genetic algorithms (Refs. 27, 35). In this approach, an initial population of test data sets is created, typically randomly. Individual data sets are selected, the instrumented program being tested is executed using the data, the program coverage of the data is recorded, and the data is placed in a temporary population. The selection and execution of data sets is repeated until either the desired coverage has been achieved, in which case the temporary population is the resulting test data suite and the program terminates, or all of the data sets have been executed. Data sets from the temporary population are then selected to proceed to the next generation. Several methods of selecting the individuals are briefly described in (Ref. 35), such as selecting the test data sets that had above average coverage results from their execution. Finally, before repeating the process with this new generation, the data is subject to crossover

and mutation. Both crossover and mutation treat a test data set as a single string of bits, and both events occur with a separate probability that can be changed to tune the algorithm. In crossover, a location is chosen in two different bit strings, the bit strings are split into two parts at that location, and the beginning portions of the bit string are reconnected to the ending portion to which they were not originally connected. In mutation, bits in the data are changed randomly.

Although theoretically interesting, this method is not practical for several reasons. One reason is that the data that is eventually evolved does not necessarily have any relationship to the data that might typically be involved in the use of the system. Another reason is that a population may not evolve to give the desired coverage. This is addressed by setting an initial number of generations, and then restarting the entire algorithm with a new initial population if a solution is not achieved in that number of populations. Finally, the instrumented program must be executed on each test data set in each generation until a population with the desired coverage is achieved, and the algorithm may need to be started over numerous times with a new initial population before a satisfactory population is obtained.

The Path Prefix Strategy

The path prefix strategy (Ref. 34) is to run the program using test data, and then repeatedly find the shortest executed path that leads to an untaken branch, derive data to follow the path and then take the untaken branch, and then execute that data. Each time a test is executed, the changes in program state along the path of execution are noted.

The fact that a test already exists that executes up to the untaken branch is utilized in determining the new test data. A variable in the condition at the untaken branch is modified so that the branch will be taken. This new data at the branch is then used, along with the state information for the execution of the path up to the branch point and the program logic, to derive the new test data for the program.

The authors describing this method did not describe the problems associated with this method. One problem is that it is easy to modify the data at the branch point to take the untaken branch, but there is no guarantee that doing so will result in data that also follows the same path as was previously taken to the untaken branch. This means that perhaps several changes will need to be made to variables in the conditions and then propagated to the beginning of the program to determine the test input data. It may be possible that no data will execute the path up to the untaken branch and then take that branch, meaning that the subpath that is trying to be covered is infeasible. Finally, if the expressions at the branches are complicated, the result is a large number of variables that could be chosen to have their values changed. Additionally, it might be necessary to change more than one variable to affect the truth value of the branch,

resulting in numerous changes to multiple combinations of variables that may need to be attempted before a successful data set is found (if one exists).

ADTEST

ADTEST (Ref. 13) is a system that uses instrumentation to force a desired execution path. This path is grown one control statement at a time, again using the same idea as the path prefix strategy did - that since the program executed to this point in the path, the data should already be close to that needed to take the desired branch. Besides being instrumented to take a desired execution path, the program is also instrumented to determine how close each control statement is to taking the desired branch in the path. These numbers are then used as the exponents in a penalty function, and the values of the penalty functions are summed. The goal is to minimize the sum of the penalty functions, which is done using an algorithm that is not detailed.

Like the path prefix strategy, this method depends on it being fairly easy to determine input data that will cause the desired branch to be taken given that the input data causes the program to execute up to the associated control statement. Also, the method may get stuck in a local minima and declare the path infeasible when there is a feasible minimum. Finally, we are unable to evaluate the minimization technique since no details about the technique were provided.

Domain Reduction

Domain reduction, a form of constraint-based test generation, is discussed by Offut (Ref. 31) and is also summarized by DeMillo and Offutt (Refs. 9, 10). The goal is to generate test data that will cause a certain branch in the code to be taken. A boolean expression, called a constraint system, is derived that must be true if the desired branch is to be taken. Therefore, the goal reduces to determining a set of values that will satisfy the constraint system.

One problem with the method, even before considering the main algorithm, is that the variables in the constraint system are not necessarily the input variables. Many of them may in fact be internal (or local) variables. Offutt points out that symbolic evaluation usually can be used to express the variables in the constraint system in terms of the input variables. Also, redundancy in the constraint system can be eliminated, and contradictory constraints[3] in the system can be eliminated from the constraint system. However, this is one additional step that must be taken, and it could result in complicated systems of constraints. Note that this symbolic evaluation is typically how formal verification is performed - all that is missing is a precondition (which is often straightforward) and a postcondition.

[3] By a contradictory constraint, Offutt means a conditional constraint such as X=a if $a > b \wedge a+1 < b$.

Once the constraint system is in terms of input variables, each input variable is assigned an initial domain. Next, constraints involving constant values are used to reduce the domain of the variable within that constraint. First, any known values for variables are substituted for the variables throughout the constraint system. Next, each constraint that compares a variable to a constant is used to reduce the domain of the variable. For example, if the constraint is $x < 0$, then the domain of x can be reduced to consist of only negative values. If the domain for a variable is reduced to one value, then that is noted and the value can be substituted into the constraint system in place of the variable on the next iteration of reducing constraints involving constants. If a domain for a variable is reduced to contain no values, then the domain is infeasible, and the constraint system satisfaction procedure will have to be run again with different choices made. The process of substituting known values into the constraints and reducing constraints involving constants is repeated until there are no variables for which a value has been determined in that iteration.

Next, each constraint involving two variables is reduced. Unfortunately, this is not described in any detail in any of the papers. The algorithm for the domain reduction procedure (Ref. 31) appears to be incorrect in this section of the algorithm, since it appears possible that different values could be chosen for the same variable in reducing different constraints containing that variable. The goal of this part of the procedure, however, appears to be to try to reduce domains to single values.

If the attempt to reduce constraints with two variables does not result in at least one variable with a known value, then a variable which has not been given a known value is chosen, and is set to be an arbitrary value from its domain. This choice is noted, and the algorithm returns to the beginning, substituting that value into the constraint system and trying to reduce constraints that contain a variable and a constant.

Problems with this algorithm are numerous. First, it may be difficult to satisfy complicated constraint systems. Second, loops and arrays present problems when symbolic evaluation is used to relate internal variables to input variables. Third, the algorithm may need to be attempted many times before the constraint system can finally be satisfied, although it is stated by Offutt that the run time of the algorithm is not significant compared to the time involved in running the tests.

Dynamic Domain Reduction

Dynamic domain reduction (Ref. 30) is an improvement on the domain reduction method, and is intended for generation of test data at the software unit level. This method requires the domains of the input variables, the control flow graph of the program, the start node of the graph, and the goal node of the graph. An untried path from the start node to the goal node, incorporating at most one loop structure, is

chosen. If no untried paths exist, then the path is either infeasible or the test data for that path is too difficult to generate. The goal is to traverse the chosen path in the control flow graph, using statements to reduce the variable domains. If a statement is not a control statement, then that statement is symbolically evaluated, potentially resulting in the modification of variable domains. If a statement is a control statement, the condition of the control statement is used to reduce the variable domains. However, if the current domains of the variables in the condition cannot result in the condition being evaluated with the proper truth value, then the domains are infeasible.

Handling of infeasible domains is closely related to the method used to reduce the domains. The heuristic used to reduce domains using constraints (boolean subexpressions in the condition) is to split overlapping domains in the region of overlap. For example, if x has a domain of -20 to 20, y has a domain of -10 to 35, and the boolean expression is x<y, the resulting domains might be -20 to 5 for x and 6 to 35 for y. Should the domains become infeasible, the algorithm backtracks to the most recent domain split where the split could have occurred at a different value, and domains are split using a new value as the dividing point. A parameter is used to determine the maximum number of splits that can occur for any given domain - when that number of splits is reached, the algorithm backtracks further to find a domain that can be split differently. If the algorithm backtracks all the way to the start node of the control flow graph, then a new path to the goal node is chosen, and the traversal of the graph following that path is attempted.

The main disadvantage of this approach is that loops are still not adequately addressed. Also, the backtracking involved with the method can be time consuming, and there is still no guarantee that data will be found even though data that will execute a path to the goal exists.

The Dynamic Approach

Another approach to automated test generation is described by Korel (Ref. 20), which uses a dynamic approach. The goal of this approach is to generate test data that will cause a specific path to be executed. The first part of the approach is to run the program on some set of data, observing the path that is executed and accumulating data flow information. The subpath that matches the desired path is noted, as well as the first control statement that resulted in deviation from the desired path. Then, a series of exploratory moves is made, followed by a series of pattern moves.

In the exploratory move phase, input variables that influence the first control statement result that deviated from the desired path are determined using data flow analysis. Executions are performed with all input variables remaining unchanged, with the exception of a small change to one of the influencing input variables. The executions are continued until either a small change results in the condition of the

control statement being closer to the desired truth value, or until all influencing variables have been attempted. If all influencing variables are attempted and the proper branch from the control statement is still not being taken, then the method terminates without generating test data.

In the pattern move phase, the variable that resulted in an improvement in the control statement condition is modified in the same direction, but by a larger amount. The amount that the value is changed increases until either a value is found that causes the desired branch to be taken, the execution does not cause the subpath leading to the branch to be executed, or the value does not make the condition of the control statement closer to the desired truth value. In the first case, the next control statement that results in a deviation from the desired path is found, and the exploratory and pattern moves are repeated for that statement. If the path executed is the desired path, then the current input data is the result of the method. In the second case, the size of the change to the value is reduced until a value is found that follows the subpath and makes the control statement condition closer to the desired truth value. Then the exploratory move phase is repeated. In the third case, the exploratory move phase is repeated.

There are two problems with this method. The first problem is that it may be difficult to determine whether the control statement condition is closer to yielding the desired truth value for control statements with complicated conditions. The second problem is that the search method used results in a local minimum as opposed to a global optimum, which may result in the failure to generate test data for some feasible paths.

The Chaining Approach

Korel refined his dynamic approach, and referred to the modified approach as the chaining approach (Ref. 18). There are two major differences between the dynamic approach and the chaining approach. The first difference is that in the dynamic approach, there is a path that is to be followed. However, in the chaining approach, there are certain control points (not necessarily all) that are to be required to follow a certain branch. The second difference is that the chaining approach generates event sequences and searches for input that satisfies those event sequences in the situations in which the dynamic approach fails to find an answer. An event sequence is a sequence of nodes that are to be executed, along with variables that must not have their values redefined before the next node in the event sequence is executed.

When the dynamic approach (or attempts to find data satisfying an event sequence) fails to find an input that results in a certain branch being taken, the node containing the associated control statement is identified as a problem node. The nodes where each variable in the control statement condition are last defined before reaching the control statement are used in generating one new event sequence for each node,

with the variables defined in that node as the variables that may not be redefined before reaching the next node in the event sequence. The method then tries to find an input on which the event sequence is traversed by executing the program with some input and observing the execution. If the event sequence is traversed successfully, then the input data has been found. However, if the input data causes a branch to be taken from which there is no subpath that reaches the next event sequence node without modifying the variables that are required by the event sequence to not be redefined, one of two actions are taken. If the condition of the control statement that resulted in the branch being taken involves only boolean variables, then the control statement node is identified as a problem node, the event sequence is modified to create new, longer event sequences, and the algorithm attempts to find data traversing one of the new event sequences. If the condition of the control statement involves comparison operators, then the algorithm uses direct search methods to attempt to find data that will cause the alternative branch from the control statement to be taken. The direct search method uses a measure of how close the condition of the control statement is to taking the alternative branch to guide the search.

The main question involving this approach is why the dynamic approach is attempted before switching to the approach of generating and attempting to traverse event sequences. Korel states that the search for input data used in the dynamic approach when an alternative branch needs to be taken fails often. The reason the dynamic approach is attempted first is probably because of differences in algorithm complexity. Attempting to successfully traverse an event sequence requires that data flow analysis be performed. One reason why the analysis must be performed is to identify the last definition nodes of variables in the conditions of control statements. The other reason is to identify what variables will be redefined based on the branch taken from control statements.

Assertion-Oriented Test Data Generation

Korel investigated using the chaining approach in a system that generates test data based on assertions embedded in programs (Ref. 19). The assertions were embedded into Pascal source code, using the special comment delimiters (*@ and @*). These assertions are then converted into executable statements by a preprocessor. Korel provides two kinds of assertions in his approach - boolean formula assertions, and executable code assertions. The boolean formula assertions are simply assertions that are restricted to constants, program variables, comparison operators, and logical operators. These are translated into conditional statements such that the condition will be true if and only if the assertion fails. Reaching the statement inside one of these conditional statements would indicate failure of an assertion, and therefore would expose a fault in the program. The executable code assertions are stated as the body of a function that assigns the value false to the special variable assert whenever the body detects that the assertion has failed. These assertions are translated into complete

Pascal functions that return the value of the special variable assert. The function is then called from the point of the assertion in a conditional statement, and the value returned is negated in the condition. Therefore, reaching the statement inside the conditional statement would indicate failure of an assertion, and therefore would expose a fault in the program.

To generate the test data, the chaining approach is used to attempt to generate test data that executes the statement inside the conditional statements for each assertion in the preprocessed program. If such data can be generated, then the data exposes a fault in the program (assuming that the assertion is stated correctly). One problem with this is that one kind of assertion that is often used is a statement of the precondition. This assertion is used to assure that the function is not called using values that are not in the domain of the function. However, this approach to test generation may generate data that would never be passed to the function, and therefore presents that data as exposing a fault. Another problem is that some assertions might be stated more clearly using the notation used in APP than by writing the body of a function, leading to fewer incorrect statements of assertions. Perhaps the notation of APP involving the `all` quantifier and the `some` quantifier could be added to this approach, reducing the cases in which bodies of functions would need to be written. The advantage of the approach taken by this assertion method is that is does provide the ability to specify assertions that are algorithmically-based, such as those involving the concept of permutation, which cannot be completely specified in APP. This method also suffers from the fact that the thought involved in using formal verification is needed in creating the specifications, but the results obtained are not as strong as those obtained from using formal verification.

5.2.4 Automated Selection of Regression Tests

Regression testing is verifying that changes made to software were implemented properly and did not have unintended side-effects. The specific area to be considered in AAV&V is, given a set of tests that are believed to provide adequate testing of the software before it was modified, how to select only the tests to run on the modified software that could possibly result in exposing an introduced fault. One problem with selective regression testing is that changes may be made to the software which are not adequately tested in the current test procedures. This means that additional tests will still need to be developed by the tester and the test procedures will need to be updated. Some selective regression testing methods determine a desired test coverage based on the software changes, making it possible to run the selected tests on instrumented software and determine whether additional tests are necessary to provide the desired coverage.

TestTube

TestTube (Ref. 3) is a system that uses previous test coverage information, determines changes made to the software, and selects the tests that previously covered the changed portions of the code for regression testing. When tests are rerun because of changes to the software, the new coverage of each test is recorded. The basic idea is that tests that do not execute any changed code cannot be affected by the changes made to the software.

One problem with this approach to selective regression testing occurs for languages with pointers and arrays that are not bounds-checked. With these constructs, it is possible to inadvertently alter the contents of some other variable of the program than was intended. This can result in the rerunning of all tests being the only regression testing strategy that is guaranteed to execute all tests that may expose an error.

RegTest

In the RegTest algorithm (Ref. 38), a history of what tests previous executed which sections of the program is also maintained and used. The algorithm first constructs program dependency graphs (PDGs) for the original program and the program with modifications. In this system, a PDG consists of a control flow graph (CFG) and a data flow graph (DFG), combined to form one graph. Next, the algorithm uses the PDGs and the test history to select tests that must be rerun in regression testing. Finally, the algorithm determines what parts of the program need to be covered by the regression tests.

In selecting tests, the PDGs are compared to locate new, deleted, and modified nodes. A naïve approach, taken by most regression testing methods, is to identify any tests that executed those nodes as tests that must be rerun. However, there are paths of execution that may execute (or had executed, in the case of deleted nodes) these new, deleted, or modified nodes, but the effects of the change have no effect. This is true when the node involves a definition of a variable, and there are paths that do not use that definition. The selection algorithm is designed to use data flow information, along with control flow information, to reduce the number of unnecessary tests that are selected.

To determine the sections of the code that must be covered, the algorithm uses the data flow information to locate def-use pairs that must be covered in regression testing. A def-use pair needs to be covered if either the definition node or the use node is new or modified, or if the def node is dependent (control or data, directly or indirectly) on some new or modified node.

The algorithm can be extended to work on entire programs instead of simply procedures by constructing a system dependency graph (SDG) instead of a PDG. The

algorithm, however, has the same problems as TestTube for programming languages that provide pointers and arrays that are not bounds-checked.

Semantic Guided Regression Test Selection

Binkley (Ref. 1) introduces the concept of selecting regression tests based on the semantics of the program. His approach involves two steps - constructing a reduced program that incorporates all of the affected points of the program, and selecting regression tests for testing the reduced program. The selected tests could then be used in executing the reduced program, saving time in regression testing by not only reducing the number of tests that must be run but also by eliminating execution that is guaranteed not to reveal a fault. Another advantage is that since the tests are being run on a smaller program, it should be easier to locate bugs should faults be exposed during the regression testing.

To construct the reduced program, directly affected points must first be identified. These are defined as nodes in the system dependence graph that are in the modified program but not in the original program, combined with nodes that have different intraprocedural edges in the two programs. The set of affected points must then be determined, and partitioned into strongly affected points and weakly affected points. An affected point is where a component in the in-line expansion of the modified program either does not have a corresponding occurrence in the in-line expansion of the original program or the corresponding occurrence computes a different sequence of values than the occurrence in the in-line expansion of the modified program when the programs are executed using the same input. A strongly affected point is either when there is no corresponding occurrence in the original program or when the in-line expanded programs compute different sequences of values when corresponding scopes of a procedure would be invoked with the same actual parameters. A weakly affected point is an affected point that is not a strongly affected point. Algorithms for determining these affected points, based on the algorithm for taking a forward slice with respect to the directly affected points, are presented. Next, a graph is constructed that captures the behavior of the affected points, based on the algorithm for taking a backward slice with respect to the strongly affected points and a partial backward slice with respect to the weakly affected points. The final step is to remove infeasibilities from the resulting subgraph. At this point, the graph represents a reduced program that captures the program components in which a semantic difference can occur.

To select the regression tests that are to be rerun, the algorithm first identifies new, deleted, affected, and preserved components. Tests that exercise only deleted and preserved components do not have to be rerun. For new components, the only tests of interest are those that that exercise some other component in the same calling context in such a way that this component and the new component have common execution patterns. A common execution pattern is when two components of two possibly

different procedures have a calling context from the two procedures such that they are exercised the same number of times on all input values. For affected components, test cases are identified in which a component of the original program with a common execution pattern is exercised when the test is run on it. Of these test cases, only those that execute a directly affected point in the original program need to be rerun. Two components have common execution patterns when the union of the calling context slices that are the control predecessors of the nodes associated with the components are isomorphic. A calling concept slice is constructed using part of the algorithm for a backward slice and information about parameter passing. So, to determine the tests that need to be rerun, it needs to be determined for each affected and new component whether the original program had a component that had a common execution pattern. If so, and that component is not already being exercised by a selected test, then select a test case that exercised that component in the original program that also exercises a directly affected point in the program.

There are several problems with this method. One is that the selection is not safe as defined by Rothermel and Harrold (Ref. 39). To overcome this fault, all tests that executed a component in the original program that had the same execution pattern as a new or affected component could be chosen. The reason this is not done in the original algorithm is that the goal of the algorithm is to achieve some desired coverage of the affected portions in the modified program. The algorithm is designed to work where parameters are passed by value-result, the program does not contain arrays or data structures, there are no reference variables or pointers, and there are no global variables. These assumptions are fairly limiting. Another assumption is that the current test suite for the original program provides the desired coverage of the program, and that achieving that level of coverage is enough to certify the program. It is also assumed that the program will produce the same output each time it is run with the same input, which might not be true for real-time avionics software.

Selective Regression Test Data Generation

In (Ref. 14), the use of slicing in selective regression testing is discussed. This paper suggested that instead of using the information about the changes made to the program to select existing tests, the information could be used to assist in generation of tests to provide the desired coverage of the program changes. These tests could be generated manually, but could also be generated by automated means. To simplify the process of generating tests, slicing techniques could be used to reduce the program to the slices necessary to develop the needed test cases. This method has the advantage that the program on which selective regression testing is to be performed does not need to be instrumented to determine the execution profile of each test. Also, since the test cases generated cause the desired test coverage to be met, there is no need to instrument the program to determine if that coverage was met or to determine code that has not been adequately tested. This eliminates any concern about selective regression testing

affecting the real-time performance of the code or increasing the size of the code such that it would no longer fit into the available memory. The method, however, does still have drawbacks in the area of arrays and pointers permitting the modification of memory beyond the bounds of arrays.

5.2.5 Formal Methods

A formal method is a method that has a mathematical or logical basis. Formal methods may be used to specify requirements of a system, or to model the design of a system. A formal methods system often includes a theorem prover or model checker that can be used to ensure that certain properties are satisfied by a formal specification. An extensive use of formal methods would be to formally state requirements, verify that the requirements are complete, consistent, and guarantee desired properties, formally state the design and verify that the requirements are satisfied, and develop the implementation and verify that the design is satisfied.

The formal method we are most interested in is formal verification of source code. A popular way to formally specify the implementation and what is to be proven about it is through the use of Hoare sentences (also known as Hoare triples). A Hoare sentence consists of three parts - the precondition, the implementation, and the postcondition. A precondition is a formal statement of conditions that must be true for it to be valid to use the function or procedure in the implementation. Examples of preconditions would be `delta_x <> 0`, or "the actual parameters for `x` and `y` are not the same reference". A postcondition is a formal statement of conditions that are to be true when a call of the function has just returned, provided that the precondition was satisfied when the function was called. Examples of postconditions would be "slope has the value of `delta_y` divided by `delta_x`", or "the reference for `x` now contains the value that was contained in the reference passed for `y`, and the reference for `y` now contains the value that was contained in the reference passed for `x`". The implementation is simply the source code for the function or procedure that is to be formally verified, perhaps translated into a formal notation.

EHDM

EHDM (Refs. 11, 40) is a theorem prover that provides the capability to state and prove Hoare sentences. The implementation is specified as a formula that relates the signature, or prototype, of a procedure to its implementation, which is specified using an Ada-like syntax. The procedure signature, with actual parameters used in place of the formal parameters, may then be viewed as another operation in the programming

language. Program variables are handled by defining state variables in EHDM, where the type of the object that is to be contained in the state object is also specified.

One problem with EHDM is conducting a proof using formulas that contain quantifiers. EHDM is not capable of finding the correct instances to be used in place of the quantifiers, so the person developing the proof must provide the instantiation to EHDM. This means that the user must already have a clear picture of how the proof will proceed. This limitation is not present in many other theorem provers, such as PVS (Refs. 8, 32, 41) (which, like EHDM, was developed at SRI International). One additional concern about EHDM is that no mention is made of how looping constructs are handled by the theorem prover. This is a concern because most software verification approaches require that a loop invariant be provided. A loop invariant is a formal statement that must be true at the beginning of each iteration of the loop. The disadvantage of requiring a loop invariant is that it is one more formal statement that must be provided by the user before the formal verification may be attempted.

EVES

EVES (Refs. 2, 21) is another verification system that is capable of formally verifying the correctness of computer programs. EVES provides two different syntaxes for writing formal specifications. The original syntax of EVES, Verdi, has a LISP-like syntax. A Pascal-like syntax, s-Verdi, is also provided for EVES. An example of an EVES specification, in s-Verdi, for an integer square root procedure (Ref. 21) is shown in Figure 5-4. Mention is also made of supporting user-defined syntaxes, although no details of how this may be done is provided. The theorem prover for EVES is called NEVER, and contains a relatively small set of proof commands.

EVES has several disadvantages. First of all, loop invariants and measures are

```
procedure iroot (lvar x: int, pvar y: int) =
  pre 0 <= x and x <= maxint
  post y^2 <= x and x <= (y+1)^2
  begin
    y := 0
    loop
      invariant 0 <= y and y^2 <= x
      measure x - y^2

      exit when not y * y < x - 2 * y
      y := y + 1
    end loop
  end iroot;
```

Figure 5-4 Example of an EVES s-Verdi Specification

required to formally verify code containing looping constructs. A measure is a formal statement that serves as an upper bound on the number of iterations (or, in the case of recursion, recursive calls) that will be performed, and is necessary for proving termination of the program. Second, when formal verification is performed, a verification condition is formed. This verification condition includes proof of termination, proof of proper domains for variables, and proof of type correctness. The output of the verification condition by NEVER does not have a clear, direct correspondence to what is trying to be verified. This is compounded with detailed output from NEVER about the proof process, which would be meaningless and potentially confusing to users who are not expert users of theorem provers. Some theorem provers, such as PVS, provide the capability for the user choosing to suppress the detailed output of the proof process.

Penelope

Penelope (Ref. 26) is a formal verification system that was developed under STARS (Software Technology for Adaptable, Reliable Systems) to formally verify software written in a subset of sequential Ada. Penelope specifications are written in Larch/Ada and may make use of Larch Shared Language (LSL) traits, which are formal descriptions of abstract concepts. A Larch/Ada specification also consists of three parts - the precondition, the implementation, and the postcondition. An example of a Larch/Ada specification for swap is shown in Figure 5-5. Formal verifications are performed in Penelope by assuming that the postcondition must be true, then using the code to determine what must be true at the beginning of the function or procedure for the postcondition to be true, and then finally determining that this calculated precondition is implied by the stated precondition. The statement that the calculated precondition must be implied by the stated precondition is the verification condition of the proof. Penelope provides a sizeable number of proof commands that the user may use in proving the verification condition. However, groups of these proof commands

```
procedure swap(x : in out integer; y : in out integer)
  --| where
  --|   out x = in y;
  --|   out y = in x;
  --| end where;
is
  temp : integer;
begin
  temp := x;
  x := y;
  y:= temp;
end swap;
```

Figure 5-5 Example of a Larch/Ada Specification

are similar to each other, which can cause confusion about what command is the correct one to apply in a given situation.

One problem with Penelope is that it also requires that each loop have a formally stated loop invariant. The invariant is treated as a precondition for the loop, which results in the formation of multiple verification conditions. For example, if a program has one loop, then it must be shown that the loop invariant must be true if the stated precondition is true, and it must also be shown that the postcondition must be true if the loop invariant is true. Another problem, as mentioned above, is the numerous, confusing, proof commands that must be mastered to perform a proof. One further problem with Penelope is that it requires familiarity with Larch Shared Language traits, and users need to know not only how to use these traits in specifications but also may need to know how to create their own traits. One nice feature is that Penelope embeds information about the proof in the original source code. A major reason this is done is so the proofs can be rerun automatically if minor changes are made to the code. However, a better solution might be to simply note in the source code the extent to which a subprogram has been formally verified and the date, and place the detailed proof information in a separate file for use by Penelope.

Hoare Rules for real-time

Effort has been made to expand Hoare sentences, and the Hoare rules used to manipulate them in the proof process, to work for real-time programs (Ref. 15). These modifications permit the expression of the concept of time in preconditions and postconditions, as well as communications over channels. Real-time programs are often designed to not terminate, but the classic Hoare rules are designed to prove conditional correctness - that if the program terminates, the postcondition will be satisfied. These rules have been modified to handle formal verification of programs that may not terminate, and the terminology of precondition and postcondition has changed to assumption and commitment, respectively.

This modeling of real-time, multi-process, communicating systems makes several assumptions. One assumption is that each process runs on its own processor, which often is not the case in real-time systems in the real world. Another assumption is that the time involved in executing each statement, such as assignment, is known, as well as the time involved in the overhead of while loops and conditional statements. In actuality, these may vary widely based on the complexity of the expressions within the statement. There are also assumptions associated with communications. Communications are assumed to occur asynchronously over unidirectional channels that connect two processes. Messages are not buffered - if the receiver is not attempting to read a message, then the message is lost. However, if the process attempts to read a message and no message was sent, it will wait for the next message that is received.

The programming language used to specify real-time programs was created for use with the modified Hoare sentences and proof rules. The language tends to describe the abstract behavior of the implementation as opposed to stating the actual implementation that is used in the real-time program. Verifications have been performed using the PVS theorem prover, but the level of automation is not mentioned. Also, these verifications are of an abstraction of the implementation, which is on a higher level than the actual implementation is likely to be.

Tecton

Tecton (Refs. 16, 17) is a system that is capable of doing automated formal verifications of functions. Tecton presents its output in graphical format as a proof tree. To keep the proof tree compact, extra tables are created and maintained by the verification system. Also, only the information that is necessary to follow the logic of the proof is provided for each proof step - this information consists of the proof rule that was applied, and any additional formulas from the tables that were used. The proof trees can still get quite long, so the proof is broken into pages, with hypertext links maintaining the connection of the proof tree.

Proof goals are stated as Hoare formulas, in a three-column format. The first column is the precondition, the second column is the formal representation of the function, and the third column is the postcondition. Hoare-style proof rules may be automatically applied to generate chains of subgoals that terminate when the formal representation of the code has been eliminated using back substitution.

The program, and preconditions, postconditions, invariants, and assertions, are stored in a program table. Each statement and assertion has a unique symbol associated with it. The symbol is then used in the Hoare formulas in place of the statement or assertion to minimize the size of the proof trees. The assertions can contain conjunctions and implications. Invariants are required for each loop in the program, and state what is true at the beginning of each iteration of the loop. Assertions, while not necessary, are often useful in the proof to provide a point where the proof tree can be split. During the proof process, each invariant and assertion causes two branches to be generated in the proof tree, as do conditional statements.

A rule table appears on each page of the proof, and lists each rewrite rule that is used on that page. A rewrite rule is a statement about the equality of two expressions. Rewrite rules can be made conditional by specifying the conditions under which the rewrite rule applies as a logical formula. Any variables used in the rules table are implicitly universally quantified.

A parts table may also appear on each page of the proof. The table may contain either formulas or terms, and is used to reduce the size of the subgoals displayed in the proof tree. This table may also contain logical implication formulas.

Tecton has several flaws. One is that it only works on variables with integer types. Also, the back substitution method used to process the program statements requires that loops have loop invariants. At times, the user needs to state and prove lemmas in order for the formal verification of the function to be completed. It is also not capable of dealing with some of the concepts used in sophisticated programming, such as dynamic memory allocation, pointers, and references. Many of the ideas used in the user interface, however, could be useful in the design of a more sophisticated automated formal verification system.

MELAS

MELAS (Refs. 42, 43) takes a novel approach to performing formal verification. MELAS executes the unit under test inside a debugging environment and provides a way to verify the satisfaction of preconditions and the postconditions resulting from that execution. Instead of executing using actual values, symbolic values are used in the execution of the function. MELAS is connected to an inference engine via run-time analysis oracles.

MELAS was developed with the intention of using it to formally verify the correctness of function templates written in C++, such as those provided in the C++ Standard Template Library (STL) (Ref. 29). Therefore, the run-time analysis oracles take the form of C++ classes. Object instances of these classes, represented with symbolic values, are then created by a test driver function, and the function to be tested is called. A key point to be made is that the original code of the function being tested did not have to be modified for it to be symbolically executed. The run-time oracle is mainly used to define operators involving the symbolic values. When an oracle is called, such as when two symbolic values are being compared, the inference engine is used to determine if a value can be inferred from the existing facts. If it can infer a value, then that value is returned for the comparison operator. If it cannot infer a value, the user may be prompted to make a decision for the comparison operation, the response is recorded as a fact for later use by the inference engine, and the user's response is returned for the comparison operator. An alternate approach, necessary for formal verification, is to use a case analysis approach to require that both possible results be attempted before the function is considered verified. In this approach, an analysis database is used to track possible results that have not yet been successfully verified. MELAS provides a set of run-time analysis oracles for iterators used in STL as well as for different kinds of relations.

Besides writing the driver program, the user may have to develop additional run-time oracles if the ones provided are not sufficient. One additional requirement on the user is that he or she provides a specification of the function that is being verified. This specification is written as a C++ class template, and has private data members for all parameters of the function as well as data to which those parameters may be iterators. The class also has three public member functions, all of which take the same parameters that are passed to the function that is being formally verified. One function, called `precond`, first determines if the precondition of the function has been satisfied, and then stores the information involving the parameters if it has. Another function, called `postcond`, uses the state of the parameters at the end of execution and the values saved in the private data members by the `precond` function to determine if the postcondition has been satisfied by the execution. The final function, called `post_update`, has two uses. It can be used to apply the postconditions of the function to the current execution state without executing the function. This is useful for formally verifying functions that call other functions. A second use of the function is in the formal verification of functions that contain loops, which is done by induction. The program is first symbolically executed without entering the loop. The program is then symbolically executed again, executing the body of the loop one time. The `precond` function is then called (on a new instance of the class template) to evaluate the precondition and save the current values of the formal parameters and any associated memory that they are iterators to. Then the `post_update` function is called (using the same new instance) to apply the inductive hypothesis to the state of the symbolic execution. Finally, the `postcond` function is called to verify that the postcondition has been satisfied for the inductive case.

One disadvantage of this approach is that the user must specify not only preconditions and postconditions, but must also state an induction hypothesis as well as associate function calls with breakpoints in the source code being formally verified. There is also a fair amount of setup work involved in using MELAS, although some of it could probably be eliminated. Another disadvantage of this approach is that it will only work for generic functions where all parameters have generic types - this is necessary for the run-time oracle. If the parameters are not all generic, then the code must be changed to make them generic or to hard-code a type for symbolic representation and the run-time oracle. The language must also support operator overloading, including assignment, and encapsulation, such as classes in C++ or packages in Ada. Given this, there are still faults, such as not detecting that memory not allocated for use by the program has been accessed or changed (unless the program crashes or the fault causes the postcondition to not be satisfied). Finally, the debugger used must have certain capabilities, such as setting conditional breakpoints, attaching function calls to breakpoints, disabling and enabling breakpoints, setting variables, and operating from a user-supplied script. The main advantage of MELAS is that it allows formal verification to be performed in a setting that software engineers are more familiar with than theorem provers - symbolic debuggers.

AVaSE

AVaSE (Ref. 5) is a proof-of-concept formal verification system that is built upon PVS (Refs. 8, 32, 41). The goal in designing AVaSE was to support complicated concepts supported by programming languages, such as arrays, pointers, dynamic memory allocation, aliasing, and abstract datatypes. The system was to be able to perform generic verifications of generic functions and containers so that the time involved in formal verification could be amortized over the numerous instantiations of the component. The C++ Standard Template Library (STL) (Ref. 29) was used for examples of generic components to formally verify. An additional goal in designing AVaSE was to minimize the knowledge required of the user in the area of automated theorem provers. Proof strategies, which are like proof subroutines, are used to eliminate the need to be familiar with the commands of the theorem prover - a proof can be performed using just one command.

AVaSE also contains a set of definitions and axioms that are used to perform the formal verification using forward symbolic execution. Musser and Wang (Ref. 28) developed axioms to perform formal verifications by hand, but these axioms were not stated in a way such that they could be automated. However, the general concepts and the notation were used in AVaSE. For example, to avoid the need for loop invariants, Musser and Wang used induction to symbolically execute loops. AVaSE also uses an inductive scheme for symbolic execution of loops, although the induction is modelled as recursion.

The major part of the work required of an AVaSE user is the formal specification of what is to be proven. The user must state preconditions, postconditions, and formalized implementations. The preconditions and postconditions state predicates used to relate values to one another and a memory function that relates variables (or array indices) to their contents. The formalized implementation supports basic operations found in most programming languages, such as assignments, conditionals, while loops, declarations, and function and procedure calls. AVaSE also requires that the user provide additional information, such as how to handle calls of the function or procedure that is being verified, as well as an induction hypothesis for functions that contain loops.

If the program involves abstract concepts such as lists, then the user must also provide a set of axioms, written in the PVS specification language, which formally describe those concepts. These axioms also define manipulations that involve the abstract concepts, such as appending lists.

There are a number of problems with AVaSE. One problem is that AVaSE is slow. This appears to be caused by the decision procedures of PVS - PVS is very

powerful, but that power can also make it slow. In this case, much of the power of PVS is unnecessary, and developing a specialized system that incorporates only the needed proving power should dramatically improve speed. An additional problem is that formal statements beyond preconditions and postconditions must be formulated. However, the formal implementation was chosen for ease of representation in the PVS specification language, and the description of how to handle a call of the function or procedure follows directly from the preconditions and postconditions. Therefore, in a specialized system, the formal implementation could take some other form (perhaps IRIS), and how to handle a call to the function or procedure could be derived automatically by the system. There are also problems related to programming language constructs. One such problem is while loops - whereas loop invariants are not necessary, a function containing a loop has the restrictions that it cannot contain nested loops and that any processing preceding the loop must consist only of trivial initializations. An additional problem is that some language standards leave some decisions to the compiler vendors that could alter the result of statements depending on the decision that is made. An example of this from the C++ programming language would be a statement in which two references are incremented - if those reference are actually the same reference, the result of the statement can differ depending on the order of execution of the two increments. AVaSE depends on the user to indicate in the formal specification of the function that there is an indeterminate ordering of execution of subexpressions that have side-effects so that all possible orderings of execution can be used in the symbolic execution. It might be possible for a specialized system that handles only one programming language to automatically detect such statements and generate the cases. However, the advantage of using a general formalization for the function, such as IRIS, would be that the same system could be used to formally verify code written in any programming language that can be translated to IRIS[4]. One final problem is that the user may need to define abstract concepts for use in proofs using a formal specification language. Most systems, such as Penelope (Ref. 26) (through the Larch Shared Language traits) and Tecton (Refs. 16, 17), provide libraries of abstract concepts for use in specifications of preconditions and postconditions. Unfortunately, it is impossible to provide a system containing all of the concepts that any user may ever need. Also, providing the libraries is not enough - the user must know what is in them and how and when they are to be used.

5.3 EVALUATION OF METHODS

In Phase I of the AAV&V program, a literature survey of verification and validation techniques was performed (Ref. 22). To evaluate the literature, a set of

[4] Because a translation from a programming language to IRIS may require the creation of new IRIS nodes or some modifications of existing IRIS nodes, it may be necessary to perform some minor modification of the existing formal verification system before it can support programs written some other programming language.

formal evaluation criteria was developed. Many of the criteria of that survey are applicable to the survey that we have performed, and therefore appear in our formal evaluation criteria. However, some criteria that either were adequately addressed by ImpACT or did not apply to software change verification and validation were omitted. At the same time, some additional criteria, specific to the task of software change verification and validation, were added to the existing criteria. In our evaluation criteria, the categories remained Technical Effectiveness, Automation Requirements and Restrictions, and Effort Required to Perform Verification and Validation. All sub-categories, with the exception of Fault-Tolerant, were also retained. A table summarizing the evaluation criteria used for evaluating software change verification and validation methods is provided in Table 5-1.

This section takes a look at the evaluation criteria that were used to evaluate the methods located during our literature survey. These criteria were then used in designing a database using Microsoft Access. Each method was then evaluated, and those evaluations were used in populating the database. These evaluations, formatted as reports generated using the database, may be found in Appendix B.

5.3.1 Technical Effectiveness

The evaluation criteria for technical effectiveness determine how well a candidate method for software change verification and validation supports the needs of the activity. The criteria in this category are divided into four subcategories. These subcategories, and the criteria within each of them, are discussed in the following subsections.

Real-Time Performance

Nearly all of the methods for automated support of testing provide no direct support for issues involving real-time performance. The only real-time issue directly supported by any of the methods is that of satisfaction of timing constraints. These constraints specify the maximum amount of time between events of interest in the system.

Table 5-1 Evaluation Criteria Summary

CATEGORY	SUB-CATEGORY	AREA	EVALUATION CRITERIA	METRIC
Technical Effectiveness	Real-Time Performance	Timing Constraints	Can the candidate method take into account timing constraints?	Yes/No/Partially
	Software Characteristics	Metric	Does the candidate method provide a metric for quantifying results?	Yes/No/Partially
		Compatibility with Specification	Can the candidate method determine if the software performs according to a given specification?	Yes/No/Partially
		Correct Algorithm Implementation	Can the candidate method determine if the algorithms are correctly implemented in the software?	Yes/No/Partially
		Functions with Infeasible Paths	Does the candidate method adequately address problems with infeasible paths?	Yes/No/Partially
	Requirements/ Features	Assumptions	What are the assumptions associated with the candidate method?	Descriptive
			How restrictive are the assumptions associated with the candidate method?	Slightly/Moderately/ Largely
		Oracle	Can the candidate method evaluate software performance without the need for an oracle?	Yes/No/Partially
			If an oracle is necessary for the candidate method, what is the level of complexity of the oracle?	Descriptive
		Restrictions on Software Requirements/ Specifications	What restrictions/conditions are placed upon the software requirements/ specifications to allow use of the candidate method?	Descriptive
		Modifications to Tested Software	What modifications to the software under test are required to implement the candidate method?	Descriptive
		Adverse Effects	Can the candidate method be used without requiring modifications that might result in residual adverse effects on the tested software?	Yes/No/Partially
			If modifications are required that might result in residual adverse effects, what are the modifications that are required and the effects?	Descriptive
		Always Yields Result	Will the candidate method always yield a result?	Yes/No
			If the candidate method does not always yield a result, under what conditions does it not yield a result?	Descriptive
		Retest Minimization	Does the candidate method attempt to reduce the amount of retesting/ reverification necessary?	Yes/No/Partially
		Code Minimization	Does the candidate method minimize the amount of code to which the method is being applied?	Yes/No/Partially
		Coverage	Does the candidate method guarantee a specified coverage level for retesting/ reverification?	Yes/No/Partially
		Level of Assurance	What level of assurance does the candidate method provide?	High/Moderate/Low

Table 5-1 Evaluation Criteria Summary (Continued)

CATEGORY	SUB-CATEGORY	AREA	EVALUATION CRITERIA	METRIC
Effort Required to Perform V&V	Time/Skill	Human Interaction	How much manual effort* is required to perform an analysis with the candidate method?	Small/Moderate/ Large
		Skill Level	How much additional user training is required to support the candidate method?	Small/Moderate/ Large
		Computer-Based Analysis	How much computer-based analysis* is required to perform software V&V with the candidate method?	Small/Moderate/ Large
		Total Time	What is the total duration for performing an analysis* with the candidate technique?	Small/Moderate/ Large
*Effort normalized based on fixed amount of code.				
		Enhancement/ Extension	Is the candidate method amenable to enhancement or extension?	Yes/No/Partially
			If the candidate method is amenable to enhancement or extension, to what extent?	Descriptive
		Usable with Other Techniques	Can the candidate method be combined with other techniques?	Yes/No/Partially
Automation Requirements/ Restrictions	Hardware/ Software	Implementation Hardware	What hardware is required to implement the candidate method?	Descriptive
		Test Hardware	What special-purpose test hardware (e.g., hardware-in-the-loop) is required to implement the candidate method?	Descriptive
		Software	What software or auxiliary software tool (e.g., semantic analyzer, theorem prover, etc.) is required to implement the candidate method?	Descriptive
	Test Data	User Input	What information is required from the user to implement the candidate method (e.g., software specification, timing requirements, etc.)?	Descriptive
			What is the format of the information required from the user for the candidate method?	Descriptive
		Test Generation	Can the candidate method assist in the development of test cases?	Yes/No/Partially
			If the candidate method can assist in the development of test cases, how?	Descriptive
		Probe Data	What data does the candidate method generate or require to probe the software/hardware under test (e.g., test cases, test inputs)?	Descriptive
		Output Data	What data does the candidate method require from the software/hardware under test (e.g., program responses to inputs)?	Descriptive
	Miscellaneous	Diagnostics	What diagnostics are provided by the candidate method?	Descriptive
		Effort	How much effort is required to automate the candidate method?	Small/Moderate/ Large

Software Characteristics

This sub-category of criteria covers the functional and structural characteristics of the software. These criteria are mainly concerned with the critical areas of software requirements and implementation: software correctness and control paths.

The *Metric* criterion indicates whether the method provides an objective measure of the reliability of the software. This metric indicates how well the software has been tested, with respect to some criterion for sufficient testing. Often, this metric involves measuring the tests performed against a specified coverage requirement.

Compatibility with Specification evaluates whether or not the method verifies compatibility of the implementation with some form of specification. This specification could be a requirements specification, a detailed design, or some form of formal specification.

Correct Algorithm Implementation evaluates whether or not the method verifies that the implementation correctly implements the algorithm. The algorithm must be specified for use by the method.

Functions with Infeasible Paths evaluates the ability of the method to function when the source code contains paths that are infeasible path. An infeasible path is a path through the code for which no set of values exists that will cause that path to be executed.

Requirements and Features

This subcategory of criteria addresses implicit and explicit conditions that are necessary to use the candidate method. It also addresses special features that may be provided by the method.

The *Assumptions* criterion describes any assumptions made by the candidate method, and evaluates how restrictive those assumptions are.

The *Oracle* criterion indicates if the user must provide an oracle against which the results of the program are compared. If an oracle is required, then the complexity of the required oracle is also described.

The *Restrictions on Software Requirements/Specifications* criterion describes the restrictions on requirements or specifications that may be used by the candidate method. For example, formal verification methods require that a specification be provided that is written in a formal language, and the formal language is typically

specific to the formal verification system being used. These formal specifications are often difficult to write in a manner such that high reliability is provided yet the specification is at a sufficiently high level of abstraction.

The *Modifications to Tested Software* criterion describes the modifications to the software that may be necessary to be able to apply the candidate method. In particular, some methods require that the code be instrumented, either to determine proper functioning at certain program points or to indicate coverage of certain aspects of the code.

The next criterion, *Adverse Effects*, is closely related to modifications to the software. For example, adding instrumentation to the program will increase the time taken to execute the code, potentially interfering with critical timing aspects of the code. In addition, this instrumentation results in a larger executive that will require more memory, which could be in short supply.

The *Always Yields Results* criterion indicates whether the candidate method always provides a result. If the method does not always provide a result, the situations in which a result may not be provided are described. For example, some approaches to automated test data generation may not yield a result if there are only a limited number of data sets in domain that result in the desired code elements being executed.

The *Retest Minimization* criterion indicates whether the candidate method minimizes the amount of retesting of the software that must be performed. Similarly, the *Code Minimization* criterion indicates whether the candidate method minimizes the amount of code that is considered for verification and validation.

The *Coverage* criterion indicates whether the candidate method guarantees that a certain level of coverage is attained through utilizing the method. The strength of this criterion is tied to the level of coverage that is guaranteed.

The *Level of Assurance* criterion indicates the confidence that application of the method provides that the software is correct. Formal verification, for example, provides a relatively high level of assurance (with respect to the reliability of the formal specification), whereas test data generation methods provide relatively low levels of assurance since the data generated often is not representative of data that occurs during real executions of the software.

Usability

This subcategory of criteria evaluates the ability to apply the candidate method to the program. This subcategory also indicates whether and, if so, how the method can

be enhanced, extended, or used with other methods to increase its applicability to the program and to the software change verification and validation task.

The *Applicability* criterion identifies whether the candidate method can be applied to the program in its entirety. Also identified is whether the candidate method can be applied to selected portions of the program.

The *Scalability* criterion is important in considering a candidate method for recommendation. It describes how the method performs as the size of the program (or portion of the program) being analyzed increases. This criterion considers not only algorithm complexity issues, but also considers issues directly related to the person performing the analysis (such as increase in complexity of specifications).

The *Levels of Test* criterion discusses the level at which the testing is performed. Often, this is described using the level of coverage provided or attempted by the method.

The *Enhancement/Extension* criterion indicates whether the candidate method can be enhanced or extended. Enhancements could result in a method that provides improved results. Extension could extend a method to be more applicable, such as by permitting application to entire programs as opposed to only single functions. Any enhancements or extensions deemed possible are described in some detail.

The *Usable with Other Techniques* criterion is used to indicate whether the technique can be used in conjunction with, or combined with, other methods to provide improved results. These other methods are often mentioned in the enhancement/extension criterion.

5.3.2 Automation Requirements and Restrictions

The evaluation criteria for automation requirements and restrictions describe specific needs of the candidate software change verification and validation method. The criteria in this category are divided into three subcategories. These subcategories, and the criteria within each of them, are discussed in the following subsections.

Hardware and Software

This subcategory identifies any special hardware and software needs that the candidate method has. These needs include any special platform needs of the candidate software change verification and validation method.

The *Implementation Hardware* criterion identifies any specific hardware that is required in implementing the candidate method. An example of an implementation hardware need would be if a method could only be run on a graphics workstation.

The *Test Hardware* criterion identifies any special-purpose testing hardware that is needed by the method. An example of a test hardware need would be if special hardware were required to simulate input from the external environment.

The *Software* criterion identifies any special software required to implement the candidate method. Examples of such software include theorem provers and code instrumentation software.

Test Data

This subcategory specifies the restrictions and requirements that the candidate method places on data. The data considered is not only the input data, but also the output data. This subcategory also indicates whether the method can develop its own test data for executing the program.

The *User Input* criterion describes the input required from user by the candidate method, as well as the required format of the input. An example of such requirements would be a formal specification of the function, written in a specific formal specification language.

The *Test Generation* criterion indicates whether the method generates its own tests. In particular, some methods generate test data that will cause some specific execution pattern in the software to be exercised. Other methods, however, do not require any test data in order to be applied.

The *Probe Data* criterion indicates any probes that must be generated or provided for use by the candidate method. An example of such probes would be instrumentation that is added to the code being tested to determine the code that particular tests execute.

The *Output Data* criterion indicates the output that the program being tested is expected or required to produce. This output data could include the normal output of the program.

Miscellaneous

This subcategory evaluates issues involving restrictions and requirements for automation of the candidate methods. These issues are diagnostics and effort.

The *Diagnostics* criterion discusses diagnostics provided by the candidate method. These diagnostics could be used in determining the cause of a failure to verify and validate a change made to the software.

The *Effort* criterion evaluates the level of effort that is involved in automating the candidate method. All evaluations assume that the method is not currently implemented, or that the implementation is not available to TASC.

5.3.3 Effort Required to Performed Verification and Validation

The evaluation criteria for effort required to perform verification and validation measure the level of effort associated with applying the candidate method. The only subcategory is time and skill.

Time and Skill

The time and skill subcategory measures the time and specific skills required of a user of the candidate method. Also measured is the time required by the computer, as well as an overall measure of time associated with the method.

The *Human Interaction* criterion evaluates the amount of time required on the part of the user in order to apply the method and evaluate the results. This time includes the time associated with preparing specifications and test data, as well as the time associated with examining the results of the testing or verification to determine if the results were as expected.

The *Skill Level* criterion evaluates the skills that a user of the candidate method must possess. Examples of specific skills that may be necessary include writing formal specifications or using theorem provers.

The *Computer-based Analysis* criterion evaluates the time required by the computer to perform its part of the candidate method. This time includes any automated preprocessing, execution of the test, and any determination of the results.

The last criterion is *Total Time,* and addresses the total time associated with using the candidate method. This takes into consideration both the human time and the computer time. In general, the human time is the more important of the two components, although time associated with complex algorithms may need to be taken into consideration.

5.4 RECOMMENDATION

Each of the areas covered in the survey has aspects that prevent them from being the clear choice. In this section, we discuss the general problems associated with each area, and then make a recommendation for an approach to automated support of software change verification and validation.

The first area discussed in Section 5.2 was coverage criteria. While this is not itself an approach to software testing and verification, it does provide some means of measuring testing that has been performed. Unfortunately, the way the coverage is determined is by instrumenting the code to record elements that are covered during the testing. This instrumentation adds executable instructions to the program being tested, which may result in real-time requirements not being met by this instrumented program. In addition, the size of the executable will be increased, which may be a problem in embedded systems where the executive, before instrumentation, utilizes nearly all of the memory. For these two reasons, determining the coverage that results from testing may be problematic for OFPs.

The second area discussed in Section 5.2 was assertions and annotations. This area assists in determining when a test exposes a fault in the program, but does not generate the test data that causes the fault to be exposed. Code (or annotations, which are then translated into code) must be added to the program, resulting in the same problems as discussed above for coverage criteria. In addition, the assertions must be formulated and added to the code by the test personnel, resulting in additional work that these persons must perform. For these reasons, the use of assertions or annotations is not being recommended for the verification and validation of existing systems to which changes have been made. However, it is recommended that those developing new systems consider the use of these constructs to increase the reliability. If real-time performance or limited memory are issues, conditional compilation could be used so that only assertions of interest are compiled into the executive and so that the executive released contains none of the assertions.

The third area discussed in Section 5.2 was automated generation of test data. Methods in this area resulted in the generation of test data that covered specified elements of the code. Problems associated with this area are numerous. Of greatest concern is that most of the methods surveyed appear to be applicable to either individual subprogram units or small programs, but are not likely to scale well to the size and complexity that is typical of an OFP. Also of concern is that these methods too frequently fail to locate test cases that cover the specified elements of the code when such test cases do exist. Perhaps some methods could be extended to work for sizeable programs. These methods could then be alternatives to be tried in generating test data – if one approach does not find suitable data, then other approaches could be tried.

The fourth area discussed in Section 5.2 was selective regression testing. In this area, information about the changes made to the program and a database mapping tests to code sections are used to select tests that need to be rerun. To determine what code sections are executed by the tests, the code must be instrumented. Again, this results in the same problems that were discussed above for the coverage criteria methods. In addition, there is no guarantee that simply rerunning the selected subset of tests from the test database will be sufficient to test the OFP with the modifications applied to it - additional test cases may be necessary.

The final area discussed in Section 5.2 was formal verification. This area resulted in the highest reliability of any of the methods surveyed, but required substantial effort on the part of the person applying the method. The method requires that a formal specification of the function being formally verified be developed, which involves significant time. Writing the specification also requires substantial skill and training, more so than any of the other areas in the survey. Therefore, while not recommended for maintenance of existing systems, formal verification should be considered by those developing new systems. The creation of the formal specification could be incorporated into the detailed design phase, and the software could be formally verified as correct before ever being executed. Another concern associated with formal verification is how well the methods would scale, particularly with formal specifications that contain significant detail so as to provide high assurance. Finally, formal verification does not eliminate the need for testing. The formal verification could consider the hardware to have infinite precision with no hardware "bugs", and also assumes that the compiler generates object code that is consistent with the semantics of the source programming language. Therefore, the software would still need to be rigorously tested to address these limitations and assumptions. Formal verification should be used to complement software testing, not to replace it.

As the preceding discussion would suggest, it was difficult to develop a recommendation to an approach for software change verification and validation that meets the needs of maintainers and testers of OFPs. If not for the concern about real-time performance and limited available memory, the recommendation would be to perform selective regression testing combined with test coverage analysis. The coverage analysis would report any deficiencies in the regression testing that was performed. The user could then develop additional tests manually, or could employ a test data generation system to attempt to automatically generate test cases that would address the deficiencies in the regression testing that was performed. In either case, the new tests would need to be added to the test database used by the selective regression testing component of the approach.

One problem with the approach described above is that a database that maps tests to code executed by the tests must be maintained, requiring code instrumentation. The second problem with this approach is that the code elements covered during testing

must also be tracked, also requiring code instrumentation. However, if the system did not first run tests selected from an existing database of tests, both reasons for instrumenting the code would be eliminated. Essentially, we would neither select nor execute existing tests, which could be viewed as always selecting no tests from the existing test database. Instead, all code elements would be considered to be uncovered, and an automated test data generation system would be used to generate test data that would cause the elements to be covered. This approach is the same as the selective regression testing approach described above with selection, execution, and coverage determination of existing tests omitted, eliminating any need for code instrumentation.

The recommended approach is a variation of the selective regression testing approach, with all test data provided by automated test data generation in situations where the user determines that conventional selective regression testing would have undesirable side-effects. The detailed steps of this approach are as follows:

1. The changes to the source code are determined by comparing the IRIS graphs generated for the original source code and for the modified source code. The identified changes will be recorded by the system, for traceability of the testing. Since this step involves working directly with IRIS, it must be coded in Ada.
2. The changes in the source code will need to be related back to the source code files. This can be done using the source code position relations that are created during generation of the IRIS graphs. The information generated by this step will also be recorded, for purposes of traceability. This step must also be coded in Ada, and may be combined with step 1 into one program.
3. The code elements that must be covered in the testing are determined. This step depends on the changes made to the source code, as well as the coverage criterion that is used. Several criteria may be available for selection, although it is likely that only one (such as statement coverage) would be used in the initial prototype. If this step is combined with the previous steps into one program, then it must be coded in Ada; otherwise, it could be coded in some other programming language, such as C++.
4. In the case where selective regression testing is possible, a copy of the modified OFP source code is instrumented to relate the tests to the code that they execute. This step does not need to be coded in Ada, and therefore could be coded in C++.
5. In the case where selective regression testing is possible, the copy of the modified OFP source code is also instrumented to track code elements that are covered by the tests. This step also could be coded in C++, and could be combined with step 4 into one program.
6. In the case where selective regression testing is possible, the tests that are to be rerun will be determined and presented to the user. The selected tests are

recorded by the system, for traceability of the testing that was performed. This does not need to be coded in Ada, and therefore could be coded in C++. The user will then run those tests. During execution of each test, the code executed by the test will be updated in the database of test information. Also, the code elements covered by each test will be recorded.

7. Any code elements that remain uncovered after the selective regression testing will be reported to the user. In the case where selective regression testing was not possible, these will be the code elements that were determined in step 2. This step does not need to be coded in Ada, and therefore could be coded in C++.
8. As an option to the user, an automated test data generation system will be used to create tests that cover the code elements reported in step 7. Any code elements that still could not be covered will be reported to the user. Both the generated tests, and the uncovered code elements, will be recorded by the system for traceability of the testing. This step does not need to be coded in Ada, and therefore could be coded in C++.
9. In the case where selective regression testing is possible, the generated tests are added to the test database. This step does not need to be coded in Ada, and therefore could be coded in C++.
10. The user runs the generated tests, plus any others created by the user to cover possible deficiencies in the generated tests. In the case where selective regression testing is possible, the code executed by each test will be added to the database of test information. Any code that may be developed for this step does not need to be coded in Ada, and therefore could be coded in C++.

The problem becomes how to automatically generate test data. Existing approaches are likely to not scale well for handling code of the size and complexity of OFPs. However, we could take a backward slice with respect to the code element that is to be covered. We expect that this would greatly reduce the amount of code that must be considered for the generation of test data. An existing method, such as a constraint-based test data generation method, could then be applied to the backward slice. Besides the code, formally stated limitations on input values, as well as relationships between input values, are likely to be necessary. The purpose of these formal statements would be to prevent the raising of exceptions or the entrance into an infinite loop.

With this problem solved, the problem becomes one of test minimization. In the case where selective regression testing is not possible, the process described above will need to be performed for every code element to be covered. Depending on the coverage criteria used, this could be a large number of tests. However, many of the tests will be redundant - tests that were previously performed covered the elements that other tests were designed to cover - there was just no way of detecting it. To alleviate this problem to some extent, a forward slice could be taken, and elements in either of the slices could

be considered covered by the generated test. This depends on the ability to obtain a precise backward slice - it must include all code that will be executed based on the element to be covered for the purpose of test generation, and it must not include code elements that are not covered since it is also used to determine additional covered code elements. The forward slice, however, only needs to include code elements that will be covered based on the code element to be covered by the generated test data. Approaches that will further reduce the redundancy of tests are an area for further research.

The approach to software change verification and validation that is recommended in this section will significantly improve the Air Force's ability to maintain OFPs in a timely manner while providing a high level of assurance. The requirements of the user are minimal - the main task that may fall upon the user is determining whether the results of the test are what is expected, and there may be a need for human interaction during the test execution. The specification about the inputs should not require a significant amount of effort. Although not specifically mentioned as a requirement, we believe that the time involved in the testing effort is a major consideration for any approach to the problem. This eliminated the consideration of approaches (such as formal verification) that, while providing even higher levels of assurance, would require extensive effort on the part of the maintainer and tester. A major requirement that we imposed is that the method should provide the best possible support for emergency modifications to OFPs. This rules out any approach that requires substantial specification or editing of test scripts. At the same time, the proposed method uses coverage criteria as a measure for the quality of the testing. Code elements that fall under the criteria but have not been adequately tested are identified, increasing the reliability of the released OFP. Gearing the testing to the modifications made to the code ensures that the OFP is adequately regression tested, while not performing tests that have no possibility of exposing an introduced fault. The efficiency in testing resulting from this selective testing also supports emergency modifications to OFPs. In summary, the recommended approach provides a high level of assurance by executing (and developing) tests that are necessary, while excluding (some) testing that is redundant as well as testing that has no possibility of exposing an introduced fault.

6. THE AUTOMATED TESTING PROTOTYPE

Task IV was to involve development of a software change verification and validation prototype system based upon the approach recommended as a result of the literature survey and evaluation detailed in Chapter 5. Funding for development of this prototype was reallocated to Task II, the prototype ImpACT support for JOVIAL programs, shortly after the task was begun. This chapter details the design and investigative work that was performed on Task IV.

The software change verification and validation prototype was being designed with the following goals:

- *Address the specific needs and requirements involved with avionics software* -- Design and develop the prototype to address the special needs of avionics software maintainers and testers, specifically in their requirements for high-assurance, minimization of effort, and support for real-time systems with limited resources.
- *Prove that the usefulness of the recommended approach* – Prove that the recommended approach did meet the special needs of avionics software maintainers and testers by providing prototype support of selected portions of the recommended approach.
- *Extend the state-of-the-art* – The portions of the prototype that would be chosen for implementation would focus on some of the language-independent aspects of the prototype, as well as portions that represented novel approaches to the problem.

We used the following approach in the concept and design of the prototype, or would have used the following approach had the effort progressed into implementation:

- We discussed concerns about approaches being considered with some maintainers of avionics software.
- We developed a top-level design of the prototype.
- Components would be designed in detail, with some investigative efforts, and implemented individually.
- The design would account for, and the implementation would utilize, third-party software where doing so would prove beneficial in the areas of prototype capability and schedule.

The following sections describe the top-level design, the detailed design that was being performed on one of the components of the prototype, and the investigative efforts that were associated with that component.

6.1 THE TOP-LEVEL DESIGN

A top-level design was developed for the prototype, based on the recommended approach that resulted from the literature search and evaluation. We began by thinking about the interfaces and data that were external to the system that would be required in implementing the system. From this, a context diagram was developed, which is shown in Figure 6-1. In the design, the user interacts with the system using a workstation, providing requested options and viewing the tests that the system indicates must be run. The source code for the original and modified versions of the program to be tested are used to determine source code lines that have been modified. The analysis database for the original and modified versions of the program to be tested are used to determine changes in dependences that must be covered in testing, particularly in the case where the coverage criteria involves data flows. Based on the coverage selected by the user, an appropriately instrumented version of the program to be tested will be produced (if using the conventional approach to selective regression testing). The user will need to build this instrumented version of the program being tested. In the case of conventional selective regression testing a test information database, mapping tests in the test suite and the code elements that they cover, is used in selecting regression tests for execution and is updated to reflect the new coverage of those tests as they are executed. Finally, a code element coverage database is maintained. This database keeps track of the code elements that must be covered in the regression testing, as well as the code elements that are covered by tests that have been

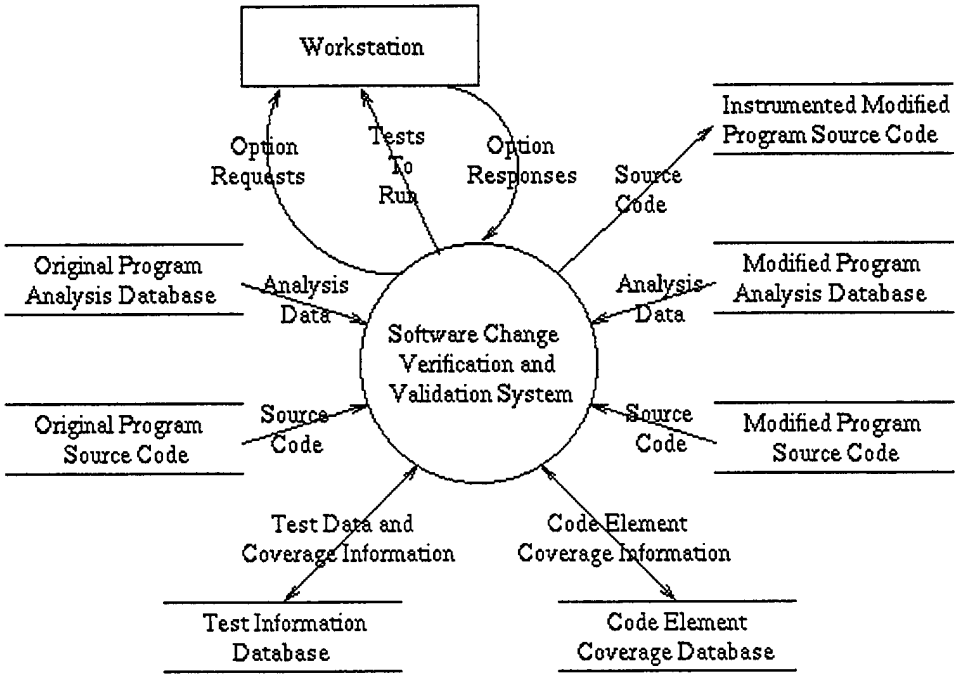

Figure 6-1 Context Diagram for Software Change Verification and Validation System

executed or that have been newly created.

After completing the context diagram, a Level 1 data flow diagrams for the system was developed. This data flow diagram, shown in Figure 6-2, contains five processes. These processes are:

- Generate Coverage Requirements
- Create Instrumented Program
- Select Existing Tests
- Generate New Tests
- Software Change V&V Control

The remainder of this section provides some of the details of these processes.

6.1.1 The Generate Coverage Requirements Process

This process is responsible for determining the code elements that must be covered in the regression testing. The names of the source code files for the program being tested are provided by the user through the Control process. The filenames are used to compare the original and modified versions of the source code being tested and determine the lines of code that were changed. Given this information, the analysis information databases for the two versions of the program are examined to select related analysis information, such as data flows, that must be covered in the regression testing. The type of coverage desired is also provided by the user through the Control process. The resulting elements that must be covered are stored in a database, as well as information about the source code lines that were modified.

6.1.2 The Create Instrumented Program Process

This process is used in cases in which conventional regression testing is possible, and is used to instrument the source code so that the coverage associated with the tests can be recorded. This process uses the modified source code and the code elements that must be covered, and creates the instrumented version of the modified program. Information about the source files, as well as the type of regression testing to be performed, is supplied by the user through the Control process.

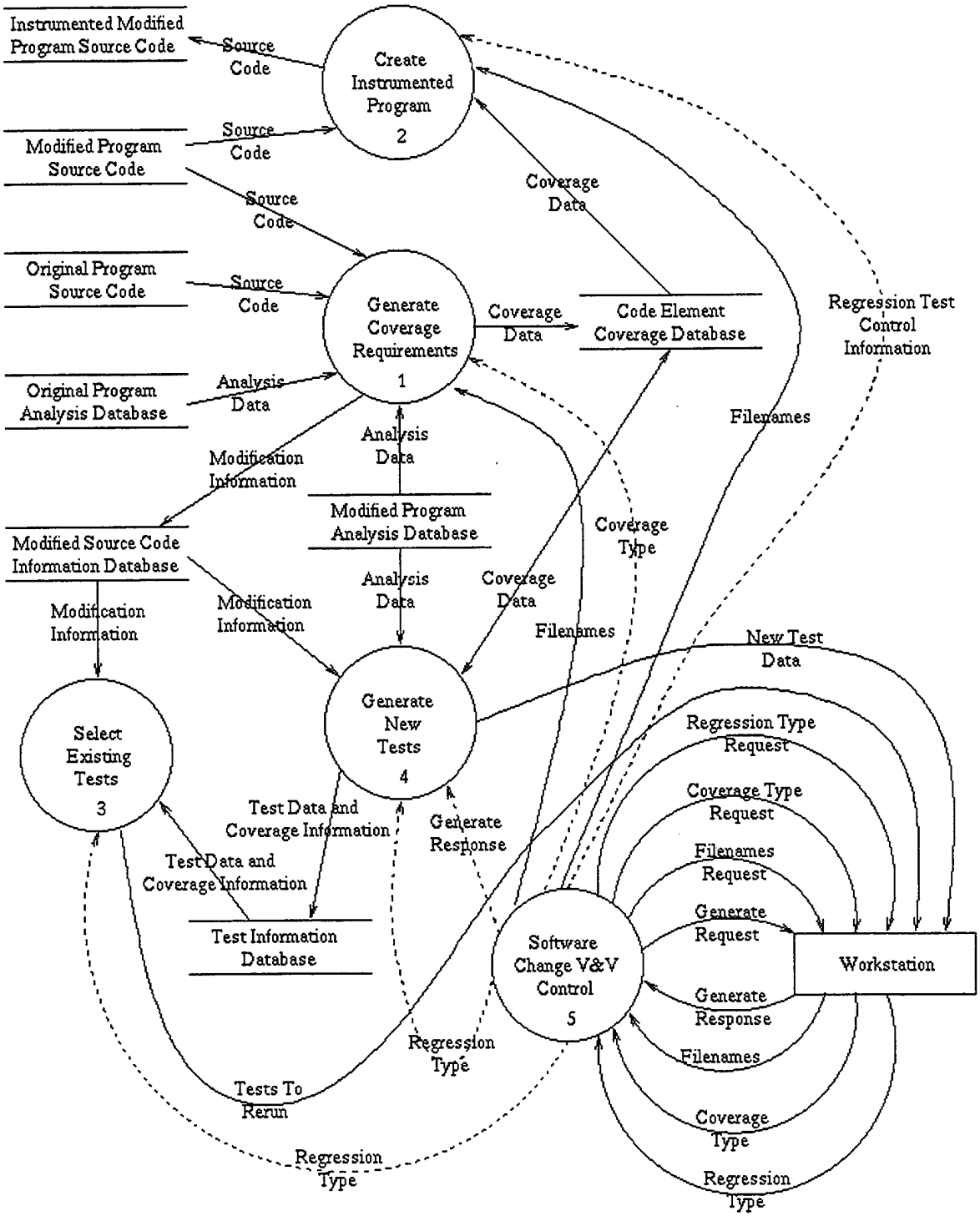

Figure 6-2 Level 1 Data Flow Diagram for the Software Change Verification and Validation Prototype

6.1.3 The Select Existing Tests Process

This process is used in cases in which conventional regression testing is possible, and is used to determine the existing tests in the test suite that must be rerun to attain (or approach as much as possible) the desired test coverage. This process uses the information about the modified source code and the database that indicates which code covered which elements, as well as the type of regression test to be performed, and presents the users with the tests that exist in the test suite that must be rerun. When these tests are rerun, the test information database is updated, and the code elements that are covered are used to update the code element coverage database.

6.1.4 The Generate New Tests Process

It is likely that the existing tests in the test suite do not provided the desired level of test coverage for the modified program. The code element coverage database, after existing tests have been run, indicates code elements that need to be covered but have not been covered. This process can be used to attempt to automatically generate tests that will result in the desired test coverage of the program being tested. The code element coverage database, the information about the modifications made to the program, and the analysis database for the program are all used in attempting to develop the new tests. In the case of conventional selective regression testing, these tests are added to the test information database. This process results in presenting the user with new test cases that must be run to attain (or approach more closely) the desired coverage. In cases where some elements remain uncovered by the testing, these elements are reported to the user so that he or she may either attempt to generate the required case through manual efforts or can determine that the case cannot possibly be created.

6.1.5 The Software Change V&V Control Process

The control process involves the interaction between the user and the software change verification and validation system. Information required from the user includes:

- Names of files in the program being tested
- The type of test coverage desired
- The type of regression testing that is to be performed (conventional, or the new, unconventional approach)
- Whether to attempt to generate additional, required test cases through automated means.

6.2 THE DETAILED DESIGN

Once the top-level design was completed to the point described above, the processes could be focused on individually. This permitted us to proceed with the detailed design.

The only component of the system that had any detailed design work performed on it was the part that would detect the changes that were made to the source code. We decided to determine the difference by determining the correspondence between the IRIS graphs for the original and modified source code files. This would allow changes not affecting the execution of the program, such as changes in white-space and addition of comments, to be easily disregarded.

The algorithm for determining the correspondence of the graphs was a recursive algorithm based on the one found (Ref. 44) in a short literature search for similar approaches. This algorithm uses a dynamic programming scheme to determine the maximum weight of correspondence by considering all possible correspondences of subtrees and continuing on with the correspondence that has the maximum weight. Although the dynamic programming algorithm is stated as working on a matrix that is the size of the number of subtrees in both the graphs, only two rows are ever used at any one point in the algorithm. This permits the storage space required in a recursive call to be linear as opposed to quadratic. In addition, the depth of the recursion is related to the height of the IRIS graph for a program statement. Therefore, as the size of the program grows, a significant increase in the recursion depth would not be expected.

To get the nodes that correspond as opposed to simply a weight, a list of pairs of corresponding nodes can be maintained. The nodes that did not correspond would be the nodes in the IRIS graph that do not also appear in the list. Using the text position information that is used for hyperlinking in ImpACT, these non-corresponding nodes could be related back to the source code, resulting in identification of modified source code lines. This information associated with the IRIS node could also be used to relate the non-corresponding node to dependence information, such as data flow dependences that would be affected by the modification.

6.3 INVESTIGATION AND IMPLEMENTATION

The only work done in implementing the process that determines the modified code and elements that need testing was some investigation of working with the two IRIS graphs. We quickly discovered a problem in experimenting with working with two IRIS graphs. The Pleiades repository manager, in which the IRIS graphs are stored, only allows one session to be active at a time. This would cause problems since we wanted to work with two repositories, one for the original code and one for the

modified code, at the same time. Doing this would require that multiple sessions be active at the same time. We asked the staff at The University of Massachusetts to confirm that what we wanted to do would be a problem, and to suggest ways to work around the problem. They suggested several solutions, but work on the prototype was halted before we received their response.

7. CONCLUSIONS AND RECOMMENDATIONS

ImpACT, the tool developed by the AAV&V-II program, will provide valuable assistance to software maintainers, and can be applied throughout much of the software maintenance process. It provides information needed to make informed decisions during planning, risk assessment, and testing, limits the amount of code that must be considered while designing and implementing modifications, and ensures the development of maintainable code. We believe that use of ImpACT, as described below, will result in:

- *More accurate plans.* Interprocedural analysis provides valuable information, such as the complexity of the code that is to be modified and critical functionality that may be impacted, that is directly related to the size of the maintenance effort.
- *Near elimination of rework.* Interprocedural analysis provides the maintainer with an improved understanding of the code being maintained and the interactions in the code. The result is better designs, implementations, and testing of the modifications, which will result in a significant reduction of out-of-phase rework. In addition, the focused testing that can be performed based on the interprocedural dependence analysis will nearly eliminate injection of new faults into the released software.
- *Significant reduction in design and implementation time.* The analysis information can be used by the maintainer to focus quickly on the code to be changed as well as the code dependent on that code. The less code the maintainer has to consider while designing and implementing the modification, the quicker the design and implementation can be completed.
- *Great reductions in the maintenance time growth rate caused by increasing complexity.* The dependence complexity metrics provided by ImpACT can be used to evaluate alternative approaches to a requested modification. Minimizing the complexity of the code is the key to having maintainable code. The more complex the code is, the more time it takes to understand the code related to a modification, to design and implement the modification, and to perform the required regression testing (because of an increase in the number of tests that need to be performed).
- *As much as a 75% (or more, depending on the number of modifications) decrease in testing time (or more testing, in the same or less time, focused on the code impacted by the modifications).* Interprocedural dependence analysis exposes code that may be impacted by a modification, and code that cannot be impacted by a modification. The only place that faults may be injected into the system is in

the modified code and the code that is impacted by that modification. Therefore, only this relatively small portion of the code needs to be retested, resulting either in a substantial decrease in testing time or the ability to perform more extensive testing of this small portion of the code.

Besides these results, ImpACT will also increase the reliability of the released, modified software.

The remainder of this chapter summarizes the activities of each task of the AAV&V-II program. It then presents the conclusions we have drawn from the AAV&V-II effort and our recommendations for future work. The conclusions, presented in Section 7.2, present a detailed discussion of the maintenance cycle benefits that result from adopting ImpACT.

7.1 SUMMARY OF TASKS OF THE AAV&V-II PROGRAM

This section summarizes the work undertaken and the results achieved during the AAV&V-II program.

7.1.1 A Production-quality AAV&V System for Ada

TASC developed a production-quality system for impact analysis that was based on the results of the original AAV&V program. Two technologies were at the center of this system - interprocedural dependence analysis and hyperlinked views of analysis information.

The interprocedural dependence analysis algorithms were initially relatively simple extensions of the intraprocedural dependence analysis algorithms provided by ProDAG. As larger programs were analyzed, it was necessary to improve the performance of several of these algorithms, the forward dominators and the reachability algorithms in particular. Although the forward dominators algorithm still resembles the original ProDAG algorithm, the reachability algorithm was completely reworked and now, we believe, yields a precise reachability analysis at an interprocedural level.

The hyperlinked views permit the user to select something within one view and observe the corresponding information in the other views. The CFG and ICFG views give the maintainer a graphical view of the control structure of subprograms and the entire program, respectively. The Dependence Metrics window allows the maintainer to view the effects that alternatives have on dependence complexity, permitting him or her to choose the alternative that least increases the complexity. This is important because as the program complexity increases, the time to make and test modifications to

the software increases. Whereas the Dependence Metrics window allows the user to manage the complexity, the combination of the Source Code Listing view, PDG view, and Statement Dependence Graph view permit the user to understand the complexity. Using these views, the maintainer can determine the code that may be impacted by a software modification. Depending on the phase of the maintenance effort, this allows the maintainer to determine the risk in a proposed modification, choose to implement a modification alternative that has the least risk or will require the least amount of testing, or understand the portions of the system that must be retested.

7.1.2 A Prototype AAV&V System for JOVIAL

TASC developed a prototype system for impact analysis, based on the system developed for Ada, for the JOVIAL programming language. The JOVIAL dialect that was chosen for prototype support was J73, which is defined in MIL-STD-1589C. TASC developed a language processing component that would generate an IRIS representation for a large subset of J73. Some minor modifications were also necessary to the CFG Generator to support some IRIS constructs that were added specifically for JOVIAL. The remainder of the system for Ada, however, was able to be reused for the JOVIAL system with no modifications. Besides providing prototype support for JOVIAL, this task also displayed the advantages of basing software tools on a language-independent representation.

7.1.3 Survey on the Science of Testing

TASC conducted a thorough survey of literature in the areas of automated support for software testing and automated formal verification. The literature was summarized and was evaluated against a set of evaluation criteria. An approach to automated support for the testing of implemented software changes was formulated. This approach involved test coverage criteria, selective regression testing, and automated test data generation, but was novel in its attempt to support the special needs of real-time, embedded software. Before the draft of the report on the survey, evaluation, and recommended approach could be completed, effort was directed away from this task in favor of additional effort on the prototype impact analysis support for JOVIAL. The information contained in the unreleased draft of that report is contained in Chapter 5 of this final technical report.

7.1.4 Prototype for Automated Testing Support

TASC began the design of a prototype for the approach recommended as a result of the survey on the Science of Testing. A context diagram and a first-level data flow diagram were created for the system. Investigation of detecting the changes to the software based on IRIS graphs was begun shortly before the effort was redirected to the

prototype impact analysis support for JOVIAL. The design artifacts and a description of the investigation that was performed are included in Chapter 6.

7.2 CONCLUSIONS

This report has documented the results of the Advanced Avionics Verification and Validation Phase II (AAV&V-II) program. The goal of the program was to advance the state-of-the-art in automated V&V techniques and tools for real-time, mission-critical software. One of the main results is a software tool that will permit software maintainers and testers to do their work more efficiently while providing a product with greater reliability. This tool, called ImpACT, provides information needed to make informed decisions during planning, risk assessment, and testing, limits the amount of code that must be considered while designing and implementing modifications, and ensures the development of maintainable code.

In planning, ImpACT's interprocedural analysis capabilities can be used to assist in determining the complexity of the maintenance to be performed. This complexity, as well as the criticality of the functionality impacted by the modifications, largely determines the time that will be needed to complete the maintenance cycle. The code to be modified can be examined for its dependence complexity. The higher the complexity is, the longer the modification can be expected to take. This is true because the more substantial the impact of the change on the system, the more care must be taken in designing and implementing the modification, and the more tests that must be rerun. In addition, if dependence analysis of the code to be modified reveals that critical sections of the system will be impacted, even more care will be needed in designing and implementing the modification, and the testing will likely need to be more thorough. Therefore, ImpACT can provide valuable, reliable information related the size of the maintenance effort quickly, as opposed to guessing, blindly taking some metrics average, or manually examining the code.

In risk assessment, ImpACT's interprocedural analysis capabilities can be used to expose buried dependences of the proposed modification. Because the dependence analysis can follow chains of dependences, it can expose dependences that may not be directly dependent on the modification but are indirectly related to the modification by following a chain of direct dependences. One of these distant, indirect dependences might be a critical portion of the system. Realizing that the modification would impact this critical functionality permits a tradeoff analysis – is the risk of introducing a fault in the critical functionality worth the benefit of making the modification, particularly in the case of a request for an enhancement? Without the information provided by ImpACT, this dependence on the critical functionality might not be realized, resulting in implementing enhancements that might have risky consequences.

In design and implementation, ImpACT can be used to identify and reduce the amount of code that must be considered in implementing a modification. The Control Flow Graph and Interprocedural Control Flow Graph can be used understand the logic flow of modules and the program, respectively, and help the maintainer to focus on the code that is to be modified. Without this information, the maintainer would need to look through (potentially outdated) documentation and the source code to try to locate the code to be modified. Once the code to be modified is located, the Program Dependence Graph can be used to find the code dependent on that code. This is important because it is the only code with which the software engineer must be concerned when designing and implementing the modification. Without this information, there is a high probability (especially in large, complex, embedded system) of rework, caused by the modification having an adverse effect on some other part of the system. This adverse effect likely would not be detected until integration or system testing and, without ImpACT, could go undetected and be included in the released system. The later the adverse effect is detected, the more it costs, in time and money, to fix it. Worse, if the injected defect were to make it into the release, the result could be serious financial loss, serious injury, or loss of life. Knowledge of what code is impacted and must be considered, besides avoiding fault injection, will reduce the code that a careful software engineer must understand before designing and implementing the change, thereby reducing the time that must be spent understanding the code.

The complexity metrics provided by ImpACT can be used by the software engineer to design and implement more maintainable code. As the dependence complexity of the code increases, the time required to perform a maintenance task increases. The dependence complexity directly affects the time needed to analyze the risks, to design and implement the modification, and, as we shall see, the time to test the system after the modification. Maintenance typically increases the complexity of the code because something is being added to the system that was not in the original design. The complexity metrics can be used to evaluate alternatives to a modification to determine the one that has the least negative effect on the dependence complexities. For example, if one alternative to a modification added 45 dependences and another alternative added only 30 dependences, implementing the alternative that added 30 dependences would, with respect to the one with 45 dependences, result in less testing, fewer dependences to consider in the next modification, and less impact if this same functionality needed to be modified in the future.

In testing, the interprocedural dependence capabilities show what code was affected by the change. This information is used by the tester to select tests that will need to be rerun, and is also used to determine if any new tests are needed and what they will need to test. Without the analysis information, the two choices are to either retest the entire system or guess at what needs to be retested. Neither approach provides information about the potential need for new tests. If all tests are run, then a large amount of the tests are testing functionality that was not impacted by the change,

and therefore cannot expose any injected faults. Understanding the code impacted, and that therefore must be retested, would permit the testing to be performed in a shorter amount of time, or permit more intense testing of the impacted code. As an example, if the impacted code is even one-tenth of the program, then one would expect that only approximately one-tenth of the tests would need to be rerun, meaning that it would be possible to have the same assurance about the system with one-tenth the time spent in testing (or additional testing could be performed on the impacted code to have greater assurance). If the tester guesses at the tests that must be rerun, it is likely that some test that should have been run, which tests impacted code, will not be rerun. As mentioned above, injected faults making it into the release can have grave consequences.

In addition to the benefits from ImpACT, the proposed approach to automated software testing could provide additional benefits in the area of software V&V. Although ImpACT presently exposes the code that must be retested, it does not present the tests that must be rerun and does not provide feedback on the sufficiency of the tests that are executed. The proposed approach to automated software testing would fill that gap.

The ImpACT software has been demonstrated to persons associated with avionics software maintenance for the B-1B and the AC-130. Both groups have expressed serious interest in the AAV&V technology. In fact, ImpACT has been installed for the B-1B avionics software maintainers, and they were provided with a training class on its use.

7.3 RECOMMENDATIONS

The AAV&V-II program has been successful in meeting its goals and has begun the process of transferring some its results into application by avionics software maintainers. However, more is necessary to have a successful technology transition, and additional technologies can be provided to assist avionics software maintainers and testers. In particular, TASC recommends:

- Improvements to the interprocedural dependence analysis algorithms
- Enhanced support for the display and manipulation of large graphs
- Complete support for MIL-STD-1589C JOVIAL
- Support for additional languages, such as C++, ATLAS, and VHDL
- Combined software and hardware analysis
- Development of the prototype for automated testing

The effort required to address each of these areas is described in the following sections.

7.3.1 Improvements to the Interprocedural Dependence Analysis Algorithms

During the development and demonstration of the ImpACT support for JOVIAL, it was discovered that some of the analysis algorithms do not perform well for large programs. At this time, two of the algorithms have been improved to perform more quickly and to be more precise. However, the data and control dependence algorithms run for a substantial time on a 1750 procedure line of code program before running out of memory. Therefore, TASC recommends continuing to improve both the time and resources required by these algorithms.

The time can be improved using the same approach that was used in improving the interprocedural forward dominators algorithm. There, timing probes were added to the code to isolate the portions of the code that were consuming the most time. Once those sections of code have been identified, the code can be examined for ways to improve its performance.

The resources can be improved by examining the logic of the program. We have noticed that the analysis algorithms tend to leak a substantial amount of memory. Many of the leaks can be located and removed by simply tracing the logic of the program, either by hand or through the use of a debugger.

If ImpACT is to be applied to sizable OFPs, or even significant subsystems of OFPs, the interprocedural dependence analysis algorithms will need to run much more quickly and will need to use resources much more responsibly.

7.3.2 Enhanced Support for the Display and Manipulation of Large Graphs

As mentioned previously, both the ICFG and PDG may be very large graphs. To enhance the ability to use the ICFG, TASC proposes the design and implementation of a reference window concept within ImpACT. This reference window shows the entire ICFG, and another window shows the detail of a small portion of the ICFG. The reference window has a box around the portion of the ICFG for which detail is displayed in the other window. To enhance the ability to use the PDG, TASC proposes the design and implementation of a hierarchical display of PDG information. At the top level, the PDG would consist of subgraphs of highly dependent nodes. As subgraphs are expanded, greater and greater amounts of detail can be displayed, but extraneous detail remains hidden in unexpanded subgraphs. These two different approaches are necessitated by the different characteristics of these two graphical views.

7.3.3 Complete Support for MIL-STD-1589C JOVIAL

To successfully transition ImpACT for JOVIAL, the entire language will need to be implemented. This effort includes handling multiple source code files. To complete

the support, the IRIS graphs that have been designed but not implemented would have actions added to the JOVIAL grammar to generate the IRIS representation. In addition, it may be necessary to make some additional minor modifications to the CFG Generator. To support multiple files, the approach taken for the analysis of Ada programs would be examined and a similar approach would be implemented.

Another related issue is the various dialects of JOVIAL that are in use. If a target user organization provided TASC with the LRM for the specific dialect they use, the existing grammar could be expanded to permit that syntax and any additional actions could be designed and implemented. This approach is possible since we assume that the source code has valid syntax, permitting the grammar rules to be relaxed and expanded.

7.3.4 Support for Additional Languages

Support for additional languages, such as C++, ATLAS, and VHDL, could be added by simply developing syntactic and semantic analyzers for each language. Some modification would also likely be necessary to the CFG Generator. C++ would be an important language since it is one of the most popular languages at this time, and could therefore also provide opportunities for productization by a software tool vendor. Providing support for ATLAS would permit application to a large amount of test program sets that must be maintained as the system under test is changed. Supporting VHDL would permit hardware designers to determine the impact from changes that they make. Providing this support, however, would also likely require work in the area of dependence analysis of multiprocess programs.

7.3.5 Combined Software and Hardware Analysis

In examining the B-1B CITS OFP, it became obvious that a significant number of OFPs are likely closely related to the hardware. For example, an OFP might set some bit in the hardware and read some other bit to get a result. Although there may be a dependence in the system between the two bit variables, this dependence may not be generated by analysis of the software alone. Therefore, TASC proposes investigating the possibility of expanding ImpACT to provide a system dependence analysis as opposed to only a software impact analysis. Adding support for VHDL, mentioned above, would likely help to move impact in this direction.

7.3.6 Development of the Prototype for Automated Testing

ImpACT shows the maintainer or tester what code may be impacted and therefore needs to be tested, but that is the limit of its support. Implementing the proposed approach to automated support for software change verification and validation will extend this support. This approach will automatically detect the

software changes and the impacted code, will determine what tests from an existing test suite must be rerun, and will indicate the level of test coverage that is achieved. If the level of testing is not acceptable, or if the program cannot be instrumented so that the conventional regression testing support is possible, the proposed approach will attempt to generate test data sets that will provide the required level of testing. Throughout the entire process, the proposed approach would record information about the testing required and the testing performed. Clearly, this approach would result in more efficient and reliable testing.

APPENDIX A
EVALUATION CRITERIA

The remainder of this appendix details the criteria that were used in evaluating the methods that were reviewed and evaluated in the study of methods for automated support of the verification and validation of software changes. A Microsoft Access database was defined using these criteria, the contents of which are supplied in appendix B. These criteria are also discussed in Section 5.3.

CATEGORY	SUB-CATEGORY	AREA	EVALUATION CRITERIA	METRIC
Technical Effectiveness	Real-Time Performance	Timing Constraints	Can the candidate method take into account timing constraints?	Yes/No/Partially
	Software Characteristics	Metric	Does the candidate method provide a metric for quantifying results?	Yes/No/Partially
		Compatibility with Specification	Can the candidate method determine if the software performs according to a given specification?	Yes/No/Partially
		Correct Algorithm Implementation	Can the candidate method determine if the algorithms are correctly implemented in the software?	Yes/No/Partially
		Functions with Infeasible Paths	Does the candidate method adequately address problems with infeasible paths?	Yes/No/Partially
	Requirements/ Features	Assumptions	What are the assumptions associated with the candidate method?	Descriptive
			How restrictive are the assumptions associated with the candidate method?	Slightly/Moderately/ Largely
		Oracle	Can the candidate method evaluate software performance without the need for an oracle?	Yes/No/Partially
			If an oracle is necessary for the candidate method, what is the level of complexity of the oracle?	Descriptive
		Restrictions on Software Requirements/ Specifications	What restrictions/conditions are placed upon the software requirements/ specifications to allow use of the candidate method?	Descriptive
		Modifications to Tested Software	What modifications to the software under test are required to implement the candidate method?	Descriptive
		Adverse Effects	Can the candidate method be used without requiring modifications that might result in residual adverse effects on the tested software?	Yes/No/Partially
			If modifications are required that might result in residual adverse effects, what are the modifications that are required and the effects?	Descriptive
		Always Yields Result	Will the candidate method always yield a result?	Yes/No
			If the candidate method does not always yield a result, under what conditions does it not yield a result?	Descriptive
		Retest Minimization	Does the candidate method attempt to reduce the amount of retesting/ reverification necessary?	Yes/No/Partially
		Code Minimization	Does the candidate method minimize the amount of code to which the method is being applied?	Yes/No/Partially
		Coverage	Does the candidate method guarantee a specified coverage level for retesting/ reverification?	Yes/No/Partially
		Level of Assurance	What level of assurance does the candidate method provide?	High/Moderate/Low

CATEGORY	SUB-CATEGORY	AREA	EVALUATION CRITERIA	METRIC
Technical Effectiveness (cont.)	Usability	Applicability	Is the candidate method applicable to the program as a whole?	Yes/No/Partially
			Is the candidate method applicable to selected (e.g., critical) portions of the software?	Yes/No/Partially
		Scalability	How does the implementation/ performance of the candidate method vary with the size of the program (or module) under investigation?	Descriptive
		Levels of Test	What are the levels of test provided by the candidate method (e.g., down to logic branch, statement, logic statement, etc.)?	Descriptive
		Enhancement/ Extension	Is the candidate method amenable to enhancement or extension?	Yes/No/Partially
			If the candidate method is amenable to enhancement or extension, to what extent?	Descriptive
		Usable with Other Techniques	Can the candidate method be combined with other techniques?	Yes/No/Partially
Automation Requirements/ Restrictions	Hardware/ Software	Implementation Hardware	What hardware is required to implement the candidate method?	Descriptive
		Test Hardware	What special-purpose test hardware (e.g., hardware-in-the-loop) is required to implement the candidate method?	Descriptive
		Software	What software or auxiliary software tool (e.g., semantic analyzer, theorem prover, etc.) is required to implement the candidate method?	Descriptive
	Test Data	User Input	What information is required from the user to implement the candidate method (e.g., software specification, timing requirements, etc.)?	Descriptive
			What is the format of the information required from the user for the candidate method?	Descriptive
		Test Generation	Can the candidate method assist in the development of test cases?	Yes/No/Partially
			If the candidate method can assist in the development of test cases, how?	Descriptive
		Probe Data	What data does the candidate method generate or require to probe the software/hardware under test (e.g., test cases, test inputs)?	Descriptive
		Output Data	What data does the candidate method require from the software/hardware under test (e.g., program responses to inputs)?	Descriptive
	Miscellaneous	Diagnostics	What diagnostics are provided by the candidate method?	Descriptive
		Effort	How much effort is required to automate the candidate method?	Small/Moderate/ Large

CATEGORY	SUB-CATEGORY	AREA	EVALUATION CRITERIA	METRIC
Effort Required to Perform V&V	Time/Skill	Human Interaction	How much manual effort* is required to perform an analysis with the candidate method?	Small/Moderate/ Large
		Skill Level	How much additional user training is required to support the candidate method?	Small/Moderate/ Large
		Computer-Based Analysis	How much computer-based analysis* is required to perform software V&V with the candidate method?	Small/Moderate/ Large
		Total Time	What is the total duration for performing an analysis* with the candidate technique?	Small/Moderate/ Large

*Effort normalized based on fixed amount of code.

APPENDIX B
SURVEY OF AUTOMATED TESTING TECHNIQUES

The remainder of this appendix contains the evaluations of the various methods that were reviewed and evaluated in the study of methods for automated support of the verification and validation of software changes. The information was entered into a Microsoft Access database that consisted of fields for the various evaluation criteria. This appendix was generated by defining a report form for the database, and printing that form. Note that each method consists of three pages of evaluation report.

APPENDIX C
PORTABILITY ISSUES

ImpACT is currently hosted on a Sun Solaris environment, but the majority of the software could be ported to another Unix system without modification. The source code is written in Ada, and the user interface uses OSF/Motif. The only other requirement that we are aware of is that the Arcadia software we leveraged now requires that the operating system be POSIX compliant.

APPENDIX D
DEVELOPMENT ENVIRONMENT

ImpACT was implemented on a Sun Solaris environment so that it would be available on a commonly-used platform. The version of Solaris that we developed under is 2.5.1. The source code was all written in Ada, and the compiler used on the project was the SunAda 3.0 compiler. The user interface was written using OSF/Motif, and Ada bindings for Motif were therefore necessary. These bindings were provided by SERC through their Ada/Motif 2.0 product, which works for Motif 1.2.x under X11R5. SERC provides bindings for a large number of combinations of hardware, operating systems, and Ada compilers.

The source code was stored in two areas. Source code developed or modified by TASC was under configuration management control through Clearcase. The pathname of the Clearcase VOB was `/vobs/aavv`. Arcadia source code that was not modified by TASC was not placed under Clearcase control, but instead was stored in the directory `/home/aavv/Arcadia`. If the source files were to be located in directories other than these, modifications would be necessary to the Makefiles.

REFERENCES

1. Binkley, David, "Semantics Guided Regression Test Cost Reduction," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, August 1997, pp. 498-516.

2. Chen, Jian, and Han, Jun, "A Review of EVES," Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland, Australia, Technical Report No. 93-5, May 1993, accessed via URL ftp://ftp.cs.uq.oz.au/pub/TECHREPORTS/SVRC/tr93-5.ps.Z.

3. Chen, Yih-Farn, Rosenblum, David S., and Vo, Kiem-Phong, "TestTube: A System for Selective Regression Testing," *Proceedings of the 16th International Conference on Software Engineering*, IEEE, pp. 211-220.

4. Clarke, Edmund M., Wing, Jeannette M., et al., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, December 1996, pp. 626-643.

5. Cook, Robert E., Jr, "Automated Verification of Generic Computer Software Components," Doctoral Thesis, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, January 1998.

6. Cook, Robert E., Jr., Mitchell, Kevin J., and Ung, Elizabeth C., "Advanced Avionics Verification and Validation Phase II (AAV&V-II) Program: ImpACT User's Manual (Draft)," TASC Technical Report TR-06664-2, 15 April 1998.

7. Cornet, Steve, "Software Test Coverage Analysis," Bullseye Testing Technology, accessed via URL http://www.bullseye.com/webCoverage.html.

8. Crow, Judy, Owre, Sam, Rushby, John, Shankar, Natarajan, and Srivas, Mandayam, "A Tutorial Introduction to PVS," Computer Science Laboratory, SRI International, Menlo Park, CA, updated June 1995, also presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, FL, April 1995, and also available via URL http://www.csl.sri.com/reports/postscript/wift-tutorial.ps.gz.

9. DeMillo, Richard A., and Offutt, A. Jefferson, "Experimental Results from an Automatic Test Case Generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, April 1993, pp. 109-127.

10. DeMillo, Richard A., and Offutt, A. Jefferson, "Constraint-Based Automatic Test Data Generation," accessed via URL http://www.isse.gmu.edu/faculty/ofut/rsrch/papers/cbt.ps (also in *IEEE Transactions on Software Engineering*, vol. 17, no. 9, September 1991, pp. 900-910).

11. "EHDM Specification and Verification System - Version 5.X, Supplement to the User's Guide and Language Manuals," Computer Science Laboratory, SRI International, Menlo Park, CA, August 25, 1991.

12. Formal Methods Specification and Verification Guidebook for Software and Computer Systems, NASA Jet Propulsion Laboratory (JPL), 1995, accessed via URL http://www.jpl.nasa.gov/quality/sftwr/fm2.htm#top.

13. Gallagher, Matthew J., and Narasimhan, V. Lakshmi, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, August 1997, pp. 473-484.

14. Gupta, Rajiv, Harrold, Mary Jean, and Soffa, Mary Lou, "Program Slicing-Based Regression Testing Techniques," *Journal of Software Testing, Verification, and Reliability*, Vol. 6, No. 2, June 1996, pp. 83-112.

15. Hooman, Jozef, "Extending Hoare Logic to Real-Time," accessed via URL http://www.win.tue.nl/cs/tt/hooman/EHLRT.ps (also in *Formal Aspects of Computing*, vol. 6, 1994, pp. 801-825).

16. Kapur, Deepak, Musser, David R., and Nie, Xumin, "An Overview of the Tecton Proof System," *Proceedings of a Workshop on Formal Methods in Databases and Software Engineering*, Concordia University, Montreal, May 15-16, 1992, also Technical Report 92-25, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1992.

17. Kapur, Deepak, and Musser, David R., "Tecton: A Framework for Specifying and Verifying Generic System Components," Technical Report 92-20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1992.

18. Korel, Bogdan, "Automated Test Data Generation for Programs with Procedures," *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, ACM, pp. 209-215.

19. Korel, Bogdan, Al-Yami, Ali M., "Assertion-Oriented Automated Test Data Generation," *Proceedings of the 18th International Conference on Software Engineering*, IEEE, 1996, pp. 71-80.

20. Korel, Bogdan, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, August 1990, pp. 870-879.

21. Kromodimoeljo, Sentot, Saaltink, Mark, Craigen, Dan, Pase, Bill, and Meisels, Irwin, "A Tutorial on EVES using s-Verdi," accessed via URL ftp://ora.on.ca/pub/doc/95-6018-60.ps.Z.

22. Loyall, Joseph P., and Mathisen, Susan A., "Advanced Avionics Verification and Validation: Final Technical Report," TASC Technical Report TR-6277-10, 24 February 1994.

23. Luckham, David, Sankar, Sriram, and Takahashi, Shuzo, "Two-Dimensional Pinpointing: Dealing with Formal Specifications," *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 74-84.

24. Luckham, David C., and von Henke, Friedrich W., "An Overview of Anna - A Specification Language for Ada," *IEEE Software*, Vol. 2, No. 3, March 1985, pp. 9-23.

25. Luqi and Goguen, Joseph A, "Formal Methods: Promises and Problems," *IEEE Software*, Vol. 14, No. 1, January/February 1997, pp. 73-85.

26. Marceau, Carla, "Penelope Reference Manual, Version 3-3," Lockheed Martin Tactical Defense Systems, STARS-AC-C001/001/00, Asset identifier ASSET_A_874, 02-SEP-94, accessed via URL http://www.asset.com/WSRD/ASSET/A/874//ASSET_A_874.tar.gz.

27. Michael, Christoph C., McGraw, Gary E., Schatz, Michael A., and Walton, Curtis C., "Genetic Algorithms for Dynamic Test Data Generation," Technical Report RSTR-003097-11, RST Corporation, Sterling, VA, Version 1.1, May 23, 1997, submitted to *Automated Software Engineering '97*, accessed via URL ftp://ftp.rstcorp.com/pub/techreports/atcgp.ps.

28. Musser, David R., and Wang, Changqing, "A Basis for Formal Specification and Verification of Generic Algorithms in C++," Technical Report 95-1, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, January, 1995.

29. Musser, David R., and Saini, Atul, *STL Tutorial and Reference Manual*, Addison-Wesley, Reading, MA, 1996.

30. Offutt, A. Jefferson, and Pan, Jie, "The Dynamic Domain Reduction Procedure for Test Data Generation," accessed via URL http://www.isse.gmu.edu/faculty/ofut/rsrch/papers/dd-gen.ps.

31. Offutt, A. Jefferson, "An Integrated Automatic Test Data Generation System," accessed via URL http://www.isse.gmu.edu/faculty/ofut/rsrch/abstract/josi.html (also in *Journal of Systems Integration*, vol. 1, no. 3, November 1991, pp. 391-409).

32. Owre, S., Shankar, N, and Rushby, J. M., "The PVS Specification Language (Beta Release)," Computer Science Laboratory, SRI International, Menlo Park, CA, April 12, 1993, also available via URL http://www.csl.sri.com/reports/postscript/pvs-language.ps.gz.

33. Poston, Robert, "A Guided Tour of Software Testing Tools," accessed via URL http://www.aonix.com/Products/Testing/10xpart3.html.

34. Prather, Ronald E., and Meyers, J. Paul, Jr., "The Path Prefix Software Testing Strategy," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 7, July 1987, pp. 761-766.

35. Roper, Marc, Maclean, Iain, Brooks, Andrew, Miller, James, and Wood, Murray, "Genetic Algorithms and the Automatic Generation of Test Data," Empirical

Foundations of Computer Science Report EFoCS-19-95, Research Report RR/95/195, Department of Computer Science, University of Strathclyde, Glascow, Scotland, accessed via URL http://ftp.cs.strath.ac.uk/Research/EFOCS/Research-Reports/EFoCS-19-95.ps.Z.

36. Rosenblum, David S., and Weyuker, Elaine J., "Prediction the Cost-Effectiveness of Regression Testing Strategies," SIGSOFT '96, ACM, pp. 118-126.

37. Rosenblum, David S., "A Practical Approach to Programming with Assertions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 1, January 1995, pp. 19-31.

38. Rothermel, Gregg, and Harrold, Mary Jean, "Selecting Tests and Identifying Test Coverage Requirements for Modified Software," *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, ACM, pp. 169-184.

39. Rothermel, Gregg, and Harrold, Mary Jean, "A Framework for Evaluating the Regression Test Selection Techniques," *Proceedings of the 16th International Conference on Software Engineering*, IEEE, pp. 201-210.

40. Rushby, John, von Henke, Friedrich, and Owre, Sam, "An Introduction to Formal Specification and Verification Using EHDM," Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991, also available via URL http://www.csl.sri.com/reports/postscript/csl-91-2.ps.gz.

41. Shankar, N., Owre, S., and Rushby, J. M., "The PVS Proof Checker: A Reference Manual (Beta Release)," Computer Science Library, SRI International, Menlo Park, CA, March 31, 1993, also available via URL http://www.csl.sri.com/reports/postscript/pvs-prover.ps.gz.

42. Wang, Changqing, "Integrating Tools and Methods for Rigorous Analysis of C++ Generic Library Components," Doctoral Thesis, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, July, 1996.

43. Wang, Changqing, and Musser, David R., "Dynamic Verification of C++ Generic Algorithms," IEEE Transactions in Software Engineering, Vol. 23, No. 5, May 1997, pp. 314-323.

44. Yang, Wuu, "Identifying Semantic Differences Between Two Programs," Software -- Practice and Experience, Vol. 21, No. 7, July 1991, pp. 739-755.