NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

TECHNOLOGY TRANSFER OF THE COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)

by

Robert P. Cooke, Jr.

September, 1996

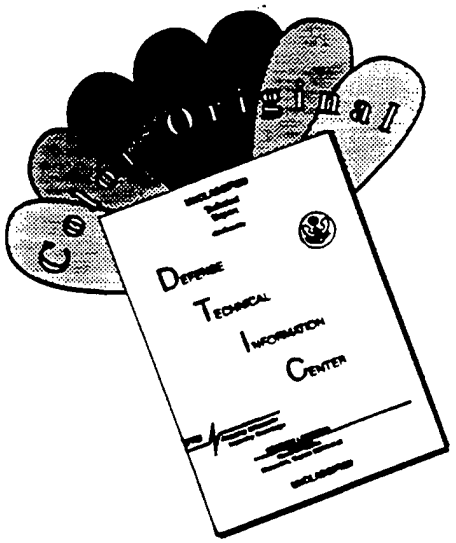
Thesis Advisor: Luqi

Approved for public release; distribution is unlimited.

19970129 057

DTIC QUALITY INSPECTED 4

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

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

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September, 1996.	Master's Thesis

4. TITLE AND SUBTITLE TECHNOLOGY TRANSFER OF THE COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)	5. FUNDING NUMBERS
6. AUTHOR(S) Robert P. Cooke, Jr.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

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

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

13. ABSTRACT *(maximum 200 words)*

The inability of the Department of Defense (DoD) to accurately and completely specify requirements for hard real-time software systems has resulted in poor productivity, schedule overruns, and software that is unmaintainable and unreliable. The Computer-Aided Prototyping System (CAPS) provides a capability to quickly develop functional prototypes to verify feasibility of system requirements early in the software development process. It was built to help program managers and software engineers rapidly construct software prototypes of proposed software systems. CAPS was developed by the Software Engineering Group at the Naval Postgraduate School (NPS) in Monterey, California.

This thesis investigates the transfer of technology of CAPS from NPS to DoD and the commercial industry. The effective transfer of technology requires user awareness of the technology and the ability to utilize the technology. Thus, a strategy is prepared for implementing the technology transfer of CAPS at NPS. To aid in this implementation, the quality and effectiveness of existing CAPS technical documentation is evaluated and recommendations for enhancement provided. Information dissemination materials are developed as part of this thesis which include three levels of CAPS briefings to potential sponsors, a home page, and a CD-ROM multimedia presentation. The implementation of this strategy will not only maximize the transfer of technology to the users, but also provide the optimum use of DoD software engineering resources available.

14. SUBJECT TERMS CAPS,technology transfer, prototyping, computer-aided prototyping, DoD software engineering	15. NUMBER OF PAGES 332
	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	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

TECHNOLOGY TRANSFER OF THE COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)

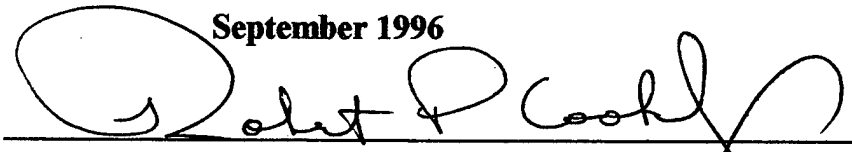
Robert P. Cooke, Jr.
Lieutenant, United States Navy
B.S., Old Dominion University, 1987

Submitted in partial fulfillment
of the requirements for the degree of

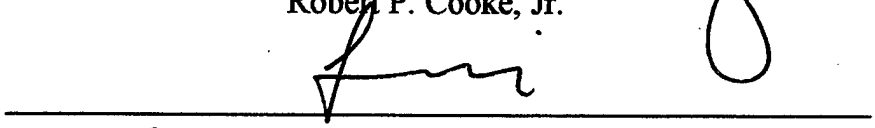
MASTER OF SCIENCE IN INFORMATION TECHNOLOGY MANAGEMENT

from the

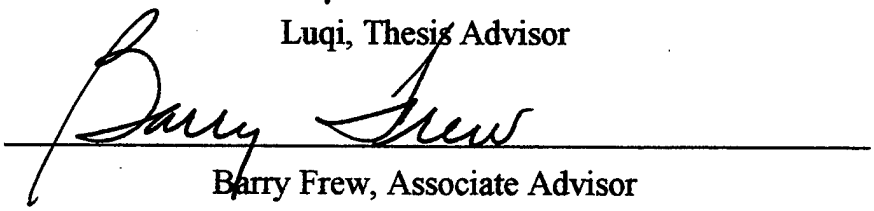
NAVAL POSTGRADUATE SCHOOL
September 1996

Author: 

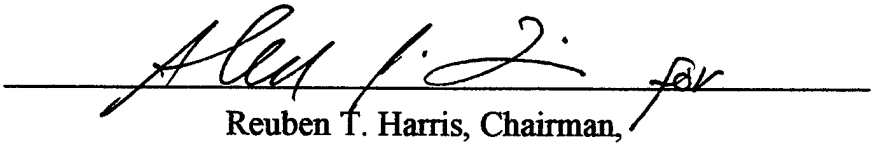
Robert P. Cooke, Jr.

Approved by: 

Luqi, Thesis Advisor



Barry Frew, Associate Advisor



Reuben T. Harris, Chairman,
Department of Systems Management

ABSTRACT

The inability of the Department of Defense (DoD) to accurately and completely specify requirements for hard real-time software systems has resulted in poor productivity, schedule overruns, and software that is unmaintainable and unreliable. The Computer-Aided Prototyping System (CAPS) provides a capability to quickly develop functional prototypes to verify feasibility of system requirements early in the software development process. It was built to help program managers and software engineers rapidly construct software prototypes of proposed software systems. CAPS was developed by the Software Engineering Group at the Naval Postgraduate School (NPS) in Monterey, California.

This thesis investigates the transfer of technology of CAPS from NPS to DoD and the commercial industry. The effective transfer of technology requires user awareness of the technology and the ability to utilize the technology. Thus, a strategy is prepared for implementing the technology transfer of CAPS at NPS. To aid in this implementation, the quality and effectiveness of existing CAPS technical documentation is evaluated and recommendations for enhancement provided. Information dissemination materials are developed as part of this thesis which include three levels of CAPS briefings to potential sponsors, a home page, and a CD-ROM multimedia presentation. The implementation of this strategy will not only maximize the transfer of technology to the users, but also provide the optimum use of DoD software engineering resources available.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 2

B. THE PROBLEM STATEMENT 3

C. OBJECTIVES OF THIS RESEARCH 4

II. COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS) 7

A. THE DOD SOFTWARE ENGINEERING CHALLENGE 7

B. TRADITIONAL DESIGN METHODOLOGIES AND TOOLS .. 10

C. WHAT IS CAPS? 11

1. CAPS Baseline Architecture 13

2. Program System Description Language (PSDL) 15

3. CAPS User Interface 17

4. Software Database 18

5. Execution Support System 18

D. HOW CAPS RESOLVES THE SOFTWARE ENGINEERING CHALLENGE 19

1. CAPS and the Software Reuse Initiative 20

a. SRI Definition and Initiatives 20

b. SRI Strategy 21

2. CAPS and Rapid Prototyping 22

E. ANALYSIS OF BENEFITS OF CAPS TECHNOLOGY 23

1. Reducing Uncertainty and Risk 23

2. Minimizing Time and Cost of Development Cycle 24

3. Increasing Acceptance of the New System 25

4. Avoidance of Opportunity Costs 26

5. Better Project Management 26
6. Bridging the Communications Gap Between Developers and Customers 27
7. Other Benefits 27
F. SUCCESSFUL SOFTWARE PROTOTYPING USING CAPS . . 27
1. C^3I Station 29
2. ATACMS 29

III. THE PROPOSED METHODOLOGY 30
A. DUAL-USE TECHNOLOGY AND TECHNOLOGY TRANSFER 30
1. Dual-Use Technology 30
2. Defining Technology Transfer 31
3. Adopting New Technology 33
B. LAUNCH POINTS FOR THE TECHNOLOGY TRANSFER PROCESS 34
1. Office of Technology Transition (OTT) 35
2. Office of Naval Research (ONR) 36
3. Navy Offices of Research and Technology Applications (ORTA) 37
4. Technology Transfer Process at NPS 37
C. THE RECOMMENDED TECHNIQUE 38
1. Phase One: Technology Innovation 38
2. Phase Two: Technology Confirmation 39
3. Phase Three: Targeting Technology Consumers 39
4. Phase Four: Technology Marketing 40
5. Phase Five: Technology Application 40
6. Phase Six: Technology Evaluation 41

IV. PREPARATION AND EXECUTION FOR TECHNOLOGY TRANSFER 43
A. INTRODUCTION 43
B. CAPS TECHNOLOGY TRANSFER PREPARATION PLAN . . 44
1. Conveying the Technology 44
2. Funding and Manpower Resources 45
3. Collaboration with NPS ORTA 47
4. Identifying Potential Users 48
C. CAPS BRIEFING PACKAGES 49
1. Executive Summary Brief 50
2. Technical Development Brief 50
3. Program Managers Brief 51
D. DOCUMENTATION 51
1. Product Knowledge 52
2. Knowledge of Users 52
3. Documentation Quality 52
4. CAPS Documentation 53
a. CAPS Tutorial 53
b. CAPS Users Manual 54
c. CAPS Installation Guide 54
d. CAPS Quick-Start Guide 54
E. INTERNET ACCESS TO CAPS INFORMATION 55
F. CAPS MARKETING MATERIALS 56
1. CD-ROM Multimedia Presentation 56
2. Information Brochures 57

V. CONCLUSION AND RECOMMENDATIONS 59

VI. THE FUTURE DIRECTION 63

APPENDIX A: CAPS TECHNOLOGY TRANSFER PLAN 65

APPENDIX B : EXECUTIVE SUMMARY BRIEF 67

APPENDIX C: TECHNICAL/DEVELOPER BRIEF 95

APPENDIX D: PROGRAM MANAGERS BRIEF 131

APPENDIX E: CAPS TUTORIAL 155

APPENDIX F: CAPS USERS MANUAL 257

APPENDIX G: CAPS INSTALLATION GUIDE 275

APPENDIX H: CAPS QUICK-START GUIDE 279

APPENDIX I: CAPS HOME PAGE 295

APPENDIX J: CAPS CD-ROM MULTIMEDIA DESIGN PLAN 297

APPENDIX K: CAPS INFORMATION BROCHURES 313

REFERENCE LIST 319

INITIAL DISTRIBUTION LIST 321

I. INTRODUCTION

From downsizing to defense acquisition reform, the face of the Department of Defense (DoD) is changing to a leaner, more cost efficient organization. DoD has embarked down various paths of organizational restructuring and policy reform to effectively and decisively meet the challenges of the future. During these changing times, DoD fights dwindling resources yet strives to maintain its military and technological superiority by keeping pace with ever emerging technology.

Technological superiority is based on scientific knowledge. DoD invests broadly in defense-relevant scientific fields. The objectives are first, to discover new knowledge, and second, to sustain a community of expert scientists who exploit new knowledge as they seek superior, new warfighting capabilities. By its very nature, basic research potentially applies to both military and non-military needs.

In the "Perry Memo" [Ref. 1], distributed June 1995, William J. Perry, Secretary of Defense, paved the way for Domestic Technology Transfer and Dual Use Technology Development (DTT/DUTD) within DoD for the 1990's and into the 21st century. DTT and DUTD are integral elements of the Department's pursuit of its national security mission. The memo states that "they must have a priority role in all DoD acquisition programs and must be recognized as key activities of the DoD laboratories...". The "Perry Memo" defined oversight authority and procedures for DTT/DUTD execution that allowed each Service to strengthen the technology transfer process by establishing a program that fosters dual use technology development, ensures exploitation of commercial technology, and nurtures technology transfer between in-house laboratories, industry, universities and not-for-profit

laboratories. Additionally, the increased sharing of facilities by the Service laboratories and industry; and participation in regional, state, and local alliances were encouraged. The impact of the "Perry Memo" changed the culture of service laboratories and agencies as evidenced by the increase in laboratory collaborations between DoD and industry to develop new technology that solves mutual problems.

It is imperative that DoD foster, to the maximum extent practical, an integration of the military, commercial industry, and academic resources, in order to achieve a more cost-effective, a single set of industrial enterprises capable of developing and building more affordable and productive military and commercial products. The defense investment in technology can be made to contribute to this integration by preferentially developing technologies that have dual use. More importantly, DoD must continue to be proactive in searching out technology and technology applications that reduce the cost of operating, maintaining, and upgrading systems that support the warfighter.

A. BACKGROUND

The inability of DoD to accurately and completely specify requirements for hard real-time software systems has resulted in poor productivity, schedule overruns, and software that is unmaintainable and unreliable. The Computer-Aided Prototyping System (CAPS) provides a capability to quickly develop functional prototypes to verify feasibility of system requirements early in the software development process. CAPS supports a revolutionary development process that spans the complete life-cycle of real-time software. CAPS is designed to incorporate the advantages of both prototyping and software reuse. CAPS exhibits great potential to enhance system development, acquisition, reduce costs, and facilitate life-cycle management of software intensive systems within DoD.

The Computer-Aided Prototyping System is a software application developed by the Naval Postgraduate School (NPS), Computer Science Department software engineering group. The development process has spanned a seven year period and Release (1)- CAPS 93 is currently being tested by various DoD agencies in developing hard real-time, software intensive systems.

Technology transfer is crucial for the success of CAPS research and development. Access to documentation and other pertinent data of the CAPS research is a key ingredient involved in effecting successful technology transfer to DoD agencies, commercial industry, and other interested technologists. A consolidated set of references and links to similar technology, properly prepared technical materials, and well constructed briefings and presentations will encourage the use of CAPS throughout DoD and the commercial industry by making technical and administrative data easily available.

B. THE PROBLEM STATEMENT

The poor past performance of DoD in acquiring and developing software intensive systems on time and under costs is well acknowledged. The DoD software engineering environment faces many challenges to resolve and reverse this trend. This thesis analyzes current CAPS technology answering the primary research question: **How can the current research and development of CAPS be collectively organized to provide effective technology transfer to similar projects for DoD and the commercial environment?**

In answering the primary research question, other preliminary questions must be considered. These questions are:

1. What software engineering problems can CAPS solve for DoD?

2. What is technology transfer and how does CAPS begin this process?
3. What funding and manpower resources are required to effect CAPS technology transfer?
4. Who are the points of contact for the research and development of CAPS?
5. How can interested technologists and users access CAPS technology?

C. OBJECTIVES OF THIS RESEARCH

The objectives for research of this thesis are focused in three areas. First, an analysis is conducted on CAPS technology for its potential use in DoD and commercial industry. Then the process of transferring new technology to DoD agencies, and between DoD and the commercial sector is researched. The results of this research are used to provide the CAPS research and development team with a plan and recommendations by which to effect CAPS technology transfer.

Deliverables consist of the write-up of this thesis which includes an evaluation of CAPS benefits, a CAPS technology transfer plan, and supporting materials required for successful technology transfer implementation. It is the objective of this supporting material to provide varying levels of information that include three CAPS briefing packages - an executive summary brief, a technical/developer brief, and a program manager brief. Existing technical documentation will be evaluated for quality, content, and effectiveness. Recommendations for revisions to technical documentation will be incorporated and presented in draft form as appendices to this thesis. Technical documentation consists of a tutorial, a quick-start guide, a users manual, and an installation guide. Deliverables which support the marketing of CAPS include an Internet home page, a multimedia presentation development plan, and information brochures.

The purpose of producing these deliverables is to promote a better understanding of the technology itself by potential users and to evaluate the relevance of CAPS to their field of research. By understanding the technology, potential users are able to assess potential benefits of CAPS and determine the feasibility of integrating CAPS into their current software development process. For current users, these deliverables will allow the optimization of time and productivity when utilizing the CAPS application. This thesis assumes the reader has a reasonable understanding of software engineering.

II. COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)

The focus of this chapter provides an overview of the Computer-Aided Prototyping System (CAPS) and its development by the Computer Science Department, Software Engineering Group at the Naval Postgraduate School, Monterey, California. This chapter highlights software engineering challenges faced by DoD. Relevant discussions on the relationship between CAPS, rapid prototyping, and software reuse are also provided. Section (E) gives an analysis of CAPS benefits to be realized by DoD and the commercial industry.

A. THE DOD SOFTWARE ENGINEERING CHALLENGE

The stakes for software development, a growing share of total technology investment, are high. Developing and maintaining software in DoD is very costly. Software, properly managed, provides key strategic and competitive advantages. Yet, while software is critical to the success of virtually all DoD systems, it has also proven to be a major cost driver.

Despite its shrinking budget, DoD has increased its investment in software from $30 billion in FY 1990 to $42 billion in FY 1995 [Ref. 2]. Improvements in software development lag behind the technical advances found in other segments of the computer industry. A 1994 Standish Group survey [Ref. 3] revealed a 31% cancellation rate for software development projects, with an average time overrun of 222% for completed projects. Only 16% of all software projects in the same study were completed on time and within budget. Even after delivery, the average software lifetime maintenance costs exceed 200% of initial development cost [Ref. 4]. Figure 1 illustrates the Standish Group survey results.

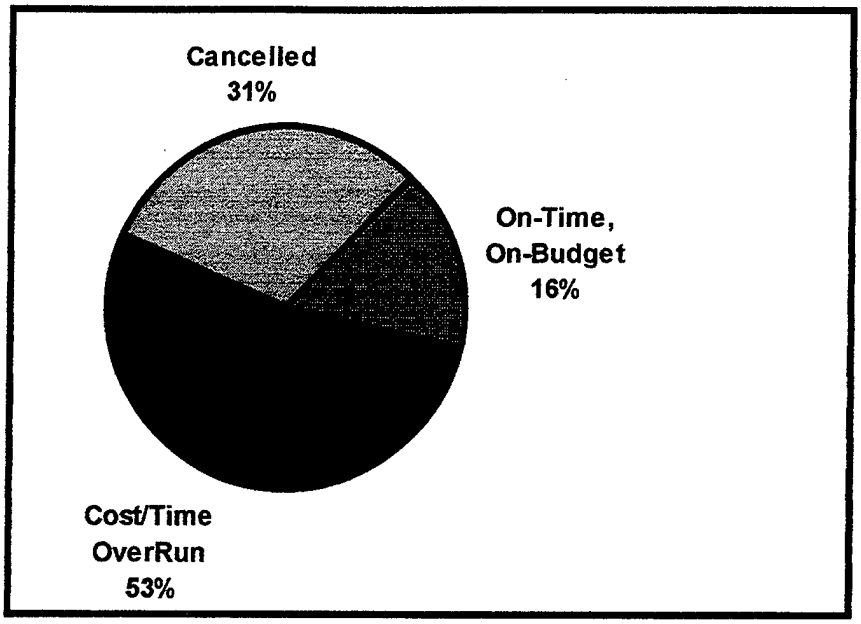


Figure 1. 1994 Standish Group survey - DoD software projects

A 1979 General Accounting Office (GAO) report [Ref. 5] concluded that 60 percent of DoD software contracts had schedule overruns, 50 percent had cost overruns, 45 percent of the software contracted for was unusable, 29 percent was never delivered, 19 percent had to be reworked to become usable, and only 2 percent was usable exactly as delivered.

Incredibly, after 15 years and 20 similar studies, the conclusions remain the same. Moreover, this dismal state of the art for software development holds true across the board, from embedded weapon control software to software for office automation. This is reported by Lloyd K. Mosemann, II, deputy assistant secretary of the Air Force, to a software technology conference in April 1994. [Ref. 5]

Even after delivery, the average software lifetime maintenance costs exceed 200% of initial development cost. Of the $42 billion in DoD software expenditures in the 1995 budget,

$28 billion was devoted exclusively to maintaining and fixing proprietary DoD software as illustrated in Figure 2. [Ref. 6]

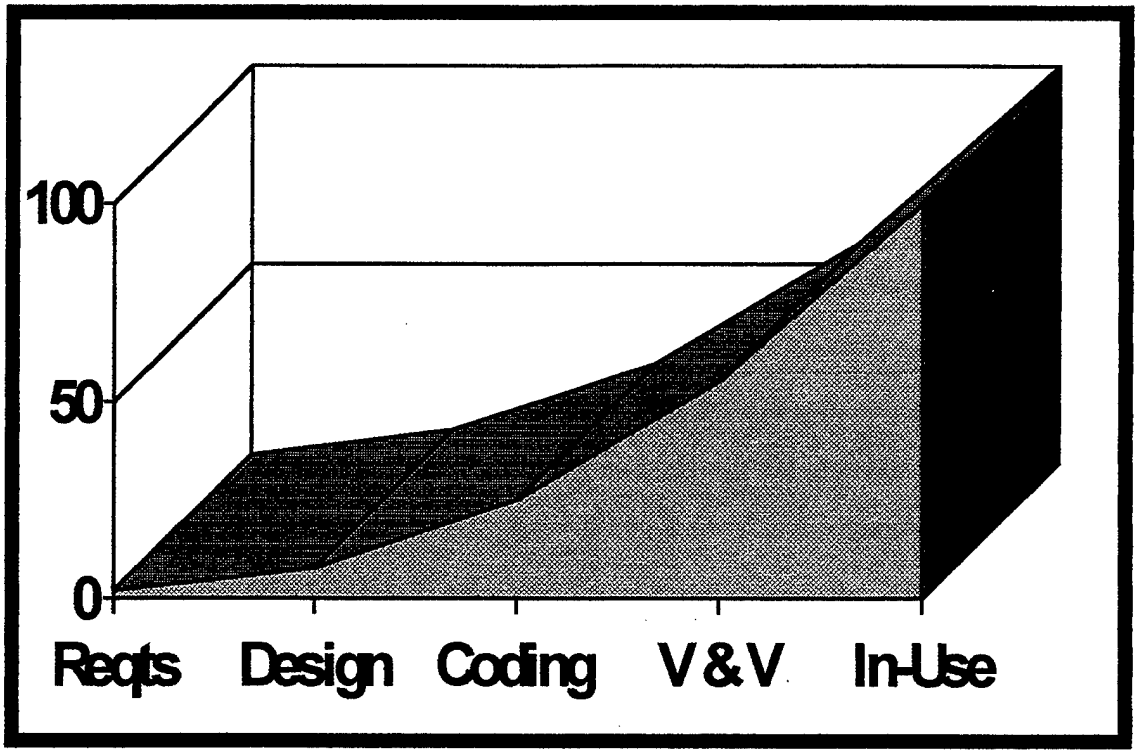


Figure 2. Cost of Fixing Software Errors

This software bottleneck drains resources and clearly impedes effective and efficient use of technology. A small improvement in software development and maintenance efficacy would yield millions, perhaps billions of dollars in savings. A major source of problems causing the software bottleneck is the inherently complex nature of large software projects and their component interfaces. The requirements for these complex systems, especially the interface requirements, are typically specified incorrectly, causing errors in design and implementation. Since the requirement specifications are not often well understood in the early phases of the development process, these errors go undetected or uncorrected until late

in the life cycle. Corrections of requirement specifications at this late phase become very expensive and time-consuming.

In DoD, this large-scale integration of complex software systems is the primary software engineering problem. The problem is further complicated by a continual pressure for functional improvement (i.e., changing requirements), creating a need for continually updated software, which in turn requires expensive specification changes and software maintenance. Traditional software development tools and methodologies have failed to do the job.

B. TRADITIONAL DESIGN METHODOLOGIES AND TOOLS

The "classical model" of system development and acquisition used in earlier years consisted of a sequence of activities which included understanding the prevailing state of the art, analyzing the market need, specifying the requirements, evaluating design options, and implementing, then supporting, the favored one. The traditional, standard military IT systems development and acquisition process involves five basic phases :

- System Requirements Definition
- System Design
- System Coding
- System Implementation/Testing
- System Maintenance

Each phase is thoroughly worked almost to completion, documented, and handed to the team for work in the next phase. Feedback flows from one phase to the next, but can, at times, be in the form of ambiguous requirements or even forgotten altogether. This

traditional approach, or "Waterfall" life cycle lacks any guarantee that the resulting product will meet the customer's needs. In most cases the blame falls on the requirements phase of the life cycle. This process assumes that a small amount of initial investment is sufficient to write a specification that satisfies all of the system requirements. It also assumes that once a linkage between the system design and the requirement specification has been established, the requirement will remain stable and unaltered during implementation.

Traditional software development methodologies attempt to inject more disciplines into the development process, ensure higher reliability and fewer errors, and provide for more efficient use of resources. Computer Aided Software Engineering (CASE) tools and prototyping methods have been developed to augment design methodologies. The intention is to avoid the disasters of cost and schedule overruns and project cancellations. Yet, traditional methods allow user evaluation of systems near the end of the process, when it may be impossible to correct some errors. Thus, traditional design techniques remain expensive, time-consuming processes that are prone to considerable errors. [Ref. 7]

C. WHAT IS CAPS?

Led by Professor Luqi, 1990 recipient of the Presidential Young Investigator Award, CAPS was developed by a small team of faculty and students at the Naval Postgraduate School (NPS) in Monterey, California. CAPS research is sponsored by the National Science Foundation, Ada Joint Program Office, Defense Information Systems Agency, Army Research Office, Army Research Laboratory, and NPS itself. Most CAPS research is conducted by students and faculty in the software engineering track, one of six specialization tracks offered in the computer science curriculum at NPS. Research projects range from proof-of-concept CAPS component implementations to working prototypes of real-world software systems.

The CAPS program has been very productive for the computer science department and NPS. Five of the past thirteen Admiral Grace Murray Hopper awards recognizing outstanding thesis research have been awarded to CAPS students. Several CAPS researchers have graduated with distinction. In addition, the CAPS program has spawned a wide variety of technical papers and conference presentations. A 1994 Journal of Systems and Software study of published work relevant to the discipline cited NPS as the leading academic institution in the field of systems and software engineering [Ref. 8].

CAPS or "Computer-Aided Prototyping System", is an integrated set of software prototyping tools that automate significant portions of the software development process, improving software quality while reducing development time and cost. CAPS was developed for the software engineering professional for the purpose of designing prototypes of hard real-time systems. It is a tool which enables software engineers to build efficient real-time system prototypes using real-time design concepts. CAPS can be used throughout a product's life cycle in assisting the engineer through software development phases such as feasibility studies, requirements analysis, design, code and acquisition.

CAPS is based on a Prototype System Description Language (PSDL), a hybrid graphical and textual language used to describe prototypes of real-time systems. CAPS can translate PSDL code into Ada code allowing it to be compiled and executed to determine whether or not the design requirements of a prototype are being met. CAPS can be used for rapid prototyping because it enables software to be designed quickly as augmented data flow diagrams and provides decision support and code generation capabilities. The editing tools convert the graphic objects into PSDL programs which can then be translated into Ada code for execution. Figure 3 illustrates the CAPS development environment.

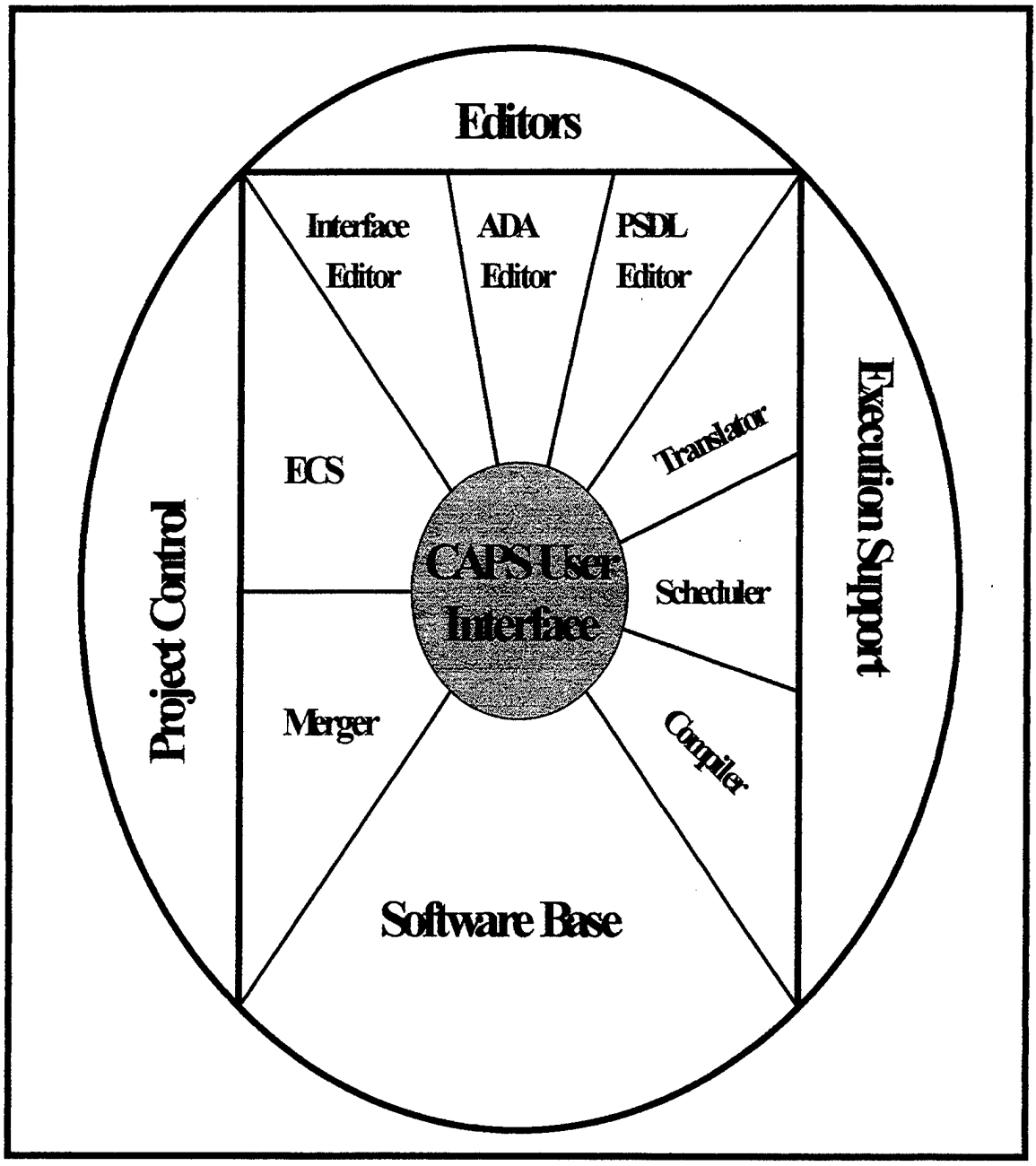


Figure 3. CAPS Development Environment

1. CAPS Baseline Architecture

CAPS (Release 1) baseline architecture consisted of editors and execution support tools. In addition to the PSDL Editor which provides some automated design completion and

consistency checking capabilities, Release 1 included an Ada Editor, Interface Editor, Translator, Scheduler (for time feasibility checks of various tasks) and a Compiler. These individual components, or prototype building tools, are linked together by a user-interface. Together, these tools make up the CAPS integrated development environment.

The newest release of the CAPS tool provides additional capabilities such as an Evolution Control System (ECS), which gives automated assistance to software managers for project planning and scheduling, designer task assignments, and estimation of project completion dates. It supports distributed prototype development. This is important especially on a large software project where there may be multiple development teams. ECS will track all proposed, ongoing and completed changes to a design and provide automated assistance to project management in controlling and coordinating concurrent development efforts.

The current version of CAPS also provides automated assistance for locating and adapting reusable components (coded in Ada or other programming languages) in a software base, thereby supporting the concept of software reuse with a high degree of automation. CAPS (Release 1) runs under the following hardware/software requirements:

- Platform: Sun
- Workstation: SPARC station
- Disk Space: 15MB[1] (See Note)
- Operating System: SunOS 4.1.1 or later
- Window Environment: X Window System X11R4 or X11R5

[1] NOTE: The 15 MB disk space requirement above does not include the space needed to install OSF/Motif, Sun Ada, or prototypes generated by CAPS. In order to avoid any paging and inaccurate timing results it is recommended that a minimum of 16 MB of main memory be available to run CAPS.

- OSF/Motif: 1.1.2 or later (with SunOS)
- Compiler: Sun SPARCompiler 1.1 (with SunOS)

The next release of CAPS (CAPS96) will support automated Ada95 program generation, and will run under SunOS 5.4/Solaris 2.4, X11R6, and OSF/Motif 2.0. To run CAPS for this release, you will need the following:

- Architecture: SPARC station
- Operating System: SunOS 4.1.1 or later with OSF/Motif 1.1.2
- Windows environment: X Window System version X11R4 or later
- Memory: 32 MB
- Swap space: 64MB
- Disk space: 130MB
- Compiler: SPARCompiler Ada 1.1 with SunOS
- Interface Tool: TAE+ version 5.3 (recommended but not required)
- Editing Tool: VADSedit (recommended but not required)

2. Program System Description Language (PSDL)

The main stages in the CAPS method are system design, construction, execution, and debugging/modification. System design begins by specifying the requirements using Prototype System Description Language (PSDL). Designers model the system's communication structure and timing and control constraints using computational graphs. CAPS then automatically transforms these graphs into written PSDL. The PSDL is transformed into working Ada software. This iterative process is illustrated in Figure 4.

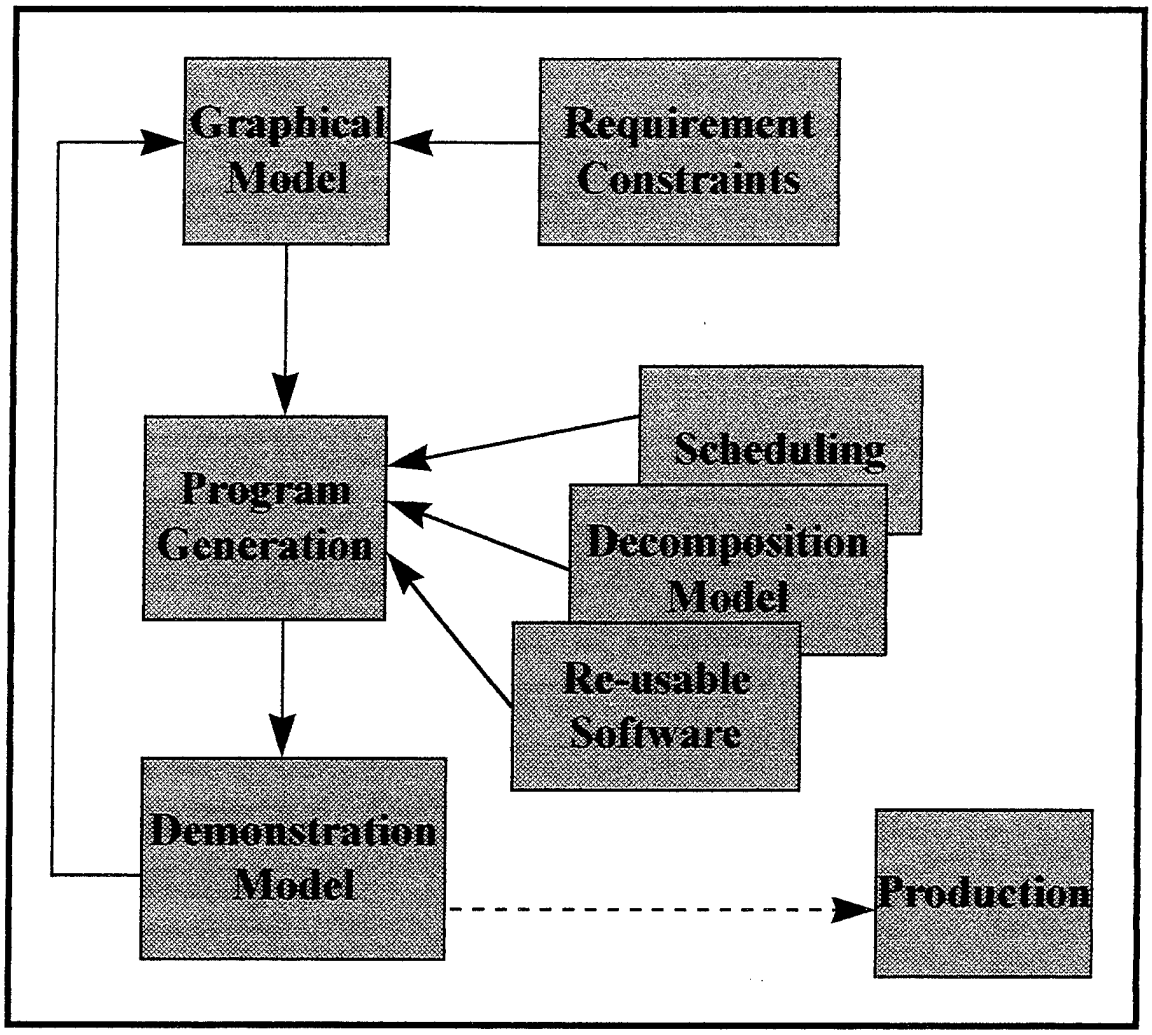


Figure 4. CAPS Iterative Prototyping Process

PSDL is well matched for use with Ada. Ada is required by many large government contracts and is becoming the programming language of choice for an increasing number of DoD software systems. Partially urged by the DoD "Ada Mandate" [Ref. 9], software developers are compiling large libraries of reusable Ada components for practical use. Ada is convenient for implementing PSDL because the mechanisms of Ada support the features of PSDL. This allows for easier interfacing to the Ada reusable components.

PSDL provides much of the power behind CAPS capabilities. It is a high-level "fifth-generation" language (5GL) with hybrid graphical and textual features. PSDL represents a significant advance over fourth-generation languages (4GL). 4GLs address a limited application domain, and "specify more of what a system is or what it should do, and less of how to do it". In comparison, PSDL addresses a wide range of systems, and specifies not only what a system should do but also the component interfaces and how they are connected. Designed specifically to support the specification of real-time systems and to help organize and retrieve reusable components in a component library, PSDL has facilities for recording and enforcing timing constraints and for modeling the control aspects of the real-time systems. [Ref. 10]

The power of PSDL is complemented by the other components that make up the CAPS tool set. These include a design entry facility, automatic design interface, program generation, a software base, a project control system, an execution support system, and a user interface.

3. CAPS User Interface

The user interface includes the graphic editor and syntax-directed editor and a browser. The two editors provide a user-friendly environment for the software engineer to construct a prototype using graphical and textual objects. The browser allows the user to view reusable components in the software Database. The editing tools convert the graphic objects into PSDL programs that are translated into Ada code for execution. The Database provides facilities for software reuse, automated system management and version control. It also keeps track of the PSDL descriptions and Ada implementations for the reusable

software components. The design portion of the Database coordinates concurrent design team efforts and manages the various design versions and documents.

4. Software Database

The software Database system consists of a software base and a design Database which provides facilities for software reusability, automated system management, and version control. The function of the software base is to maintain the PSDL descriptions and ADA implementations for all reusable software components in CAPS. The design Database coordinates the efforts of the software engineering team and manages the different versions and alternatives of the design and documents they produced.

5. Execution Support System

The execution support system consists of a translator, a static scheduler and a dynamic scheduler. The translator generates code that binds designer-supplied code together with reusable components extracted from the Database. The schedulers create the real-time schedule and drivers needed to ensure timely execution of the prototype. The Evolution Control System (ECS) supports project control by keeping track of proposed, ongoing, and completed changes, as well as automatically propagating change consequences by constructing the set of possibly affected modules, automatically scheduling project tasks, and assigning them to designers. This automated feature assists the engineer in making the correct design choices. The merger facility integrates the design decisions from concurrent design updates into the prototype. [Ref. 10]

D. HOW CAPS RESOLVES THE SOFTWARE ENGINEERING CHALLENGE

CAPS bridges the gap between the two prototyping methods (rapid and traditional) by providing automation that enables inexpensive creation of a rapid prototype of the full software system early in the software life cycle. In addition, CAPS enforces software language and design standards that discourage risky methodology shortcuts in the design process. This enables requirement specifications refinement up front, and potential problems to be eliminated prior to the next phase.

Prototyping and automatic code generation reduce maintenance costs by ensuring that modified requirements are valid before they are implemented and by alleviating the brittleness of manually constructed code with respect to real-time constraints. CAPS supports an iterative prototyping process characterized by exploratory design and extensive prototype evolution. CAPS provides improvements over contemporary software development tools and methodologies by using a fifth-generation prototyping language that enables automated real-time software development. CAPS can be used as a tool throughout a product's life cycle to assist the engineer through software development phases such as feasibility studies, requirements analysis, design, code and even acquisition. It is used as an educational tool in the teaching of software engineering and as a research tool used to design large control systems. CAPS has also successfully demonstrated a capability to support the development of large, complex, embedded software systems. CAPS has potential as a full-featured software development tool and is a prime candidate for technology transfer programs.

1. CAPS and the Software Reuse Initiative

Prominent among DoD policy initiatives that attempt to wrest more utility from fewer resources is the "Software Reuse Initiative" (SRI). The SRI is a sophisticated and comprehensive program dedicated to improving software acquisition policy and doctrine for the DoD. The SRI is an important step forward in the current milieu of government acquisition and regulatory reform. SRI addresses both management and technological challenges inherent in software development and attempts to introduce a measure of predictability into the process. [Ref. 11]

The premise of SRI is that software reuse principles can improve the way that software-intensive systems are developed and supported over their life-cycle, increasing return on investment in terms of reducing cost, time and effort. SRI literature notes several instances of successful software asset reuse to sustain the premise. Examples include a 26% reduction in labor hours required to maintain the Restructured Naval Tactical Data System, a 350% improvement in the on-time delivery rate of the software for Fujitsu's electronic switching systems, and estimated cost avoidance of $479.9 million for the Army's Tactical Command and Control System [Ref. 11].

a. SRI Definition and Initiatives

Software reuse is defined as the process of implementing or updating software systems using existing software assets [Ref. 11]. This broad definition includes not only coding, but also design, requirements, user documentation, concepts of operation, documentation, test cases and data. The goals of the initiative are to:

- Improve the quality and reliability of software-intensive systems

- Provide earlier identification and improved management of software technical risk
- Shorten system development and maintenance time
- Increase effective productivity through better utilization and leverage of the software industry.

b. SRI Strategy

SRI implementation strategy addresses both management and technical challenges. First, SRI calls for a high degree of managerial professionalism among DoD software developers by emphasizing the use of disciplined methodology, including a "software development agreement" that ensures "good requirements analysis and a continuing dialogue between customer and provider" [Ref. 12]. Next, SRI encourages the use of technical devices such as object-oriented techniques, libraries of reusable assets, computer-aided software engineering (CASE) tools, application generators and languages, and architecture development tools, among others. Finally, SRI provides a conceptual framework for effectively managing reuse technology. This framework attacks the common technical failures discovered in software development projects, particularly the failures found in efforts to reuse software assets [Ref. 12]:

- Faulty software reuse assets.
- Undocumented interfaces.
- Inadequate searching/browsing/lookup mechanisms.
- No facility for exceptions and overrides.
- Software overhead required to compile, link and execute the reusable module.
- Inadequate configuration control of the versions of reuse assets.

- Too little control over what is put in a library.

CAPS goes above and beyond the technical and managerial solutions called for by the Software Reuse Initiative, meeting and exceeding every goal established by the SRI program. CAPS meets the SRI imperative for predictability and provides significant improvements over contemporary software development tools and methodologies. By using the fifth-generation prototyping language, PSDL, CAPS automates much of the software development process. CAPS and PSDL incorporate both software component and software design databases, which allow significant software asset reuse.

2. CAPS and Rapid Prototyping

Prototyping allows software engineers to isolate problems in both requirements and designs. It has been used to test a design before beginning full production, and is recognized in DoD policy documents as a desirable element in the software design process. The underlying disadvantage is that prototyping can encourage ill-advised shortcuts through the development life cycle . Prototyping has also proven that it can become an expensive process if not properly implemented. Prototyping methods have been used to reduce errors by testing a design before beginning full production. It allows software engineers to isolate problems in both the requirements and the designs. [Ref. 13]

In the past, engineers could choose between rapid or full-systems prototyping. Rapid prototyping builds prototypes of selected components of a system. It allows the creation and testing of input designs, output designs, terminal dialogues, and simple procedures, but cannot test the interfaces necessary for a complex, fully integrated system. Full-systems prototyping,

however, is more time-consuming and expensive, and requires the use of a fourth-generation language or application generator (4GL/AG).

Rapid prototyping is a means for stabilizing and validating the requirements for complex systems by helping customers visualize system behavior prior to detailed implementation. Embedded software systems with hard real-time constraints are used for many control functions in mission-critical applications like command, control, communications, and information systems (C^3I). Development of these systems is plagued with incorrect requirements, inadequate reliability, and excessive maintenance cost.

The new method of rapid prototyping provided by CAPS greatly enhances software development predictability. CAPS combines new, state-of-the-art, technology with traditional, proven methodology to enable software engineers, for the first time, to create a full-systems prototype of the intended software system both rapidly and economically.

E. ANALYSIS OF BENEFITS OF CAPS TECHNOLOGY

In analyzing the CAPS benefits to DoD, commercial industry, and the software engineering community as a whole, the primary thesis question can be answered: "What software engineering problems can CAPS solve for DoD?" There a number of distinct benefits of using computer-aided prototyping. Technology transfer of CAPS enables these benefits to be realized not only by DoD, but by the commercial industry as well.

1. Reducing Uncertainty and Risk

When a business is experiencing problems, it may be hard to determine the source of the problem. If the source can be identified and a software solution is required, the development of a prototype will allow managers and users to understand the problem better

and clarify any grey areas. The use of a prototype helps to focus people's attention on the problem, goals and solutions. The primary benefit of software prototyping is the reduction of the risks and uncertainty involved in the development of a software system. The term risk refers to the possibility of developing a software system that is incorrect or wrong and is therefore of a poor quality.

The consequences of building software that is incorrect, or of poor quality, can be disastrous for those involved in the software development process and perhaps fatal for the users of the system. Uncertainty can be defined as: [Ref. 14]

> ...the difference between the knowledge already possessed about a problem and that which is needed about it to arrive at an acceptable solution.

There is a direct relationship between uncertainty about an information problem and the likely maintenance costs of the software system developed to solve the problem. Thus, reducing uncertainty leads to a reduction in maintenance costs. Any uncertainty of a problem at a conceptual level by the developer has serious consequences in that there is a significant chance that the design and functionality of any software system developed will be wrong. A software system developed with significant validation errors will have to be rewritten. [Ref. 14]

2. Minimizing Time and Cost of Development Cycle

Another benefit of software prototyping, when compared with the traditional project life cycle, includes the reduction in time and expense of the testing phase of the life cycle. This is due to a shift of testing to the requirements definitions phase. A study by Barry

Boehm et al., found that systems developed using software prototyping cost 40% less, and required 45% less effort.

A further example noted during the course of this research was provided from a DoD software development facility in Virginia. An avid supporter of the CAPS project, the Naval Suface Warfare Center, Port Huneme Division, East Coast Operations (NSWC PHD ECO) in Damn Neck Virginia, are extremely interested in working with the NPS software engineering group to enhance CAPS for the development of distributed computer systems. While developing distributed systems at NSWC PHD ECO, difficulties were encountered during the design phase in determining software requirements. In this case, funds were dispersed for actual hardware to determine the data for each of the components within the system. Limited prototyping was done. Hardware components were changed during the development or testing phase after discovering the hardware chosen could not meet the requirments or when new requirements were introduced after development had began. The CAPS solution would eliminate these types of costly setbacks.

3. Increasing Acceptance of the New System

Prototyping generally softens the blow of the introduction of a new computer system or software application. Because users know about the new system before implementation, there is a reduced resistance to the changes the new system brings. The fact that the system's users are involved in the development process may mean they may even welcome the new system. This involvement by the users provides critical input along each phase of the development process, thus ensuring the final system meets the requirements.

4. Avoidance of Opportunity Costs

Another benefit is the general avoidance of opportunity costs. As the partially developed prototype is tested by the end-users after each iteration, they are also incurring some benefits of using the system. Thus, the system in development begins to satisfy some of the requirement for which it has been developed before actual implementation. As users have a working system at their disposal earlier, considerable cost savings can be made by its use. The earlier the implementation of the system, the earlier cost savings can be made, which can be critical in gaining a competitive edge.

5. Better Project Management

Prototyping is a good development management tool. It incorporates the oldest management strategy of divide and conquer and breaks large tasks into smaller deliverable results. Depending upon the iteration of the prototype, the project will focus upon developing a specific part of the system. As a working prototype is being developed and tested, the developer is forced to consider training requirements, data validation and verification, performance and disaster recovery at an early stage of development. Although these factors may not be as important in early iterations, the developers soon get an idea of the true extent of the work required. Therefore the planning and costing involved within prototyping projects are more accurate.

From a DoD Project Manager's point of view, this information is invaluable. All estimates, budgets, schedules and resource requirements are identified early in the project. The managers of the requesting function and users of the system will notice the Manager's in-depth knowledge of the project. Therefore they are happier and more co-operative. The

manager often gains political credibility for his/her insight. Developer's realize at any early stage the true extent of the problem and are better motivated. In addition, sudden budget cuts or project descoping due to changing circumstances does not deprive the systems developer of the chance to deliver results, as some usable code will have already been written.

6. Bridging the Communications Gap Between Developers and Customers

The prototype acts as an excellent communication tool between software developers, project managers, managers of the function requesting the development and users. Often non-technical personnel can be dazzled by the jargon that surrounds a software development project. The use of the prototype eliminates the need for jargon, and prevents misconceptions and misunderstandings from occurring.

7. Other Benefits

Managers (unfamiliar with the software development process) often feel uneasy of the length of the systems requirements definition. They view the start of the actual programming as an important milestone in the project life cycle. Developers themselves sometimes feel that the systems requirements definition is a burden and can be unaware of the importance this phase has on the outcome of the whole project. The use of prototyping helps to counter both misconceptions: the manager's see the real work being done at an early stage, and developers get their teeth into some coding at an early stage. In actuality, the prototyping phase strengthens the requirements definition, and improves the likelihood of the project's success.

F. SUCCESSFUL SOFTWARE PROTOTYPING USING CAPS

The CAPS project has received strong support from the Navy including CNO, ONT, ONR, NRL, and NRAD/NOSC. Army agencies that have provided staunch support for this

software prototyping initiative are: Army Research Lab, Army Research Office, and CECOM. The CAPS technology has been adopted by the US Army to support the evolutionary prototyping of real-time software systems that increase productivity, decrease cost, minimize defect insertion and post development software support. The Ada Joint Program Office and the National Science Foundation are heavily involved with CAPS research because CAPS utilizes the DoD mandated programming language, Ada.

CAPS technology has been successful as a research tool in producing software prototype designs of large warfighter control systems. Examples include:

- C^3I Station
- Cruise Missile Flight Control System
- Missile Attack/Defense Systems
- ATACMS

In the design of these software intensive systems, CAPS demonstrated its capability to support the development of large, complex, embedded software. CAPS has shown extraordinary promise in defining design requirements for non-DoD software applications such as:

- Fish Farm Project
- Robotics
- Automated factories
- Telecommunication systems
- Computer controlled consumer appliances

1. C^3I Station

In 1992, The CAPS R&D team conducted an experiment [Ref. 15] to evaluate the effectiveness of the CAPS rapid prototyping methods and computer-aided design environment. The goal of the experiment was to prototype a generic command, control, communications and intelligence station and to generate the Ada code from the prototype's specification automatically. The results of the experiment were successful and proved that it is feasible to use computer-aided prototyping for practical, real-time Ada applications.

2. ATACMS

The long range Army Tactical Missle Systems (ATACMS) is one of many C^4I "End-to-End" Systems slated by DoD for development. The Director, Test, Systems Engineering & Evaluation (DTSE&E), were interested in modeling and simulations which could assist in determinig and verifying interface and component requirements. To this end, the CAPS research group at NPS assisted in the evaluation and refinement of system requirements for the ATACMS [Ref. 16] as well as demonstrated the capabilities and suitability of CAPS on a large real world system. A refined ATACMS model was developed and is currently available to ODTSE&E for use. CAPS was successful in identifying critical operational issues outlined in a Memorandum of Agreement covering the development of the ATACMS. The comprehensive use of CAPS in modeling a real-world system has confirmed the essential qualities of the system while identifying several issues for future enhancement. CAPS represents a significant opportunity for DoD to address those software development issues resulting from shortcomings in the requirements process.

III. THE PROPOSED METHODOLOGY

Although technology transfer is a complex process that is not yet completely understood, DoD, the commercial industry, and academia are beginning reap the myriad benefits as a result of these technological liaisons. Understanding of what technology transfer is and its benefits are key to realizing this process to its fullest potential.

A. DUAL-USE TECHNOLOGY AND TECHNOLOGY TRANSFER

1. Dual-Use Technology

The nature of DoD's business dictates that it be on the leading edge of creating technologies and products that, in the early development stage, have no commercial market and are beyond the normal level of risk acceptable to industry. Often years later, after these technologies have been demonstrated in defense applications, they may be adopted and employed by the non-defense, i.e., commercial, industrial sector for product design and production.

What is new in today's environment is DoD's proactive strategy to involve the commercial industrial base as soon as possible. Rapid advances in commercial technology combined with declining U.S. defense budgets have, in many cases, rendered DoD's traditional, defense-unique approach to technology development and procurement less affordable and less effective than in the past. It is critical that defense programs take advantage of cost-conscious, market-driven commercial production, and leverage the huge investments in leading-edge process technologies made by private industry. It is also important that defense technologies and systems keep pace with the rapid product development cycles driven in critical areas by a highly dynamic commercial sector.

Dual use technology policy is a key component in DoD's investment strategy for maintaining the performance superiority and affordability of U.S. military forces in this new technological and economic environment. Elements of the dual use technology investment strategy serve to: (1) ensure that key elements of the domestic commercial technology base that are critical for national security remain at the leading edge; (2) support the transition of defense-sponsored technology and the integration of military production with the commercial base; and (3) facilitate insertion of commercial technologies into military systems. [Ref. 17]

2. Defining Technology Transfer

Technology Transfer is a process through which technical information and products developed by the Federal government are provided to potential users in a manner that encourages and accelerates their evaluation and/or use. More than merely disseminating information, technology transfer techniques feature marketing of federally-developed technology and products. Many definitions of technology transfer have been developed to suit the needs of the individual organizations or activities. A common definition for technology transfer used by the commercial community is defined as [Ref. 17]:

> "The process by which existing knowledge, facilities or capabilities developed under federal R&D funding are utilized to fulfill public and private needs."

This process can be very simple or quite complex, and basically involves a technical resource (e.g., NPS, Army Research Lab, federal laboratory), a user (e.g., Naval Surface Warfare Command, small business), and some interface connecting the two. "Technology transfer" includes a range of formal and informal cooperations between federal laboratories and the public and private sectors. The purpose of the transfer is to strengthen the nation's

economy by accelerating the application of this developed technology and resources to private and public needs and opportunities. Product improvement, service efficiencies, improved manufacturing processes, joint development to address government and private sector needs, and the development of major new products for the international marketplace are the results of successful technology transfer efforts.

Technology transfer sometimes appears to be a simple communication process. However, analysis of the technology transfer process has yielded a somewhat predictable learning pattern where, comprehension of the technology is first achieved, then comes the interpretation of how the technology can be used to solve a problem; finally, the actual application of the technology to solve a problem. [Ref. 17]

Much of the Federal government's annual research funding is allocated for the development of product prototypes and processes, and applied research relating to existing or potential business markets. These efforts often result in many new, usable technologies that have direct commercial applications. In 1980, the U.S. Congress formally recognized the value of these products and the government's responsibility to the Nation's taxpayers to make this wealth of scientific information readily available. [Ref. 17]

The Stevenson-Wydler Technology Innovation Act of 1980 [Ref. 18], Public Law 96-480, laid the foundation for technology transfer within the Federal government. This law recognized the need for enhanced information dissemination from the Federal government to private industry. It also required Federal laboratories to take a more active role in technical cooperation with potential users of Federally-developed technology.

Potentially, government, industry, and academia make excellent partners in the technology business. Often government laboratories are involved in research of advanced

technology that have very high performance standards required for military, aerospace or other such applications. This research is often long term and does not always have specific application goals. Universities tend to focus more on fundamental science, and, of course, education. Industry has much more of "bottom-line" perspective on cost-effectiveness and on specific applications and are much more concerned about the profit potential of a technology.

Most technology transfer is thought of as a one-way street from government funded laboratories and institutions to industry; however, the truth is that the relationship developed through technology transfer will greatly improve operations in both segments of society. Industry can be looked upon as a vast resource of solutions for many of the technical problems faced by government research institutions. Due to the unique perspective on R&D problem solving, industry has proven to be very effective at getting things done.

Technology transfer is, therefore, not a single event nor a simple process of stimulus and response. It is a complex process growing from numerous sources of planning, policy making, learning and creativity. These events or acts occur at all points of the transfer process, from initial testing to full-scale productivity. More importantly, technology transfer should occur at all levels of society, involving policy makers, researchers, engineers, production workers, instructors and students. Technology transfer should also occur at many different locations such as the workplace, all levels of government, in academia, and even at home. Rapidly developing information technology and the development of an "information superhighway" will facilitate the transfer of technology at all levels of society. [Ref. 17]

3. Adopting New Technology

Diffusion of information about new technology is predominantly a process of communication. Anything that impedes communication within the organization, as well as

within the environment it interacts in, will jeopardize the successful implementation of the technology within the organization.

The decision to adopt new technology is heavily influenced by environmental factors. These are the events occurring in the industry, market, country and the world in general, within which the organization interacts. Ultimate users of new technology must do something different from what they have done in the past. They must change their behavior patterns. A consequence of this is that it cannot be expected that the recipients will respond to new technology quickly. They must not only assimilate facts relevant to the technology, but also change behavioral patterns that would lead them to use the technology. Also, it is human nature to resist ideas, especially those originating from outside of the organization, and this can lead to myopia or tunnel vision [Ref. 18]. A clear implication is that technology transfer requires time, patience and opportunities to experiment with a new technology.

The decision to adopt new technology is also heavily influenced by organizational factors. Organizations are more likely to be willing and able to adopt technologies that offer clear advantages, do not drastically interfere with existing practices, and are easier to understand. Adopters look unfavorably on innovations that are difficult to evaluate or which benefits are difficult to see or describe. Additionally, the decision to adopt technology is influenced by the technology itself. All other factors being equal, if the technology fails to live up to the expectations of the eventual users, then its implementation will not be successful.

B. LAUNCH POINTS FOR THE TECHNOLOGY TRANSFER PROCESS

From the Stevenson-Wydler Technology Innovation Act of 1980, which established Offices of Research and Technology at major Federal laboratories, to the Defense Authorization Act for FY 1993 which outlined the future path for technology innovation in

DoD, legislation is firmly in place to sustain DoD's efforts to achieve its dual-use technology and technology transfer goals. Executive Order 12591, facilitating access to science and technology, requires DoD to identify funded technologies potentially useful to US industries and universities. Technology transfer launch points have been established throughout DoD and industry as a result of this history of enacted legislation.

1. Office of Technology Transition (OTT)

Under section 4225(a) of PL 102-484, Div.D., Title XLII, of the National Defense Authorization Act for Fiscal Year 1993, and now codified at 10 USC sec. 2515, the Secretary of Defense was required to create an Office of Technology Transition. Section C requires that the Office:

- monitors all R&D activities that are carried out by or for the military departments and Defense Agencies;
- identifies all such R&D activities that use technologies, or result in technological advancements, having potential non-defense commercial applications;
- serves as a clearinghouse for, coordinates, and otherwise actively facilitates the transition of such technologies and technological advancements from the Department of Defense to the private sector;
- conducts its activities in consultation and coordination with the Department of Energy and the Department of Commerce; and
- provides private firms with assistance to resolve problems associated with security clearances, proprietary rights, and other legal considerations involved in such a transition of technology.

In addition to the above missions, OTT provides oversight of the Defense Technical Information Center (DTIC), and is responsible for the following programs:

- Federal Defense Laboratory Diversification (FDLD)

- Small Business Innovation Research (SBIR)
- Independent Research and Development (IR&D)

OTT also participates in the formulation of DoD policies pertaining to Dual Use and Manufacturing Science and Technology Programs.

2. Office of Naval Research (ONR)

The Office of Naval Research (ONR) coordinates, executes, and promotes the science and technology programs of the United States Navy and Marine Corps through universities, government laboratories, and nonprofit organizations. It provides technical advice to the Chief of Naval Operations and the Secretary of the Navy, works with industry to improve technology manufacturing processes while reducing fleet costs, and fosters continuing academic interest in naval relevant science from the high school through post-doctoral levels.

Established in 1946, the ONR was the first federal agency with the mission of encouraging, promoting, planning, initiating and coordinating naval research. ONR's mission is to maintain a close relationship with the international research and development community to support long-range research, foster discovery, and nurture future generations of researchers, produce new technologies that meet known naval requirements, and provide order-of-magnitude innovations in fields relevant to the future Navy and Marine Corps.

ONR's software program consists of developing the foundations for the verifiably correct design and construction of complex software systems. Research is supported in linear logic and related proof systems, software testing, formal algorithm derivation, and formal proof of correctness, among other areas. All of these research areas can be supported by the CAPS functionality.

3. Navy Offices of Research and Technology Applications (ORTA)

In order to take full advantage and derive maximum return on the country's technological investments, Congress passed legislation to encourage the transfer of federally funded technology to commercial industry. To promote this transfer, Congress mandated each federal laboratory to create an Office of Research and Technology Applications (ORTA). Navy ORTA representatives identify and assess potential technologies and ideas from their own laboratories that may be transferred to state and local governments, industry, or universities. These representatives also assist in domestic technology transfer efforts.

4. Technology Transfer Process at NPS

The Naval Postgraduate School (NPS) in Monterey, California is a DoD funded facility that fosters an extensive academic research and development environment. NPS is considered a federal laboratory for R&D and works closely with DoD agencies and commercial industry to develop emerging technology in various fields of study. Funding for R&D projects are provided by parent DoD agencies, industry, and NPS itself. The Naval Post Graduate School has been designated as a Navy ORTA and has recently established an Office of Research and Technology. The research office aids the R&D team in developing a plan of action for projects that have potential DoD use, as well as potential for commercialization. In this particular environment, the NPS process of transferring new technology within DoD relies on the ability and resources of the project R&D team to organize and "market" their innovation.

An important mechanism to facilitate technology transfer to potential commercial sponsors at NPS is the Cooperative Research and Development Agreement (CRADA). A

CRADA between a federal laboratory and industry or academia enables the commercialization of DoD-developed technology to the technological and financial benefits of both parties. CRADA's are overseen by ONR guidelines. [Ref. 19]

C. THE RECOMMENDED TECHNIQUE

The technology transfer process can be explained with the implementation of six phases; technology innovation, technology confirmation, targeting technology consumers, technology marketing, technology application, and technology evaluation. Each of the six phases is briefly described with examples of key actions which demonstrate movement through the process and indicators of transfer which serve to document progress. Actual key actions and indicators of transfer for the six phases can take a multitude of forms, with phases at times overlapping.

1. Phase One: Technology Innovation

The technology transfer process begins when a scientist or principal investigator (PI) starts communicating ideas of how science can be used to solve a problem or improve a situation in a research priority area. This technology innovation phase is represented by the exchange of information which takes place between the scientist, colleagues and administrators to advance ideas on the application of science. Any assistance which can be given to support other scientists in communicating their theories will facilitate the technology transfer process.

Such assistance can take the form of encouragement for scientists to communicate ideas with a diagram depicting how different factors interact within a research project. A diagram is the first step toward communicating and refining ideas. The next step would be

when the scientist starts discussing his or her theories with colleagues. This activity may aid the scientist in further refinement of the theories and gains suggestions for other possible commercial applications of the technology. In-house seminars and group discussions should be actively organized and supported by all scientists to encourage analysis and support or development of ideas.

After refining theories arising from the technology innovation, the scientist should submit research proposals communicating the concept to the appropriate funding agency. Such proposals should include plans as to how the research will in fact be applied. One key to obtaining acceptance of a new technology proposal is the level at which the PI is proactive in suggesting end uses for the technology they have created.

2. Phase Two: Technology Confirmation

The technology confirmation phase is represented by the PI or co-PI first conducting research which provides data in support of the underlying theory about technology and then communicating the results to colleagues, peers and administrators. Indicators documenting progress could be in-house reports which communicate research success to colleagues and administrators. Indicators of transfer in this phase would be in-house reports, presentations and or publications substantiating research success, which aids science liaison within the science community.

3. Phase Three: Targeting Technology Consumers

During the third phase of the technology transfer process decisions need to be made concerning who needs and can potentially benefit from the technology. The people involved in the targeting technology phase would be scientists, development team members, and

marketing personnel. These specialists need to be aware of factors such as cost, convenience, etc. which influence users' acceptance of new technology or factors which might serve to prevent the adoption of technology. This phase encompasses a multitude of factors for socio-economic considerations for targeting technology change. Indicators of transfer for this phase would be the interactions of science, business, and marketing personnel to "brainstorm" technology acceptance considerations.

4. Phase Four: Technology Marketing

The technology marketing phase of the process is concerned with disseminating the technology beyond the research center. Key actions for technology innovation liaison involve the talents of scientists, business leaders and marketing specialists to educate potential consumers to the social, economic and environmental benefits of the new technology. During this phase frequent interaction between research and marketing personnel is suggested. Substantial benefits can be realized by first establishing a demographic profile of anticipated consumers before organizing communication channels. Knowing where the potential client is usually gains knowledge of specialized products and or services will influence the selection of the proper communication methods.

5. Phase Five: Technology Application

The technology application phase concerns the understanding of users or consumers behavior and establishing predictable steps to monitor the commercial application of technology. The talents and skills of social and financial consultants, and marketing personnel are required to identify consumers' behavior and application patterns.

Mathematical models which weighs social and economic factors and their influence on the diffusion of technology innovations are widely used. The ratio of the number of consumers applying the technology to the number of potential consumers needs to be carefully monitored, to establish the market share reached. [Ref. 20]

6. Phase Six: Technology Evaluation

The sixth phase of the technology transfer process documents the success or lack of success of the technology to be adopted. Key actions for the technology evaluation phase are to establish assessment criteria for authenticating socio-economic and environmental benefits or harm. Evaluation guidelines should be established based on the type of technology innovation proposed. Assessing technology transfer effectiveness generally requires specific criteria which provides a basis for measuring the extent to which key actions have been attained. The method of defining specific criteria for indicators of transfer is essentially moving from broad to specific actions. The technology transfer process ends when the PI and co-PI reports the evaluation findings back to the funding agency.

IV. PREPARATION AND EXECUTION FOR TECHNOLOGY TRANSFER

This chapter consolidates the research uses that information to answer the primary research questions: **How can the current research and development of CAPS be collectively organized to provide effective technology transfer to similar projects for DoD and the commercial environment?**

A. INTRODUCTION

The initial version of CAPS (CAPS Release 1) is currently being distributed by the National Ada Clearinghouse on CD-ROM media. Over 5,000 copies have been dispersed by the distributor to those interested in the technology. This endeavor to make CAPS available as freeware to DoD agencies and interested commercial technologists has mainly been directed at soliciting interest for further funding of the CAPS development effort. Thus, initial steps in the technology transfer process has already begun. CAPS has unknowingly embarked down the technology transfer path in a somewhat haphazard way. Widespread distribution of the CAPS as freeware serves its initial purpose of making others aware that the technology exists. However, as derived from the recommended technique of Chapter III, for the transfer to be successful, preparations of key materials which facilitate the understanding of such technology must first be in place. This is evidenced by the large number of inquiries received from those that have obtained a copy of the CAPS application. Initial feedback questions lets the development team plan for future revisions or upgrades to the existing application. It also identifies the areas of weakness in the documentation and help features of the application. This chapter outlines a plan of action to continue the technology

transfer process. As part of this process, the technical documentation, electronic access media, and marketing materials required for support are defined.

B. CAPS TECHNOLOGY TRANSFER PREPARATION PLAN

The CAPS Technology Transfer Preparation Plan (Appendix A) was derived from information obtained through research of technology transfer and its phases, self-taught knowledge of the CAPS application, interviews conducted with the Director of the NPS Office of Research, and interviews with members of the CAPS R&D team. It is the intent of this plan to lay a framework for successful CAPS technology transfer implementation.

The success of the technology transfer plan rests on the individual success of each phase and the vigor at which it is pursued. Because there are a number of external factors that impact successful technology transfer, i.e., funding and culture acceptance, a time frame for each phase of the process is difficult to project. The focus is therefore placed on preliminary efforts required to prepare the CAPS technology for understanding by current and potential users, and to obtain acceptance of this new technology within DoD. Once benefits come to fruition within DoD, further actions to initiate and complete the technology transfer process to commercial industry via a CRADA may be the basis for follow-on theses.

1. Conveying the Technology

Phase I of the technology transfer plan puts the responsibility of conveying the innovation of CAPS technology on the principal investigators. They are the subject matter experts and are in the best position to collaborate with external sources and colleagues in the software development environment. This phase of the plan is ongoing process throughout the technology transfer effort. Thus far, it has been successfully implemented with initial

actions taken by the PI and co-PI's of the CAPS project. Indicators of successful transfer of the innovation are evidenced by the numerous articles and papers published on CAPS by the team in leading technology publications and journals like IEEE Software and STARS Newsletter. These efforts have produce widespread interest in CAPS technology within the DoD software engineering community and continues to cultivate. Several CAPS sponsors like the Army Research Lab (ARL), are utilizing the CAPS tool in the planning and design phases of new software intensive, hard real-time systems.

2. Funding and Manpower Resources

Not unlike other DoD components, NPS constantly competes for diminishing budget and manpower resources. Similar to other academic R&D environments, the CAPS R&D team must also seek sponsors for continuous funding support of the development effort.

Initial funding for the CAPS project ($2.5 million) was awarded through the Presidential Young Investigator Award received by the principal investigator, Professor Luqi. Currently, NPS supplies the majority of the funding with a small amount of cash flow from CAPS sponsor organizations: Army Research Lab (ARL), National Science Foundation (NSF), Naval Ocean Systems Command (NOSC), Army Research Office (ARO), Naval Surface Warfare Center (NSWC), and the Ada Joint Program Office (AJPO). The NPS ORTA holds funds earmarked for projects that are candidates for dual-use technology and technology transfer. These funds should be actively sought by the CAPS PI. In an academic environment such as NPS, there are many R&D project candidates competing for these funds. Funding is not only crucial for the continued development effort, but also needed to implement a successful technology transfer plan. Table 1 shows approximate funding required to outsource portions of the technology transfer effort if in-house resources are not

utilized. Funding amounts were computed based on surveying current industry standard prices for technical writers, multimedia production services, and marketing consultant services.

TASK	MANPOWER REQUIRED (effort in manmonths)	FUNDING REQUIRED
Develop CAPS Technical Documentation		
- Tutorial	2	$6,000
- Quickstart	.5	$1,500
- Users Manual	6	$18,000
- Installation Guide	.5	$500
Produce CAPS Multimedia CD-ROM Presentation		
- Development and Production	6	$5,000
- Reproduction	N/A	$1,000
- Distribution	N/A	N/A
Consultant Services		
- Marketing	1	$10,000.00
Total	**16**	**$42,000.00**

Table 1. CAPS Technology Transfer Outsource Funding Requirements

The primary source of research manpower at NPS is extracted from the student population. PI and co-PI typically anchor the development team throughout the R&D effort. In addition to project research, they also take on the role of project planning and management while performing other academic commitments. Although the PI serves as a stabilizer for the R&D team, the development process can become lengthy and incremental due to the continual turnover rate for graduate students at NPS. Although progress in developing the CAPS tool has been substantial, project management is lacking the continuity and cohesiveness neccessary to be effective.

An answer to this dilemma is the outsourcing of the project management responsibilities. Funding dollars for technology transfer would allow for the contracting of manpower dedicated solely to the management of the CAPS project. Contracted personnel could perform most of the tasks outlined in Appendix A, freeing the PI and thesis students to concentrate on development of the product. "Permanent" manpower provides for cohesive and focused project management, skills that the PI or co-PI may not posses. Unfortunately, in the abscence of such funding, project success relies on the PI's management skills, political prowess, funding and budgetary knowledge, and ability to convey the innovation to potential funding sources.

3. Collaboration with NPS ORTA

The NPS ORTA provided additional understanding to this research of the steps involved in the technology transfer process. The office assisted in the development of a plan of attack for marketing the CAPS application to DoD agencies and provided guidance to preparing a CRADA for commercial collaboration.

According to the NPS ORTA, after evaluating potential benefits to be gained from the new technology, R&D efforts should focus on identification of real-world problems to be solved by the proposed new technology for DoD and the commercial industry. Once these problems are identified, the R&D team can construct example prototypes using the CAPS. This provides an invaluable marketing tool that relates the technology to real-world solutions. A plan for marketing the new technology is then devised and implemented. The NPS ORTA agrees that a key factor dictating acceptance of the new technology over a wide user base is the ease of which it can be learned and implemented. The NPS ORTA is an indispensable source of information and direction and should be utilized throughout the technology transfer process.

4. Identifying Potential Users

Identifying potential users of the CAPS application will primarily emanate from the collaboration between colleagues in the R&D world. During this phase, CAPS PI and co-PI's must continue to be proactive in conveying potential benefits of computer-aided prototyping through dialogue with colleagues. Projects that can benefit from the CAPS technology need to be identified. Once targeted, R&D team members can work with the potential user to construct preliminary prototypes that show the CAPS functionality in solving real-world problems.

Various resources are utilized to obtain current research information pertaining to software development projects. One method utilized by the CAPS team is subscriptions to leading edge technology publications and journals. Another widely used method that has emerged over the past few years is the Internet. The Internet explosion has made collaboration easier between federal laboratories, government contractors, and research

universities allowing each entity to showcase current and planned research projects via the World Wide Web (WWW). Active participation in trade shows, technology seminars, and project demonstrations will allow CAPS team members to keep abreast of current projects thereby analyzing them for potential CAPS usage. Another resource that can aid in the search for potential projects are Broad Area Announcements (BAA). These announcements are disseminated by DoD through various sources and seek to solicit innovative technology solutions for a variety of potential projects from academia and the commercial industry.

Another recommendation to promote interest in the CAPS technology is to establish a set of seminars or training sessions at NPS. Potential users, both DoD agencies and commercial industry, can be solicited to send their software developers and program managers to these training sessions. This accomplishes two principal goals. First it provides a source of funding, and second, it allows those potential users to investigate the feasibility of the technology first hand. CAPS PI and R&D team members can conduct the seminars from current course material at NPS.

C. CAPS BRIEFING PACKAGES

As discussed in Chapter III, new technology introduced to organizations is most often met with resistance to change by those who are more comfortable with "business as usual". One of the key ingredients which help combat this resistance is to provide a clear understanding of not only the technology itself, but the end result benefits that this "new technology" has to offer.

To increase the chances of success, this understanding must reach all levels of the potential user organization hierarchy. To facilitate the information dissemination process, three briefing packages have been prepared that can be utilized to accomplish this end, an

executive summary brief, a technical brief, and a program manager brief. These briefs are available for potential users to gain further information on the CAPS technology or for those current users that must seek funding for the implementation of CAPS at their facility. The briefs are constructed in a common presentation application (Microsoft PowerPoint™ 7.0) and can be tailored by the interested party for custom presentations. The briefs are available for downloading via the CAPS Home Page.

1. Executive Summary Brief

Initial understanding of new technology must begin at the top of the organizational structure. Decisions to incorporate new technology often fail due to lack of support from the upper and middle management levels of the organization. Once the managers understand the technology and its benefits, they become the change agents needed to funnel the information downward to the actual implementers of the technology.

This package includes an executive summary of CAPS along with briefing slides that capture the projects essence and potential benefits of use. The Executive Summary Brief is presented in Appendix B.

2. Technical Development Brief

The purpose of this package is to aid other technologists interested in using computer-aided prototyping to design software intensive, hard real-time systems. Insight is given to the theoretical development process used in designing CAPS. This brief is more technical in nature to provide understanding of CAPS design methodology. CAPS component design details are given along with information on how to access on-line libraries of published

research articles by the NPS software engineering group. This package is provided in Appendix C.

3. Program Managers Brief

This package briefs current and potential DoD Program Managers on the functionality of CAPS and how it can be applied as a software engineering tool for software acquisition and development. The brief describes the baseline architecture required to run CAPS, some application prototypes developed at NPS using CAPS, and potential CAPS projects. This brief is contained in Appendix D.

D. DOCUMENTATION

An essential element in any R&D project, documentation, is critical to dissemination of the research efforts to fellow colleagues and interested users. The primary means of supporting users of technology is through the use of documentation. Documentation bridges the gap between developer and user. Proper documentation of the systems functionality will determine the rate at which the learning curve of this new technology will rise. Documentation that is difficult to understand by new users will result in obvious resistance to incorporate the new technology. This documentation has traditionally been in paper form but is rapidly changing. Although the most common form of documentation continues to be paper manuals, other forms of media are gradually taking over i.e, on-line documentation.

This section provides a brief overview of the essential elements in preparing documentation. Based on these elements, an evaluation of existing technical documentation was performed while learning the CAPS application for this research. To facilitate a better

understanding of the CAPS application, recommended improvements to these documents are provided.

1. Product Knowledge

The first step in producing a good document is to know the product which the document is supporting. Typically the system designers and engineers are not the individuals who reproduce the system documentation. This disparity creates a high potential for errors to be introduced into the documentation of the system. To minimize this potential for errors, it is critical that representatives of the R&D team review the draft documentation for technical accuracy. This technical review should be accomplished by someone not involved in drafting the document so that the likelihood of detecting errors is greater. [Ref. 21]

2. Knowledge of Users

The second requirement in producing system documentation is knowing your user audience. Although this audience may vary over time, the document preparation must consider this variance and then write to the lowest common level of the targeted audience. The lowest common level of user anticipated to use the CAPS is the program manager who may have no background in software development. Targeting this level of user obviously requires a more complex document because step-by-step instructions are required whereas for a developer/programmer they are not needed. [Ref. 21]

3. Documentation Quality

The quality of the documentation depends on many factors that are out of the drafter's control. Most documentation, as in this case study, is planned after the fact. Therefore it cannot be used to identify problems and correct deficiencies in the system. At best, the

documentation can only point to weak areas and attempt to guide the users successfully beyond them. [Ref. 21]

Usable, high quality documents for the CAPS product increases the potential for successful technology transfer by providing the user with a "user-friendly" point of reference to answer questions that arise during operation.

4. CAPS Documentation

a. CAPS Tutorial

The current draft of the CAPS Tutorial (Appendix E) provides a training guide that intends to take the novice user step-by-step through the design process of a system prototype by using the majority of the CAPS functions and commands. However, this document falls well short of its intended goal even after several revisions by the CAPS R&D team and previous thesis students. The user level of knowledge assumed by the tutorial is far beyond that of most program managers and even the experienced systems analyst/engineer would question the effectiveness of the document. While performing the tutorial, certain procedures did not function as indicated leaving the user guessing as to the next step in the process. Error messages were not adequately explained in the minimum on-line help that could be accessed. Polling a sample of five new users, with varying software engineering backgrounds, four out of five found the tutorial to be cumbersome and vague. The general concensus was that for the most part, completion of the CAPS tutorial furnishes the new user with only a fair understanding of CAPS functionality.

The shortcomings of this document are attributed to its development by the system designers of CAPS. Additionally, as improvements to CAPS are integrated, revision

of the documentation lags. This accounted for the majority of the problems encountered while perfoming the tutorial.

b. CAPS Users Manual

The current draft of the CAPS User Manual (Appendix F) is intended to provide information on the operations of the CAPS application. In its current form, it provides only desparate information about CAPS features and procedures. Illustrations of CAPS windows show the various functional choices available but lack detailed descriptions of those functions. This document is essential to new users in answering initial operational questions when attempting to use the application and when experienced users are not available. The CAPS Users manual is unusable in present form and requires a dedicated layout plan for development in order to be an effective document.

c. CAPS Installation Guide

The CAPS Installation Guide (Appendix G) provides sufficient loading instructions for the CAPS application. It describes the hardware environment and commands required to implement the CAPS application.

d. CAPS Quick-Start Guide

The CAPS Quick-Start Guide (Appendix H) provides the minimal information needed to develop software prototypes using CAPS. It concentrates on the use of the CAPS editors to create complete PSDL prototypes.by constructing dataflow diagrams and annotating them with timing and control constraints. It instructs the user how to connect to graphical interfaces using the CAPS interface editor and how to execute the prototype via the CAPS execution support system.

This document provides useful information to the user but has not been revised since its first publication in November 1995. The Quick-Start Guide is well constructed and serves its intended purpose.

E. INTERNET ACCESS TO CAPS INFORMATION

The latest version of CAPS, including scripts, enhanced installation instructions, and other useful information is available on the CAPS Home Page (Appendix I) and can be accessed via CAPS NPS website on the World-Wide Web at: http://wwwcaps.cs.nps.navy.mil. The CAPS home page provides access to briefings, papers and published articles references, the current version of the CAPS tutorial, users manual, and installation manual. References to CAPS papers and published articles are available but only a small number have been converted to an accessible html version. Efforts to convert the remaining papers and articles are ongoing. The CAPS home page also provides links to other related technology transfer web sites.

Additionally, CAPS Release 1 is included in Volume 2 of the Walnut Creek Ada CD-ROM Software Technology Conference (STC) Special Edition. The path to CAPS on the Walnut Creek CD-ROM is: /ada/ajpo/source-code/caps. Also, the same CAPS Release 1 is available from the Ada Information Clearinghouse (AdaIC), which provides comprehensive information about the Ada programming language free of charge to the industry. AdaIC is located just outside Washington, D.C. in the offices of the government's Ada Joint Program Office (AJPO). The URL for CAPS there is: http://sw-eng.falls-church.va.us/AdaIC/source-code/caps . The following WWW sites are accessible through the CAPS home page:

- Naval Surface Warfare Center (NSWC), Port Huneme

- Army Research Lab (ARL)
- Army Research Office (ARO)
- Ada Joint Program Office (AJPO)
- Office of Technology Transfer (OTT)
- National Science Foundation (NSF)
- DoD Technology Transition (TechTransit)
- Naval Oceanography Systems Center (NOSC) Naval Research and Development (NRAD)

F. CAPS MARKETING MATERIALS

1. CD-ROM Multimedia Presentation

Promoting a new technology to a potential user organization takes the acceptance by decision makers that control the organizations funding. Marketing the product to these decision makers and potential users is best accomplished by using the latest technology in the multimedia arena. A high-level multimedia presentation captures the essence of the CAPS project and allows for an interactive environment that can be easily viewed by the user. The CD-ROM presentation will be massed produced and distributed in the same fashion as the initial CAPS93 version. The CAPS CD-ROM multimedia presentation is a joint effort initiated as a result of this thesis effort between the CAPS R&D team and Professor Harry Li, Ph.D., Texas Tech University. This product will be utilized to market CAPS to DoD and to commercial industry. The development plan for the CD-ROM presentation is provided to interested readers at Appendix J.

2. Information Brochures

CAPS information brochures serve to provide project information and will be distributed at seminars and lectures. CAPS information brochures are provided at Appendix K.

V. CONCLUSION AND RECOMMENDATIONS

Given that the first release is circulating in only a small number of research laboratories, CAPS has proven its potential to overcome DoD software engineering challenges. In the laboratory, the Computer-Aided Prototyping System has accomplished every goal it set out to achieve in the software engineering environment. CAPS has not only improved the quality and reliability of software systems, but provided early identification and improved management of software technical risk. It has shortened system development and maintenance time, and dramatically increased productivity through better utilization and leverage of software assets. CAPS is an outstanding candidate for DoD's DUT/DTT programs.

Prototyping of complex, software intensive systems by DoD managers has just begun to be widely adopted due to its potential benefits. One such benefit of using software prototyping, discussed previously, is that it allows DoD program managers to perform crucial analysis and feasibility studies of the target system. This is an essential process required to define user and system requirements prior to making contractual commitments. CAPS provides the software engineer with prototyping tools which span the entire software development life cycle, ranging from requirements analysis to system testing and integration.

DoD's efforts to encourage software reuse will lead to reduced cost while promoting compatibility in system design. CAPS leverages software assets through their reuse. This establishes whether initial requirements correctly capture the problem by determining if delivered systems meet requirements. It ensures accurate integration of independently developed subsystems. Consequently, in less time the software is produced at a lower cost

with a higher degree of quality. The potential for CAPS to do more with less is recognized by Navy leaders and CAPS program sponsors, who have ensured that CAPS is available on-line to software systems engineers through the Defense Software Repository System (DSRS). However, CAPS as yet has not been incorporated into the SRI or funded at the levels necessary to assure widespread distribution and use. If software engineers and program managers in the field use CAPS with the same degree of success that has been proven in the lab, CAPS will fulfill every goal of the SRI beyond our most optimistic expectations.

During the course of this research, first hand experience was gained as a novice user of the CAPS application. Having minimal to no documentation in place, the learning process was slow and and at times tedious. This reinforced the fact that both on-line assisstance and technical documentation is critical to aid potential users in understanding the technology. Emerging technology is produced in several different forms for various uses. In the case of the CAPS application, it is essential that the first time user experience as little growing pain as possible during the learning process. The more users struggle with an application in the early stages of the learnng process, the less likely they are to desire to utilize or promote this "great" technology. The technical documentation of the application operation, if prepared properly, will facilitate ease of use and reduce the learning curve significantly. Well prepared documentation lends to successfull technology transfer.

Evaluation of existing technical documentation indicate that document preparation was not well planned by the R&D team. At present, this is the greatest hinderance to the successful technology transfer of CAPS. To be effective in implementing the technology transfer process this documentation must be of the highest quality possible. Due to the continual turnover of students in this academic environment, it is recommended that a

minimum of one (1) man-year worth of effort by a qualified vendor (technical writer) be dedicated to the planning and development of the required CAPS technical documentation. All current CAPS documentation has been produced by R&D team members who are heavily involved in the technical development of the application. The use of student manpower to accomplish these tasks runs the risk of continued low quality documents. This reccommendation does not preclude the use of R&D team members or in-house resources to supplement this task. Their role as subject matter experts is critical in providing valuable input to the technical writer for incorporation into the documentation. Contracting manpower well versed in technical documentation preparation will substantially increase the likelihood to produce effective, "user-friendly" materials. Adding a project management perspective unrelated to the technical development process will greatly increase the chances for technology transfer as well as provide a cohesive stable environment for future development of the CAPS application. The "catch 22" of resolving this dilemna is that funding to outsource this effort is often scarce. In order to solicit funding, a well planned technology transfer strategy must be in place, which in turn is supported by easy to use, high quality documentation.

CAPS has proven, without a doubt, that it has the potential to solve the challenges that face DoD software engineers. Enormous benefits that can be realized from this technology include cost savings and on-time delivery of systems to the user. The key to transferring this technology to DoD and the commercial industry lies with the motivation of the PI, co-PI's, and R&D team where no project management professionals exist. The current CAPS team is enthusiastic and extremely encouraged by interest shown in the technology to date. This motivation, coupled with the implementation of the technology

transfer plan presented in this thesis should invoke increased interest and increased funding by DoD software engineering agencies. This ultimately will dictate its success within the commercial industry.

This valuable CAPS tool, wholly owned by the government, should be deployed immediately as a primary software development weapon in the DoD fight for software quality and efficiency.

VI. THE FUTURE DIRECTION

The CAPS R&D team continue its development to improve the product with CAPS 96 expected to be released by mid 1997. CAPS 96 enhancements include utilizing the object-oriented version of Ada (Ada95), an improved user interface, more extensive help facilities, and portability to a PC running the SOLARIS and Linux operating systems. The CAPS team is currently preparing to promote CAPS for a new Chief of Naval Operations initiative, the SMARTSHIP project, where CAPS can be used to establish design requirements and conduct feasibility studies. CAPS is also currently involved in the design of other important research such as the Sudden Infant Death Syndrome (SIDS) Wireless Acoustic Monitoring System (SWAMS) which detects the acoustic signature of an infant without the use of electrodes.

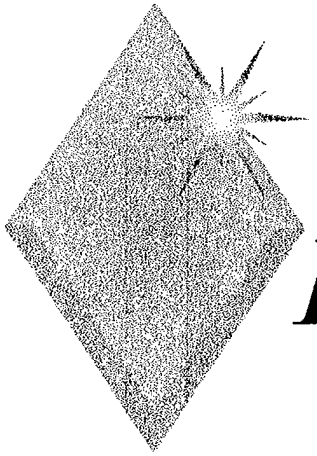
Long term development plans include work on extending CAPS to a distributed computing environment. Other long term plans focus on improved computer aid for managing and retrieving reusable software, and computer-aided generation and optimization of production quality implementation code from specifications.

The future demand for hard real-time systems within DoD and the commercial industry will continue its upward trend. The software engineering community must continue to counter challenges faced in the design and development of these software intensive systems with tools such as CAPS. An even greater importance is placed on proactively seeking to implement technology transfer at all levels between DoD, industry, and the international community. Results of technology transfer will continue to reduce software development costs and provide the best possible system that meets all the requirements of the user.

APPENDIX A: CAPS TECHNOLOGY TRANSFER PLAN

CAPS - Technology Transfer Preparation Plan			
	Action	**Resources Required**	**Responsible PI**
Phase 1 Innovation	Ongoing CAPS93 (release 1) distributed	ADA IC Clearinghouse	Luqi, Berzins, Shing
Phase 2 Planning	Collaborate with NPS ORTA	None	Luqi, Berzins, Shing
	Determine funding requirements for DUT and TT efforts	None	Luqi, Berzins, Shing
	Identify available resources. - In-House - DoD/commercial	None	Luqi, Berzins, Shing
Phase 3 Target Users	Identify potential DoD software development projects.	Internet/Publications/Journals ARL/NSWC/NOSC/ONR BAA	R&D Team, NPS ORTA, OTT
	Prepare/Update CAPS basic technical documentation. - Users Manual - Installation Manual - Tutorial - Quickstart	Thesis students/ outsource	Luqi, R&D Team
Phase 4 Marketing	Prepare generic CAPS briefs - Executive - Technical - Program Managers	Thesis students	R&D Team
	Prepare CAPS multimedia presentation CD-ROM.	Thesis students/ outsource	R&D Team
	Provide Internet access to papers, articles, theses, and briefings	Thesis students/ outsource	R&D Team
	Conduct Seminars/training	NPS facilities	Luqi, R&D Team
	Prepare/Distribute CAPS brochures	Thesis students	All
Phase 5 Application	Develop useable prototypes with DoD and commercial relevance - SWAMS - SMARTSHIP	Thesis students	R&D Team
Phase 6 Evaluate	Evaluate CAPS TT to DoD	None	R&D Team, NPS ORTA
	Initiate CRADA via NPS ORTA	Thesis students	Luqi, NPS ORTA

APPENDIX B : EXECUTIVE SUMMARY BRIEF



COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)

Executive Summary Brief

LT Robert Cooke

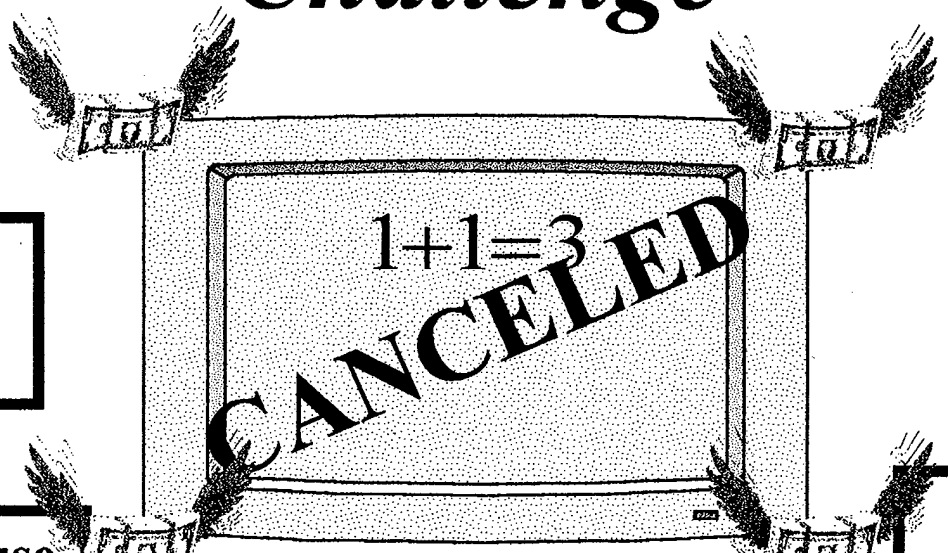
Naval Postgraduate School

wwwcaps.cs.nps.navy.mil

Overview

- **The DoD Software Engineering Challenge**
- **How DoD Meets the Challenge**
- **The CAPS Solution**
- **Future Direction**

The DoD Software Engineering Challenge



Requirements Analysis

Cost Overrun

Software Re-use

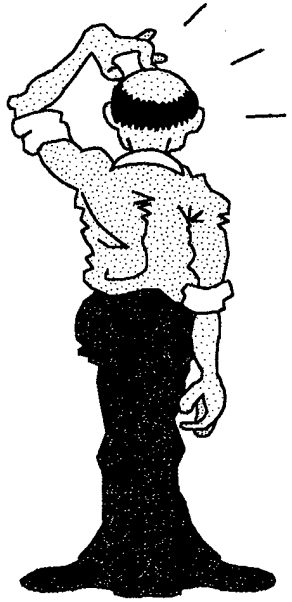
Critical Errors

Prototype

Failed Projects

Standards

Undelivered Systems



The DoD Software Engineering Challenge

- **DoD Acquires/Develops Large Complex Software Intensive Systems to Meet Mission Needs**

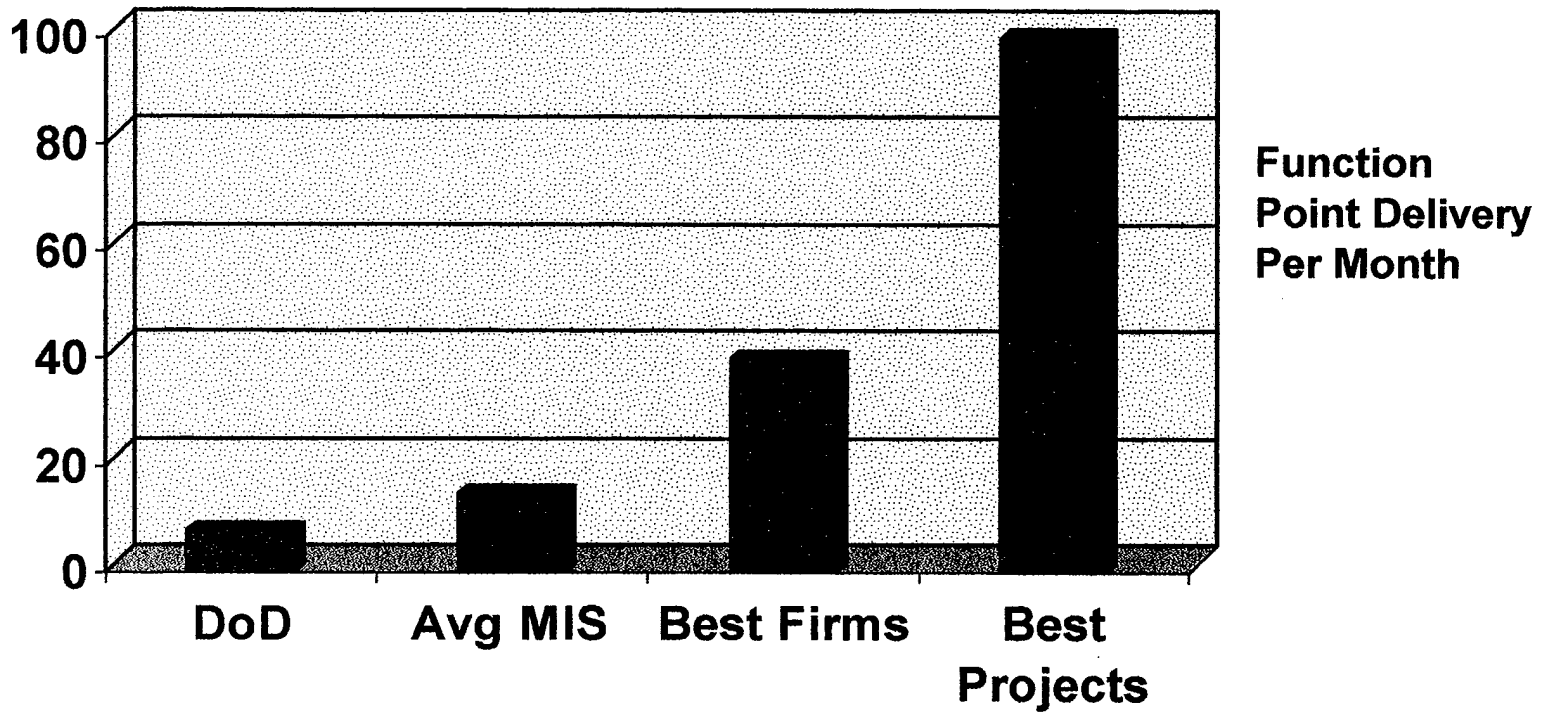
- **DoD Software Investment:**
  - **$30 Billion in FY 1990**
  - **$42 Billion in FY 1995**
  - **$83 Billion in FY 1995 on Failed Projects**

- **Software Technology Lags Far Behind Hardware Development**

- **Result: Most Large Software Projects Overbudget, Overtime, or Failure**

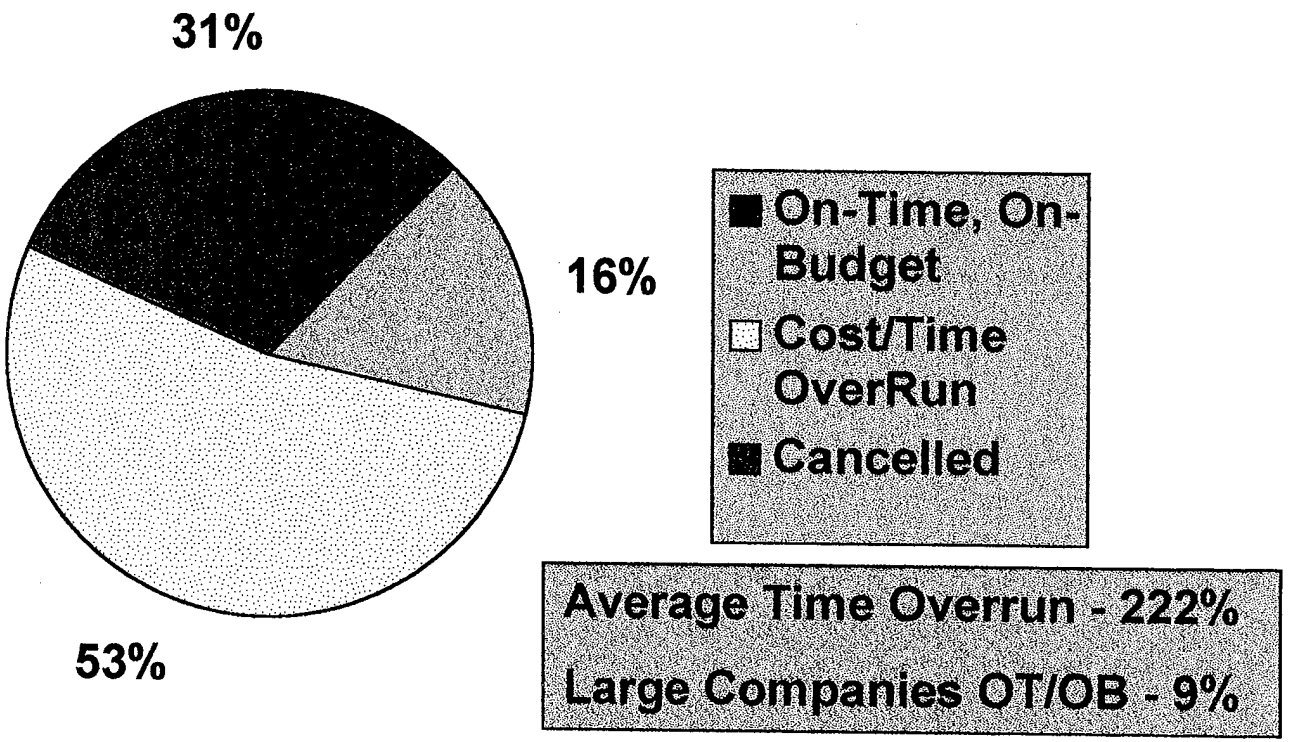
The DoD Software Engineering Challenge

Software Delivery Rates



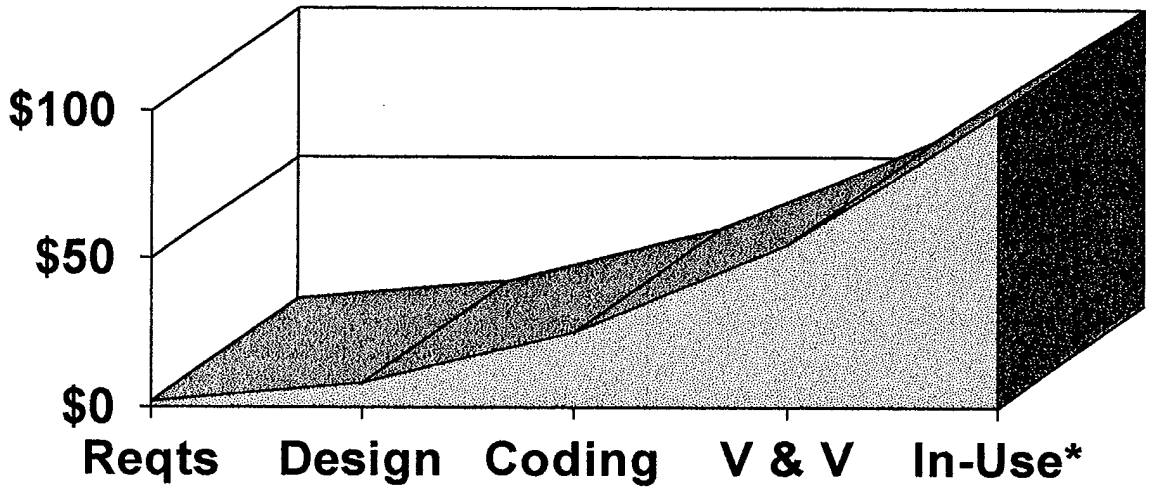
The DoD Software Engineering Challenge

Software Project Failure Rates



The DoD Software Engineering Challenge

Cost of Fixing Software Errors



*** Software Lifetime Maintenance Costs exceed 200% of Initial Development Costs**

The DoD Software Engineering Challenge

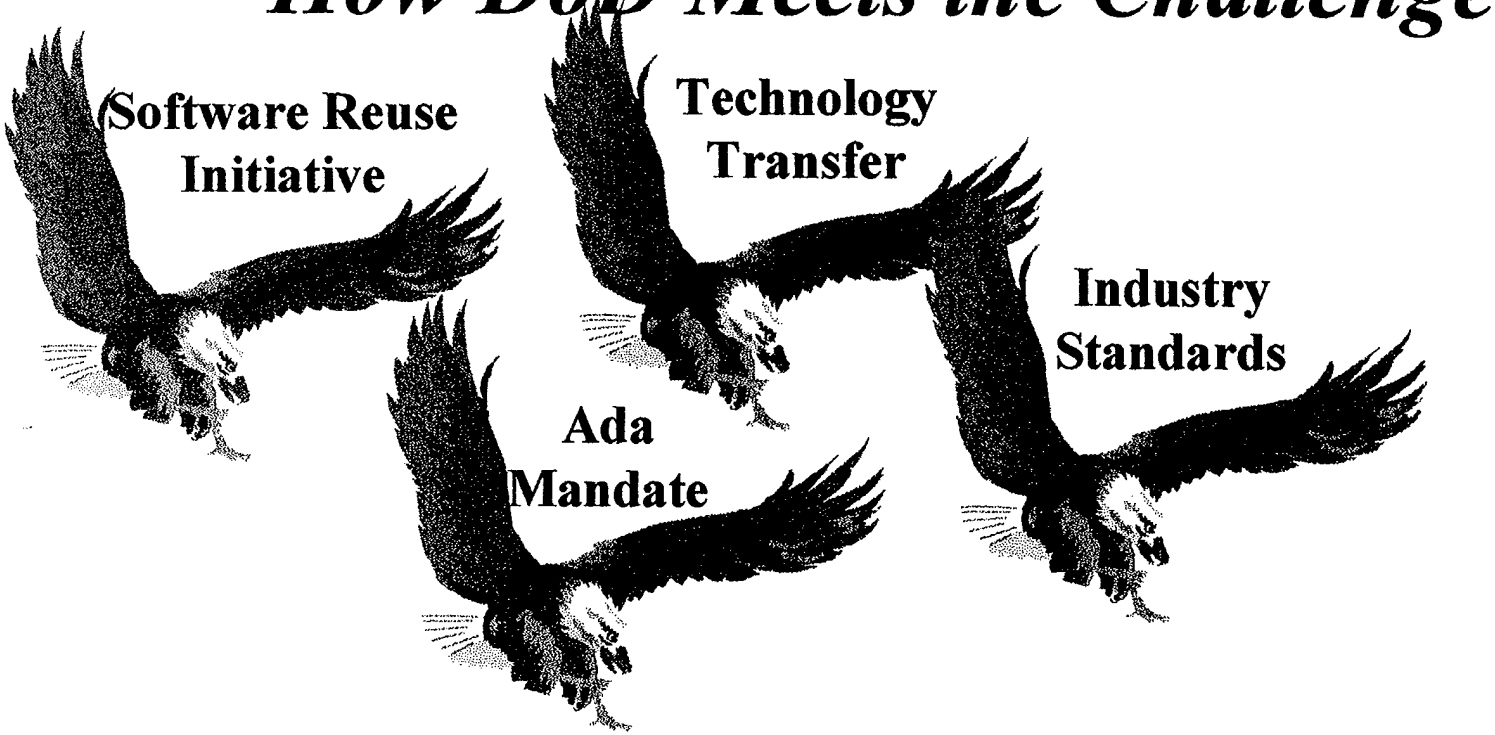
Requirements Errors Are Critical

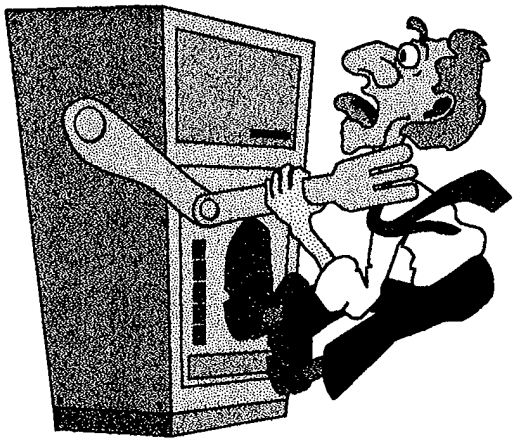
1992 Study of Voyager/Galileo Software

- **The Main Source of Catastrophic Failures: *Faults in Requirements Specification***

- **197 Significant or Catastrophic Errors**
  - ☹ **3 Coding Errors**
  - ☹ **194 *Specification Errors* of Functions & Interfaces**

How DoD Meets the Challenge





How DoD Meets the Challenge

Current Efforts

- **Industry Software Engineering "best practices" and Standards**
  - **Mil-Standard 498**
  - **ISO/IEC 12207**
- **Ada Mandate**
  - **Object Oriented - Ada 95**
- **Software Reuse Initiative (SRI)**

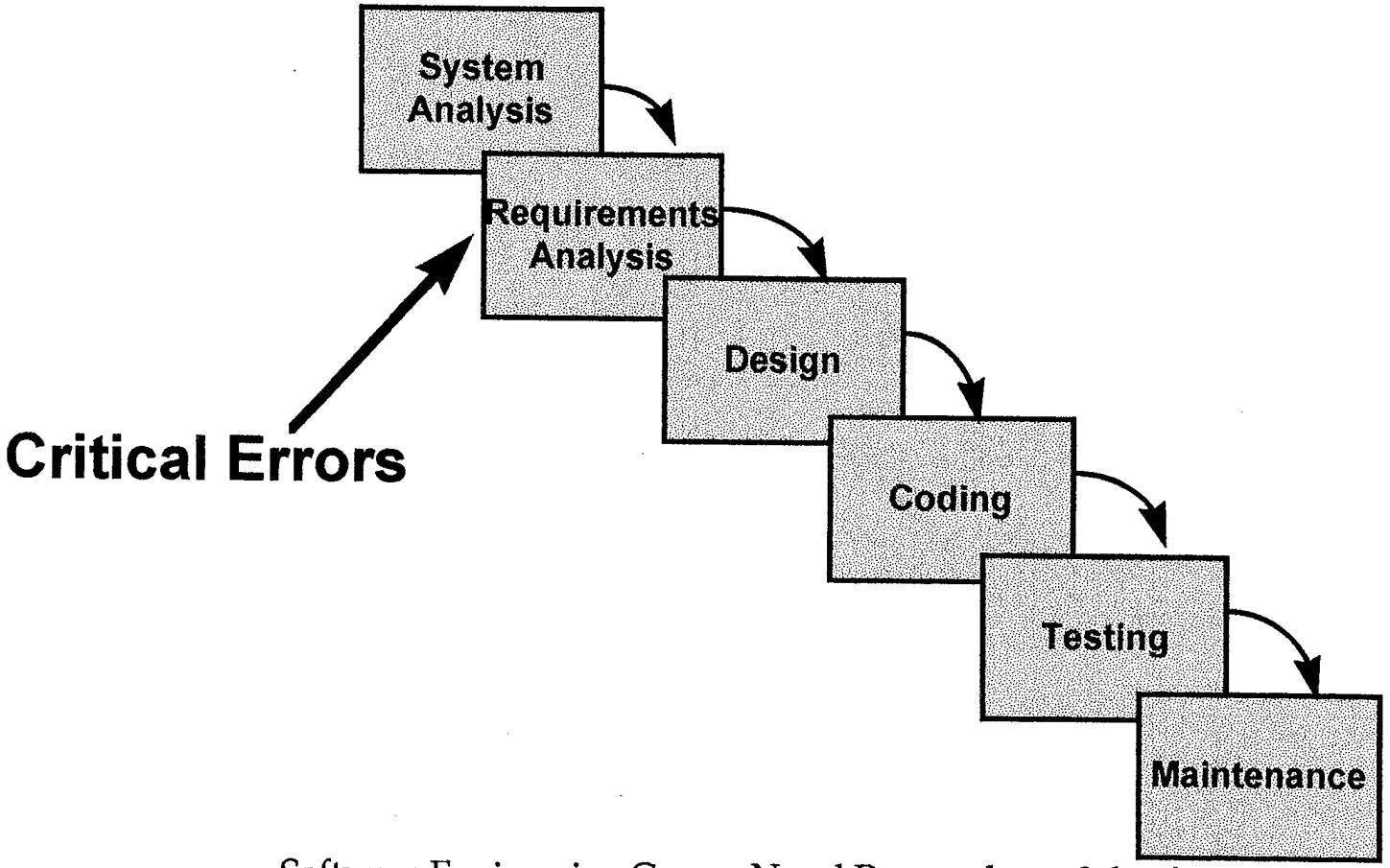
How DoD Meets the Challenge

Current Efforts

- **Acquisition Policy Reform**
  - **Max use of GOTS/COTS**
- **Innovative Teaching and Research**
  - **NPS, Georgia Tech, Texas Tech, ...**
  - **Service Academies**
  - **Technology Transfer**

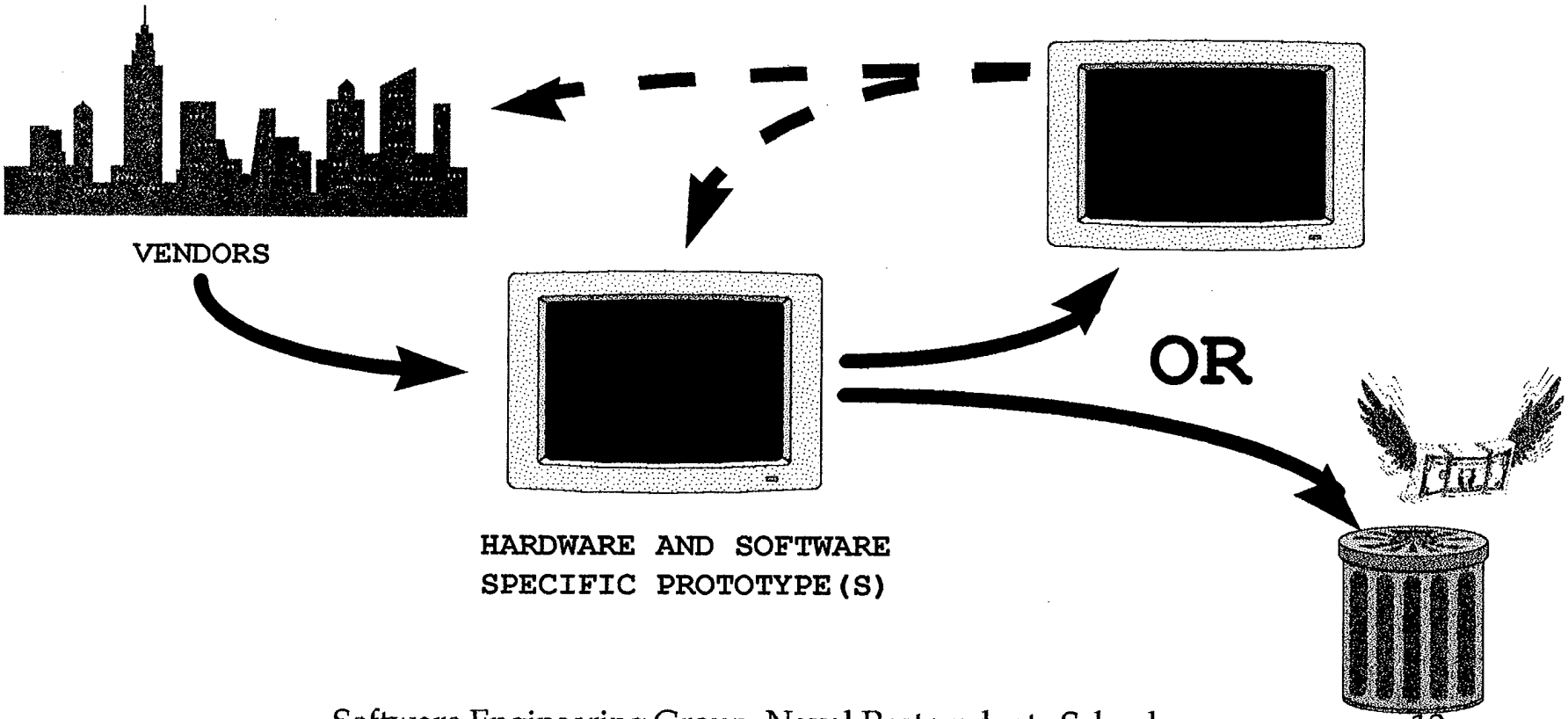
How DoD Meets the Challenge

Methodology -Traditional



How DoD Meets the Challenge

Methodology - Prototyping



The CAPS Solution

DoD efforts in right direction, but...

State-of-the-art Software Engineering Tools needed now to aid software engineers.

CAPS Technology...is here and now!!

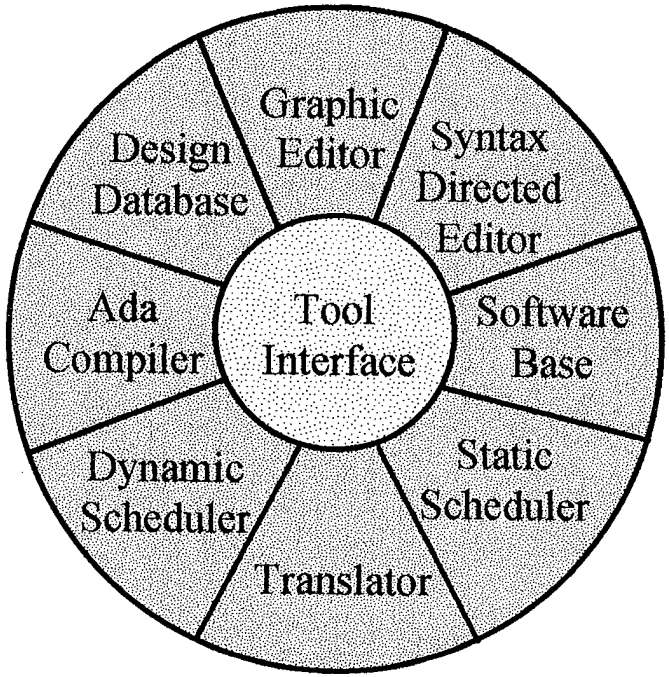
The CAPS Solution

What Is CAPS?

The "Computer-Aided Prototyping System" is an integrated set of software tools which use a fifth generation language for automated real-time software prototype development

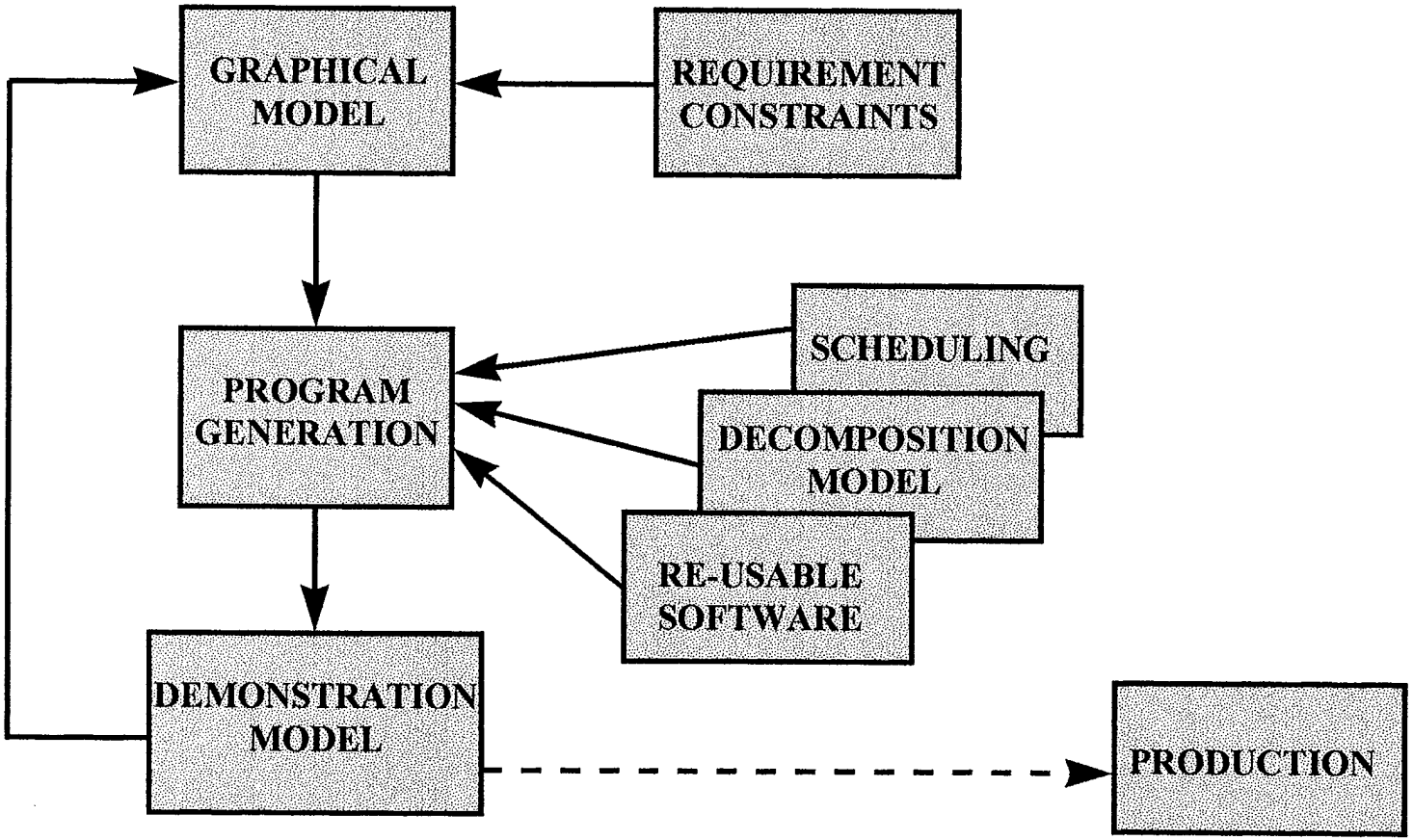
The CAPS Solution

- **Automates Software Development**
- **Improves Software Quality**
- **Reduces Development Time**
- **US Army's Software Master Plan**
- **DISA's Defense Software Repository - Free CD!**

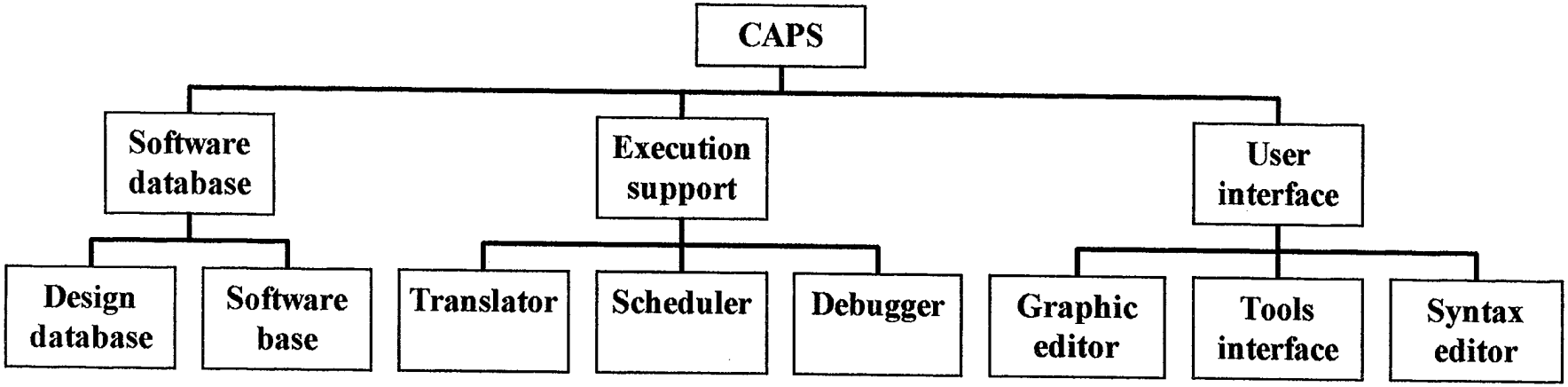


The CAPS Solution

How CAPS Works



The CAPS Solution

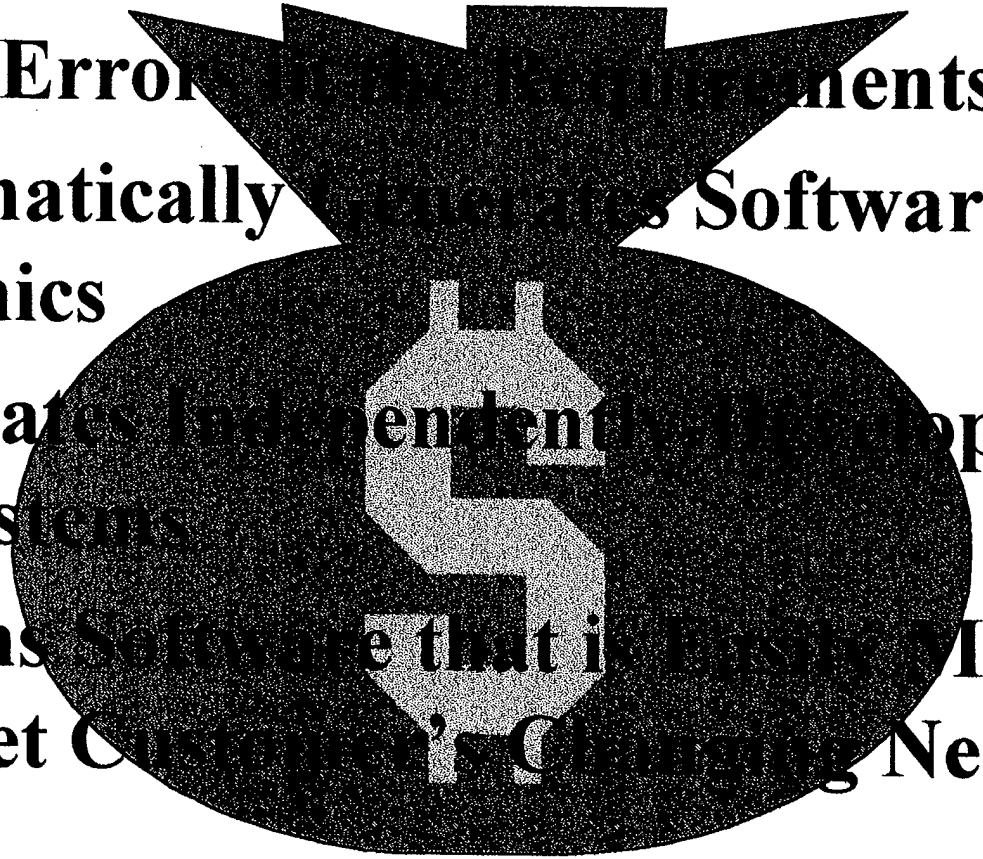


The CAPS Solution

Successful CAPS Demos

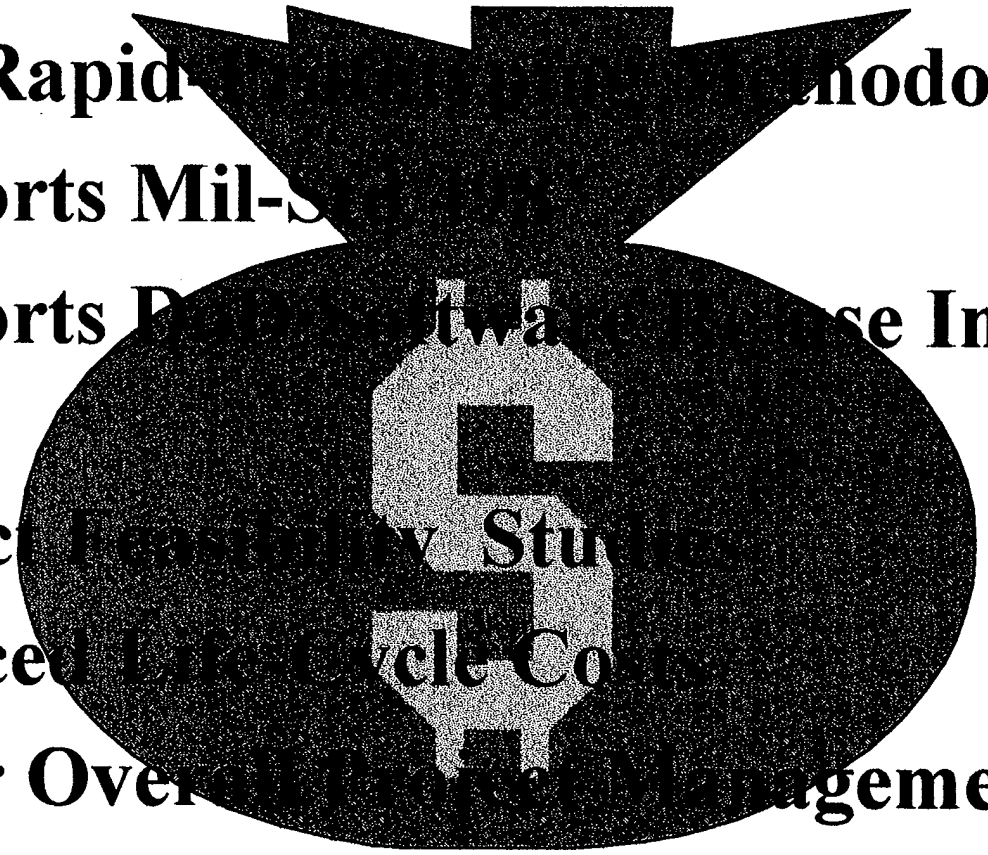
- **Generic C^3I Station**
- **Missile Defense**
- **ATACMS**
- **SWAMS**

CAPS Key Benefits



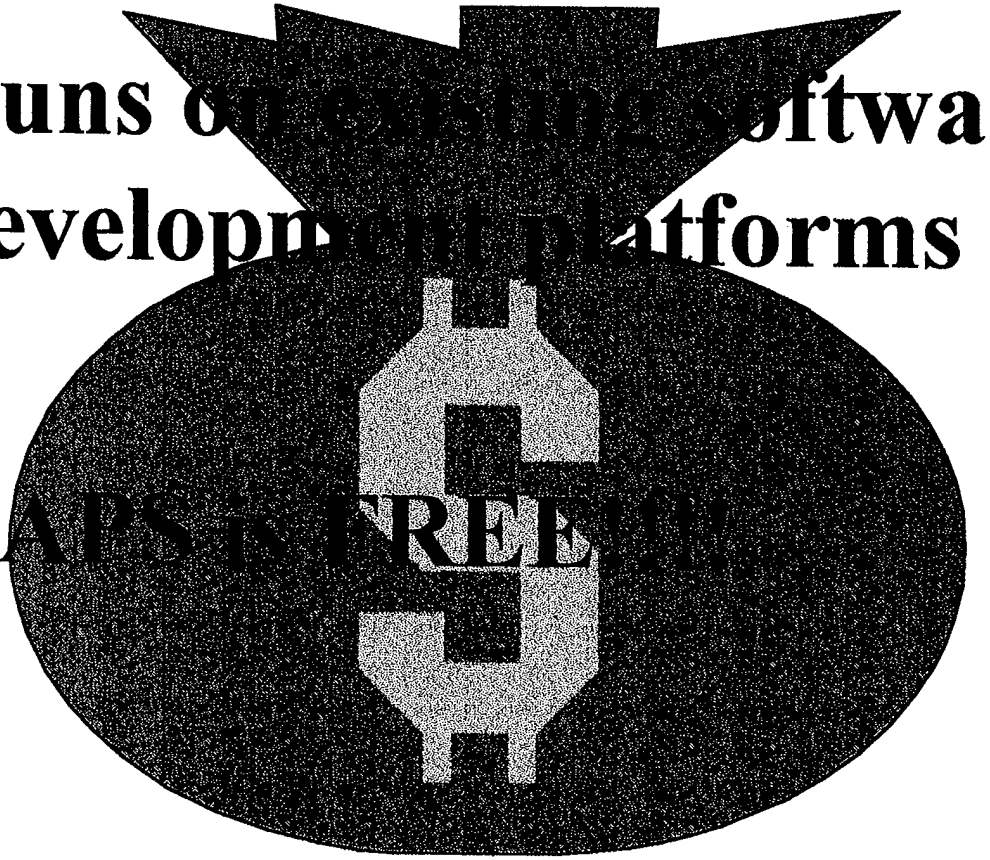
- **Finds Errors in the Requirements Phase**
- **Automatically Generates Software from Graphics**
- **Integrates Independently Developed Subsystems**
- **Designs Software that is Easily Modified to Meet Customer's Changing Needs**

CAPS Key Benefits



- **Uses Rapid- nodology**
- **Supports Mil-S**
- **Supports R tw se Initiative (SRI)**
- **Projec Stu**
- **Reduce cle Co**
- **Better Ove gement**

CAPS Key Benefits



- **Runs on existing software development platforms**
- **CAPS is FREE!!!**

NPS Software Engineering Group

- **8 Faculty Members**
- ***Presidential Young Investigator* Award**
- **Rated #1 in Universities, #3 Worldwide**
- **Created CAPS and S/E Technologies**
  - ⇨ **$2.5M R&D Effort ; 63 Thesis Projects**

Key Sponsors of CAPS Research

- NSF
- ARO
- ONR
- NOSC/NRAD



- DARPA
- AJPO
- AFOSR
- Industry

Future Directions

- **DoD Needs More Trained Personnel in Software Engineering Techniques**
  - ➪ **Increase S/E knowledge base via Training on CAPS**
- **DoD Large Software Systems Need Computer Aided Prototyping Systems**
  - ➪ **Commitment to use CAPS in DoD's Software Acquisition**

Future Directions

- **CAPS96 Enhancements due mid 1997**
  - ➪**Evolution Control System**
  - ➪**User Interface**
  - ➪**On-line Help**
  - ➪**On-line Documentation**
  - ➪**Portable to PC using Solaris and Linux OS**

Future Directions

- **Complete CAPS Technology Transfer to DoD and Commercial Industry**
  - ➪**CNO Initiative - SMARTSHIP**
  - ➪**Distributed Computing Networks - NSWC**
  - ➪**Robotics**
  - ➪**Avionics**
  - ➪**Manufacturing**

Where to get more information on CAPS

- **CAPS Home Page: http://wwwcaps.cs.nps.navy.mil**
  - **Tutorial**
  - **Reference Manual**
  - **Installation Manual**
  - **Reference Materials (books, articles, papers, theses)**
  - **Briefing Slides**
- **Free CAPS (Release 1) CD-ROM available at:**
  - **Defense Software Repository System: Walnut Creek CD-ROM**
  - **Ada Information Clearinghouse (AdaIC) Home Page: http://sw-eng.falls-church.va.us/AdaIC/source-code/caps**
- **Free CAPS Multimedia Presentation CD-ROM available Dec 96**

APPENDIX C: TECHNICAL/DEVELOPER BRIEF

Computer-Aided Prototyping System (CAPS)

Technical/Developer Brief

Software Engineering Group
Naval Postgraduate School
wwwcaps.cs.nps.navy.mil

Introduction

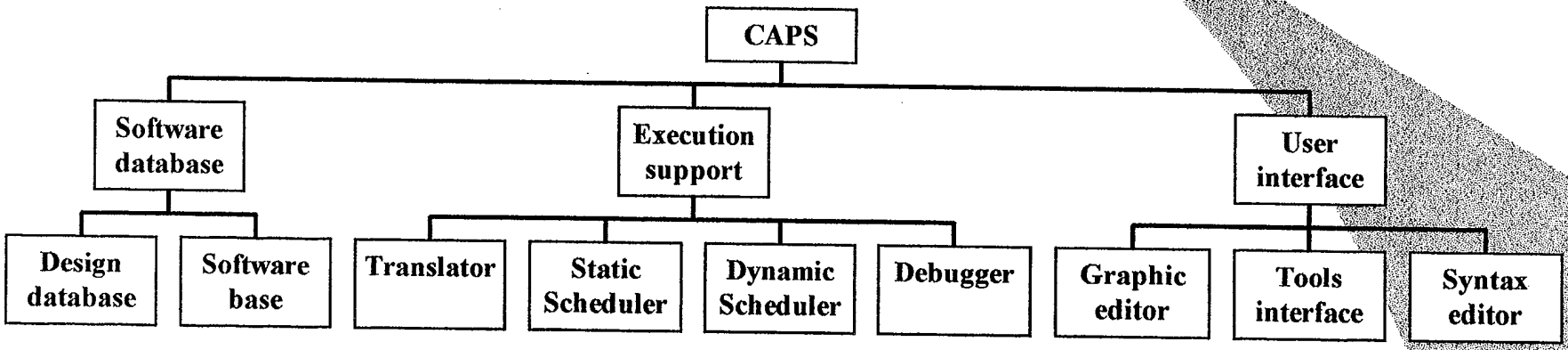
What Is CAPS?

The "Computer-Aided Prototyping System" is an integrated set of software tools which use a fifth generation language for automated real-time software prototype development"

Overview

- CAPS Components and Associated Tools
- Prototype System Description Language (PSDL)
- Where to Get More Information on CAPS

CAPS Components and Associated Tools



CAPS Components and Associated Tools

User Interface

- Supports concurrent tools
  - X-windows
  - TAE Plus
- Graphics editor
  - Automatically produces PSDL representation
- Graphical objects - operators/data streams
- Hides interface details from designer

CAPS Components and Associated Tools

Database

- Software Database
  - Tracks PSDL descriptions and Ada implementations
- Design Database
  - Manages coordination of team design efforts
- Holds reusable components
- Manages prototype configuration
- Version Control

CAPS Components and Associated Tools

Execution Support

- **Translator**
  - **Generates code that binds reusable components**
  - **Implements: control constraints, data streams, timers**
- **Static Scheduler**
  - **Allocates time slots for real-time constrained operators**
  - **Provides diagnostic scheduling information**
- **Dynamic Scheduler**
  - **Allocates time slots for non-time critical operators**
- **Debugger**
  - **Monitors timing constraints**
  - **Reports failure of runtime**
  - **Allows designer to adjust deadlines**

Prototype System Description Language (PSDL)

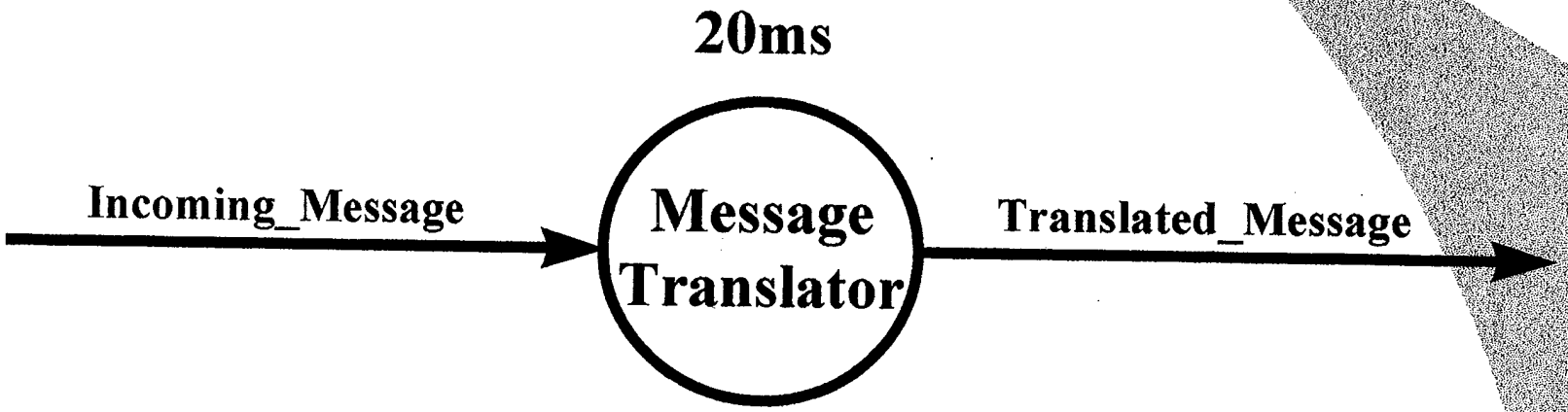
- Integrates tools in CAPS
- Provides uniform conceptual framework
- High level system description
- PSDL components are either operators or types

Prototype System Description Language (PSDL)

- PSDL computational model is an augmented graph
  - G=(V, E, T(v), C(v))
  - V= set of vertices
  - E = set of edges
  - T(v) = set of timing constraints
  - C(v) = set of control constraints

Prototype System Description Language (PSDL)

Sample PSDL Graph



Operators (circles) = vertices

Data Streams (Lines) = edges

Prototype System Description Language (PSDL)

- When an Operator *fires*, it:
  - Reads one data value from each input stream
  - Computes results if execution guard is satisfied
  - Writes at most one result value into each output stream (if output guard satisfied)

Prototype System Description Language (PSDL)

Structure

- PSDL component has two parts:
  - Specification
    - define interfaces
    - formal and informal descriptions
    - requirement traces
  - Implementation (two kinds)
    - Architecture descriptions
      - define decomposition of composite components
    - Code interface descriptions
      - define atomic components

Prototype System Description Language (PSDL)

Operators

- State machines
- Internal states are modeled by variable sets
- Empty variable set behaves like function
- Triggered by data streams or periodic timing constraints
  - Data stream = sporadic operator
  - Periodic timing constraints = periodic operator

Prototype System Description Language (PSDL)

Operators

- Two kinds of PSDL bubbles
  - Round bubbles
    - Represents part of proposed software
  - Rectangular bubbles
    - Represents simulation of external systems
    - Terminators
    - Not present in delivered system
    - MET = 0

Prototype System Description Language (PSDL)

Types

- Defines abstract data types
  - Introduces a new type name
    - Interfaces to a user-defined data structure
    - Data is private to implementation
    - New type used in stream declarations
  - Introduces a set of operations on the new type
    - Each operation is a PSDL operator
    - Instances of each operator appear as bubbles in PSDL dataflow diagram

Prototype System Description Language (PSDL)

Predefined Types

- Boolean
- Character
- String
- Integer
- Real
- Set
- Sequence
- Map
- Tuple
- One_of
- Relation

Prototype System Description Language (PSDL)

Data Streams

- Transmits data values between operators
- Data values = instances of abstract data type
- Can be :
  - Dataflow streams
  - Sampled streams

Prototype System Description Language (PSDL)

Dataflow Streams

- **Dataflow streams**
  - **Act as FIFO buffers**
  - **Models discrete transactions**
  - **Data values cannot be lost or replicated**
  - **Execution rate of producer and consumer must match**
  - **TRIGGERED BY ALL x, y implies x, y are data streams**
  - **Implies consumers never read an empty dataflow stream**
  - **Ensures each value in stream read once**

Prototype System Description Language (PSDL)

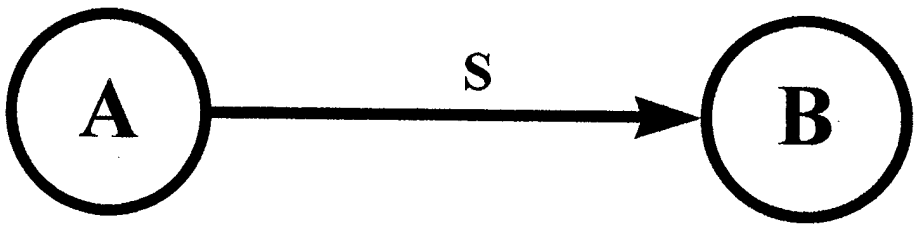
Sampled Streams

- Act as atomic memory cells
- Connects operators firing at uncoordinated rates
- Model data sources
- Data always available
- Data lost if consumer is faster than producer
- Connect producers and consumers of different periods
- Absence of TRIGGERED BY ALL z control constraint implies the stream z is sampled

Prototype System Description Language (PSDL)

Implied Precedence Constraints

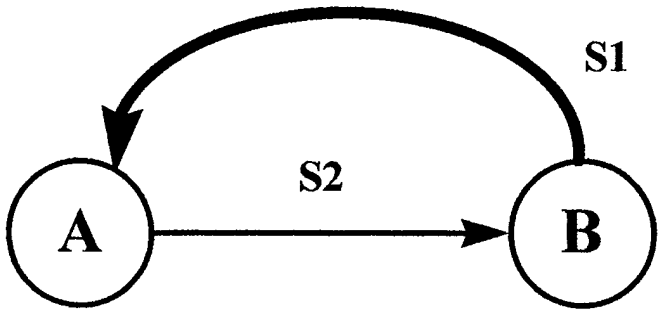
- If operator A produces a stream s that is consumed by operator B, then A must be scheduled to fire before B
- Apply only if A and B are both time critical, or if neither one is
- Affect *corresponding firings* of A and B



Prototype System Description Language (PSDL)

State Streams

- **An operator is a *state machine* if it has one or more *state streams***
  - **State streams declared in specs of parent operator**
  - **Declaration must supply initial value for the stream**
  - **Can be either dataflow or sampled**
  - **Dataflow diagram of composite state machine operator has cycles**
  - **No implicit precedence constraints**
  - **No hazard of reading uninitialized data**



Prototype System Description Language (PSDL)

Stream Types

- PSDL streams associated with data type declarations
- Streams carry only values of the declared type
  - Special data type: *exception*
    - Propagates along data streams of type exception
    - Exception handling operators have input streams of type exception
    - Input streams of type exception used in execution guards

Prototype System Description Language (PSDL)

Timers

- A software stopwatch
- Declared in implementation part of composite operator
- Values used in control constraints
- Continuously updated to record real-time
- Value remains the same when stopped
- Resets to zero
- Time expressions have units
  - ms, sec, min, hour

Prototype System Description Language (PSDL)

Control Constraints

- Adapt reusable code to designs
- Conditional execution and output
- Control exceptions and timers
- Triggering conditions
  - Discards input data if condition not satisfied
- Output guards
  - Prevents writing of output data into stream if condition not satisfied

Prototype System Description Language (PSDL)

Timing Constraints

- **Max Execution Time (MET) = longest time between beginning and completion of execution**
- **Minimum Calling Period (MCP) = min time between two successive activations**
- **Max Response Time (MRT) = longest time between operator read and write**

Prototype System Description Language (PSDL)

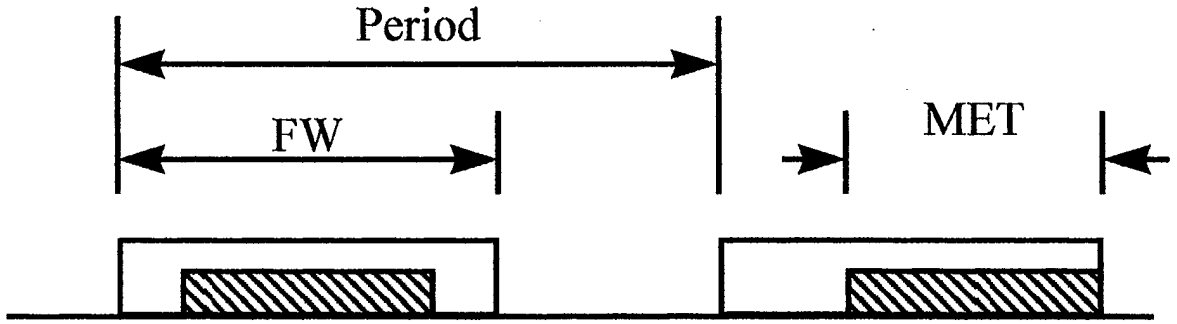
Timing Constraints

- Operator is *time-critical* if bubble has Max Execution Time (MET) annotation
- Only time critical operations can have timing constraints
- Two kinds:
  - Periodic
  - Sporadic

Prototype System Description Language (PSDL)

Constraints for Periodic Operators

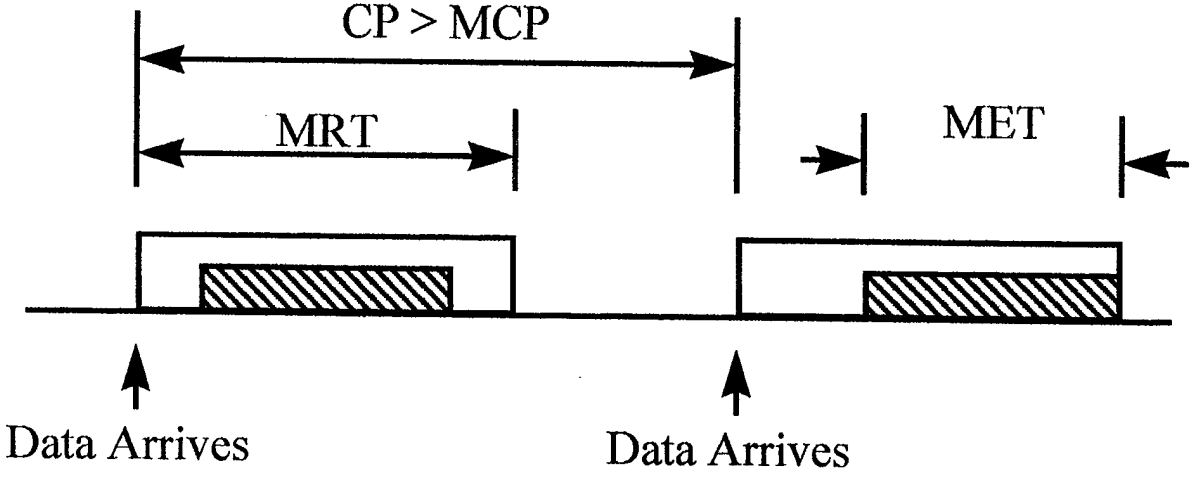
- **Period**
  - **Interval between consecutive triggering events**
- **Finish within**
  - **Upper bound between each triggering event and completion of firing**
  - **Equal to the period if not specified**



Prototype System Description Language (PSDL)

Constraints for Sporadic Operators

- **Minimum Calling Period (MCP)**
  - **Lower bound on time between consecutive input data arrival**
  - **Equal to MRT - MET if not specified**
- **Maximum Response Time (MRT)**
  - **Upper bound on time between input data arrival and completion firing**



Prototype System Description Language (PSDL)

Latencies

- Lower bound on the time from when data is written into a stream to the time when data can be read from stream
  - Models slow networks/telcom links
  - Records restrictions from external constraints
  - Annotation on data streams in dataflow diagrams

Prototype System Description Language (PSDL)

Example Prototype Specification

Hyperthermia Example

OPERATOR

brain_tumor_treatment_system

SPECIFICATION

STATES temperature: real

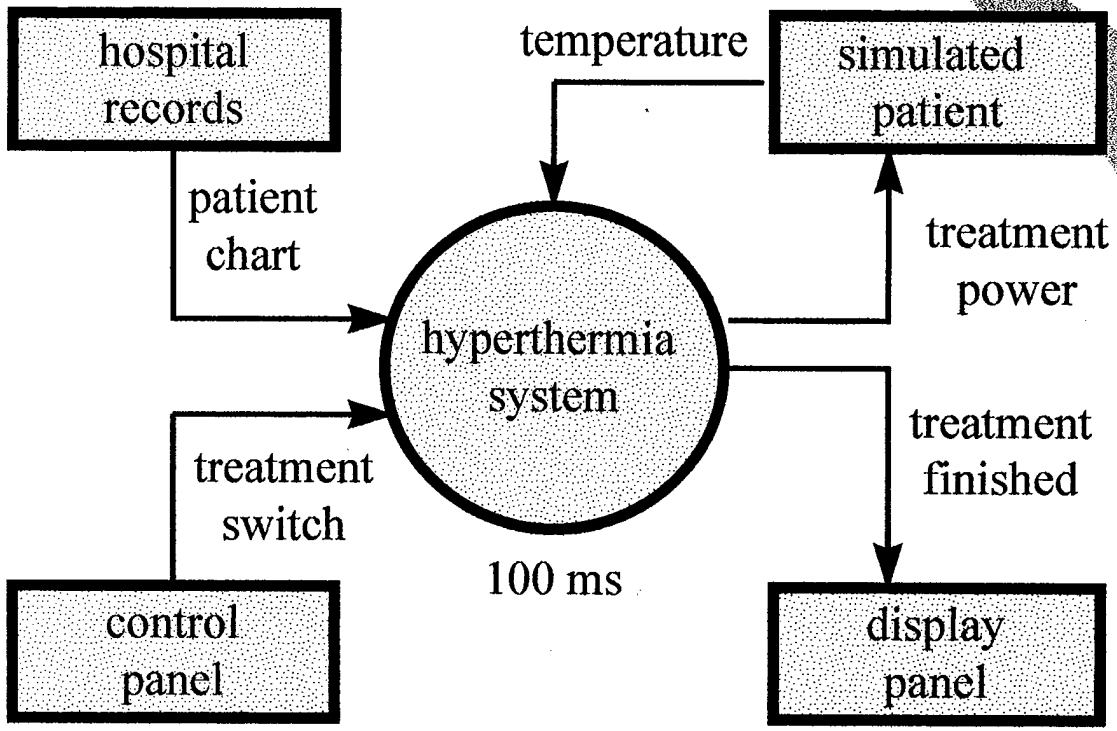
INITIALLY 37.0

DESCRIPTION { The brain tumor treatment system kills tumor cells by means of hyperthermia induced by microwaves. }

END

Prototype System Description Language (PSDL)

Example Prototype Implementation



Prototype System Description Language (PSDL)

Example Prototype Implementation

DATA STREAM
treatment_power: real,
treatment_switch: boolean,
treatment_finished: boolean,
patient_chart: medical_history

CONTROL CONSTRAINTS
OPERATOR simulated_patient
PERIOD 200ms
OPERATOR
hyperthermia_system
PERIOD 200ms
END

Where to get more information

- CAPS Home Page: http://wwwcaps.cs.nps.navy.mil
  - Tutorial
  - Reference Manual
  - Installation Manual
  - Reference Materials (books, articles, papers, theses)
  - Briefing Slides

- Free CAPS93 (Release 1) CD-ROM available at:
  - Defense Software Repository System : Walnut Creek CD-ROM
  - Ada Information Clearinghouse (AdaIC) Home Page: http://sw-eng.falls-church.va.us/AdaIC/source-code/caps

- CAPS Multimedia Presentation CD-ROM available Dec 96

APPENDIX D: PROGRAM MANAGERS BRIEF

Computer-Aided Prototyping System (CAPS)

Program Managers Brief

Software Engineering Group
Naval Postgraduate School
wwwcaps.cs.nps.navy.mil

Introduction

What Is CAPS?

The "Computer-Aided Prototyping System" is an integrated set of software tools which use a fifth generation language for automated real-time software prototype development"

Overview

- Prototyping Concepts
- The CAPS Solution
- CAPS Components
- CAPS Benefits
- Future Direction
- Where to Get More Information on CAPS

Prototyping Concepts

Purpose of Software Prototyping

- Evaluates accuracy of problem formulation
- Explores range of possible solutions
- Determines required interactions between proposed system and its environment

Prototyping Concepts

Benefits of Software Prototyping

- **Improves communication**
  - **Exposes unstated assumptions**
  - **Triggers requirements changes earlier in the process**
- **Reduces risk**
  - **Communication more certain**
  - **Properties of proposed design more certain**
- **Validates specifications**
  - **Specifications interpreted the same way**

Prototyping Concepts

Software vs. System Prototyping

- **Many systems contain embedded SW**
- **SW requirements derived from system requirements**
- **System and SW must be prototyped to resolve resource allocation and system performance issues**

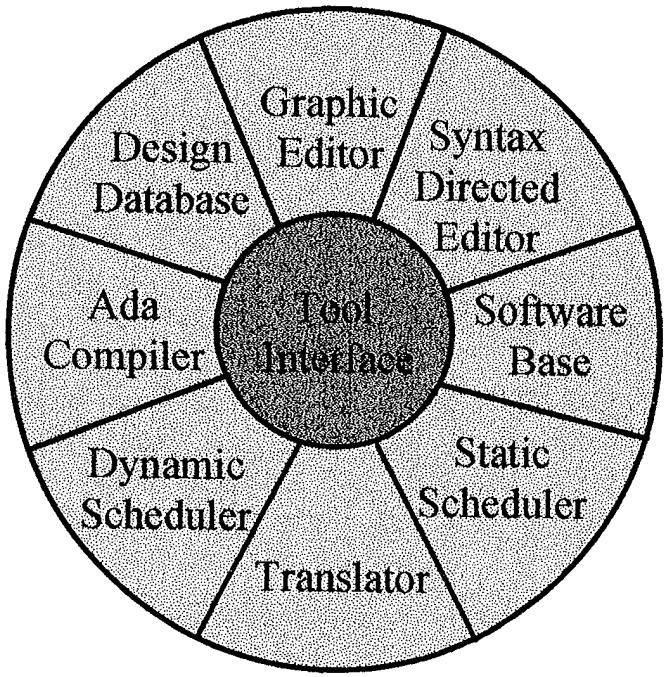
Prototyping Concepts

Properties of a Software Prototype

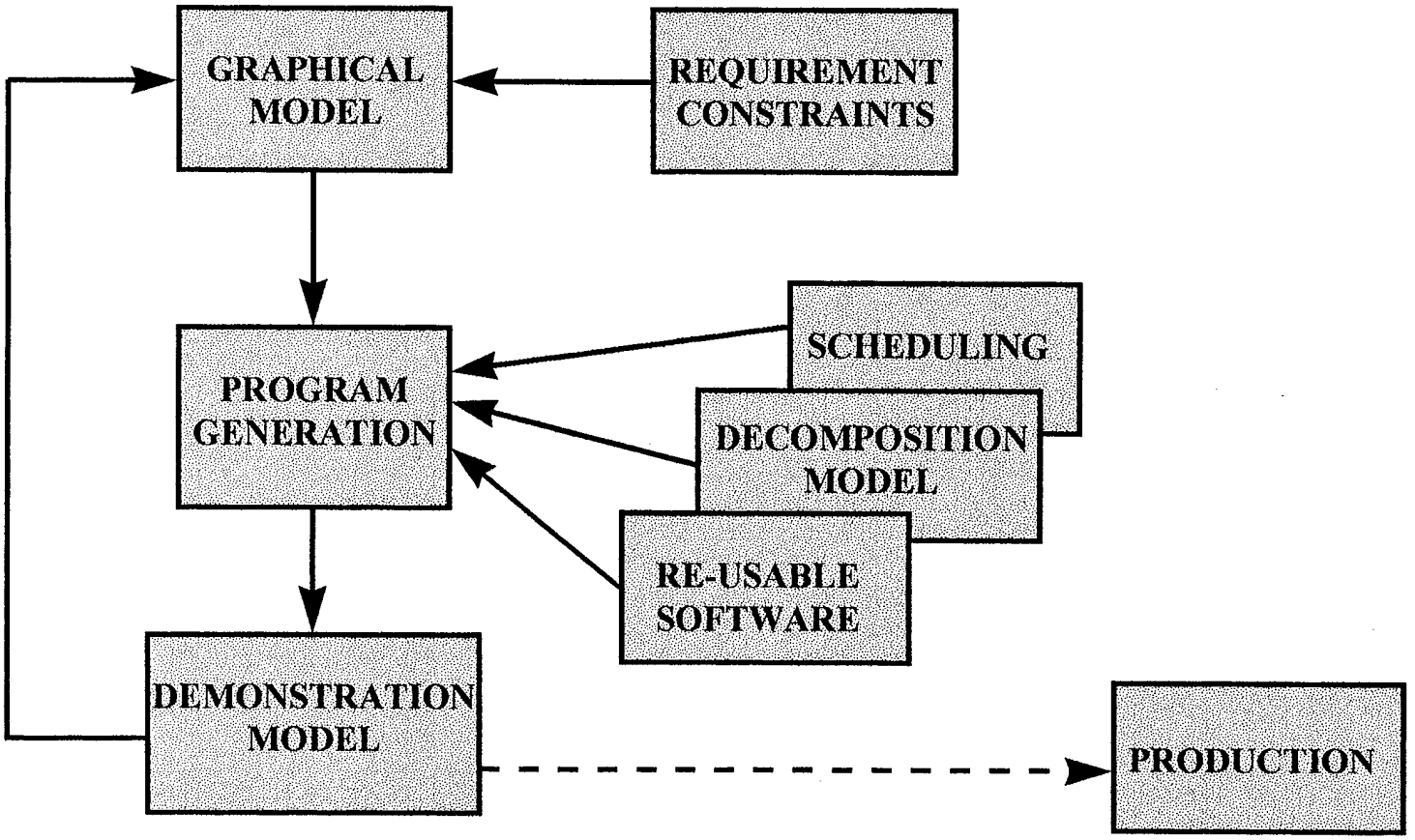
- **Executable model of target system**
- **Guides design decisions**
- **SW functions accurately evaluate uncertainty/risk issues**
- **Models the HW resources and SW architecture**
- **Runs in scaled real-time to determine timing relative to target HW**
- **Supports cost/benefit trade-off studies**

The CAPS Solution

- **Automates Software Development**
- **Improves Software Quality**
- **Reduces Development Time**
- **US Army's Software Master Plan**
- **DISA's Defense Software Repository - Free CD!**



The CAPS Solution



The CAPS Solution

Successful CAPS Demos

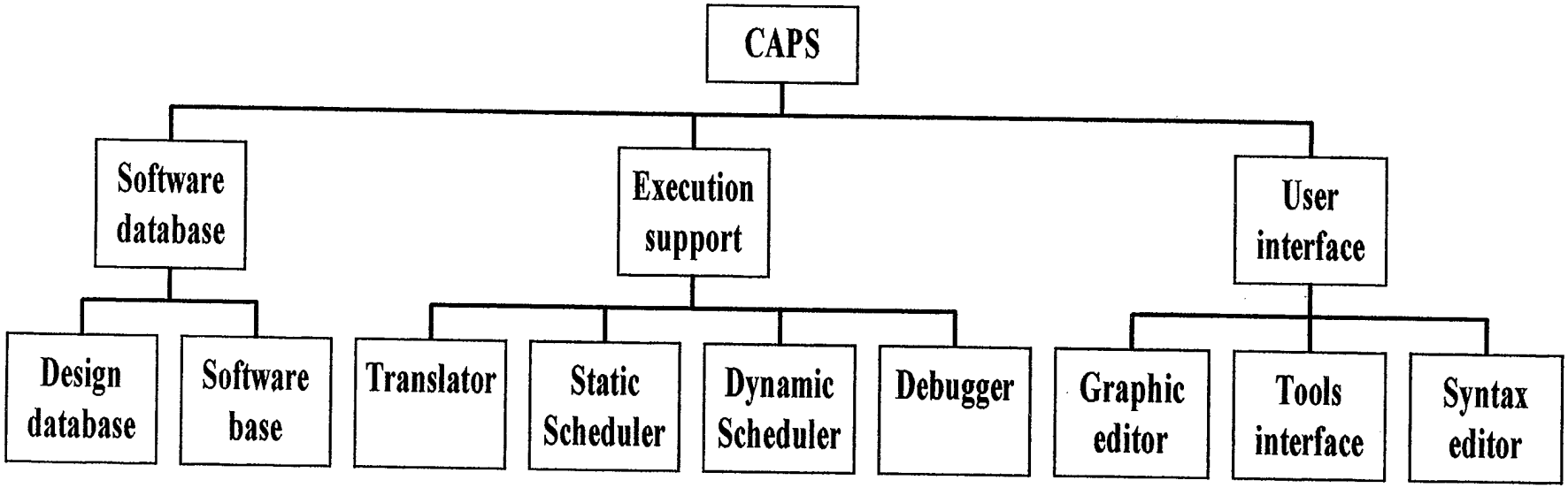
- **Generic C^3I Station**
- **Missile Defense**
- **ATACMS**
- **SWAMS**

The CAPS Solution

Potential CAPS Projects and Uses

- **SMARTSHIP**
- **Avionics**
- **Robotics**
- **Telecommunications**
- **Distributed Computing Networks**

CAPS Components and Associated Tools



CAPS Components and Associated Tools

User Interface

- **Supports concurrent tools**
  - **X-windows**
  - **Motif**
  - **TAE Plus**
- **Graphics editor**
  - **Automatically produces PSDL representation**
- **Graphical objects**
- **Hides interface details from designer**

CAPS Components and Associated Tools

Database

- **Software Database**
  - **Tracks PSDL descriptions and Ada implementations**
- **Design Database**
  - **Manages coordination of team design efforts**
- **Holds reusable components**
- **Manages prototype configuration**
- **Version Control**

CAPS Components and Associated Tools

Execution Support

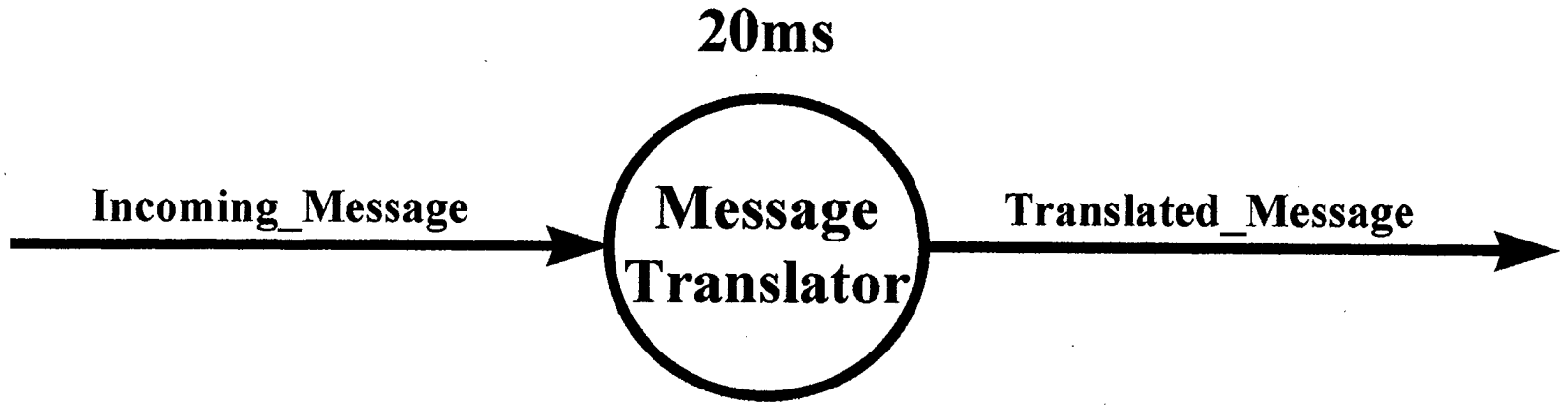
- **Translator**
  - Generates code that binds reusable components
  - Implements: control constraints, data streams, timers
- **Static Scheduler**
  - Allocates time slots for real-time constrained operators
  - Provides diagnostic scheduling information
- **Dynamic Scheduler**
  - Allocates time slots for non-time critical operators
- **Debugger**
  - Monitors timing constraints
  - Reports failure of runtime
  - Allows designer to adjust deadlines

Prototype System Description Language (PSDL)

- **Integrates tools in CAPS**
- **Provides uniform conceptual framework**
- **High level system description**
- **Augmented computation graphs**

Prototype System Description Language (PSDL)

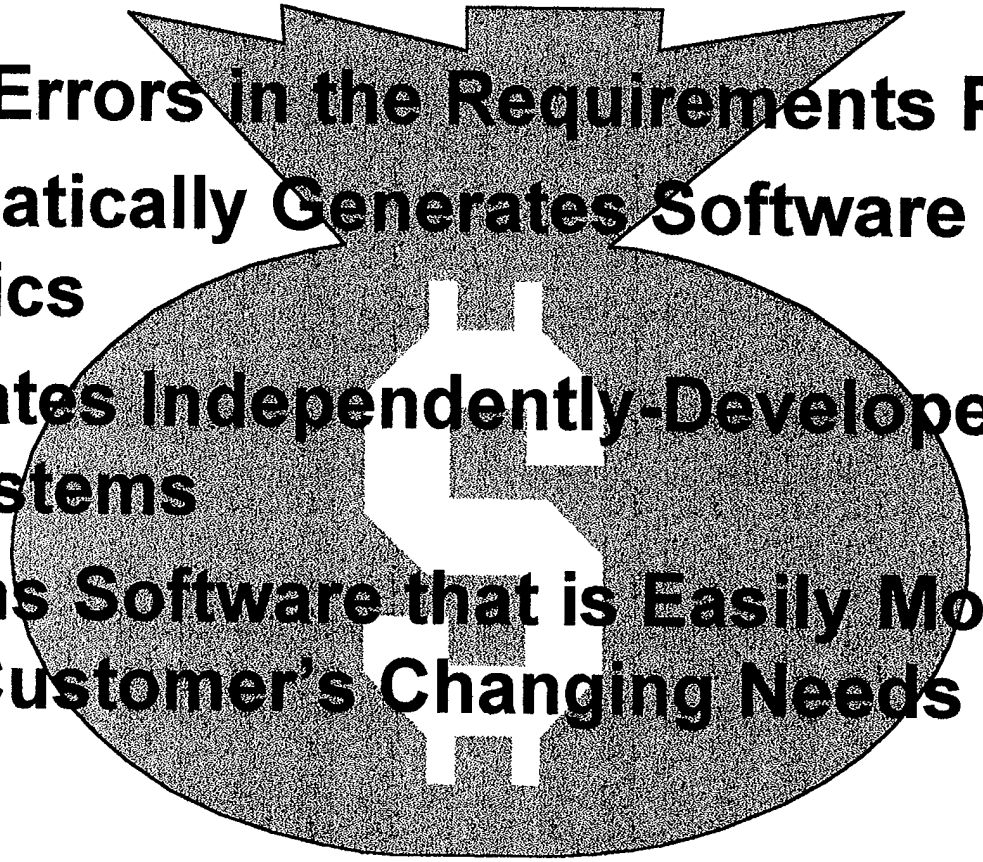
Sample PSDL Graph



Operators (circles) = vertices

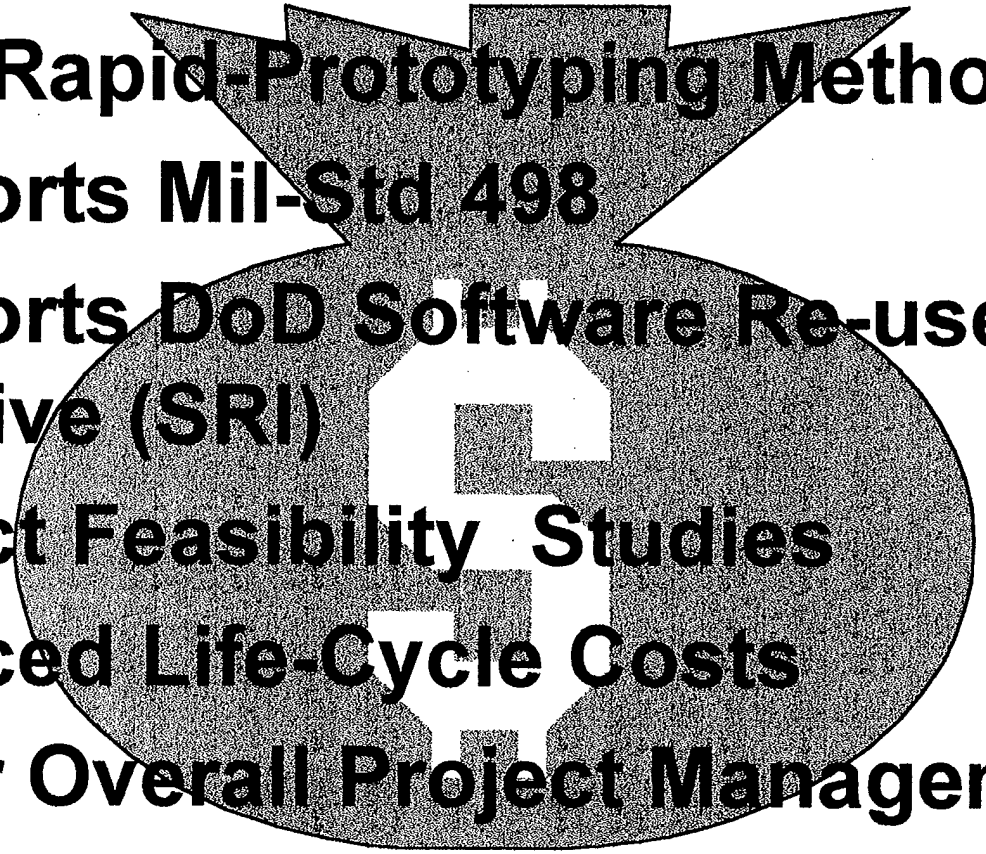
Data Streams (Lines) = edges

CAPS Key Benefits



- Finds Errors in the Requirements Phase
- Automatically Generates Software from Graphics
- Integrates Independently-Developed Subsystems
- Designs Software that is Easily Modified to Meet Customer's Changing Needs

CAPS Key Benefits



- Uses Rapid-Prototyping Methodology
- Supports Mil-Std 498
- Supports DoD Software Re-use Initiative (SRI)
- Project Feasibility Studies
- Reduced Life-Cycle Costs
- Better Overall Project Management

CAPS Key Benefits



- **Runs on existing software development platforms**
- **CAPS is FREE!!!!**

Future Directions

- **CAPS96 Enhancements due mid 1997**
  - **Evolution Control System**
  - **User Interface**
  - **On-line Help**
  - **On-line Documentation**
  - **Portable to PC using Solaris and Linux OS**

Where to get more information

- CAPS Home Page: http://wwwcaps.cs.nps.navy.mil
  - Tutorial
  - Reference Manual
  - Installation Manual
  - Reference Materials (books, articles, papers, theses)
  - Briefing Slides

- Free CAPS93 (Release 1) CD-ROM available at:
  - Defense Software Repository System : Walnut Creek CD-ROM
  - Ada Information Clearinghouse (AdaIC) Home Page: http://sw-eng.falls-church.va.us/AdaIC/source-code/caps

- Free CAPS Multimedia Presentation CD-ROM available Dec 96

APPENDIX E: CAPS TUTORIAL

Table of Contents

I. Introduction 1
A. Overview of CAPS 1
B. Organization of Chapters 2
C. The Prototyping Process 3
D. CAPS Tools 5
1. The PSDL Editor 5
2. The Text Editor 5
3. The Interface Editor 6
4. The Requirements Editor 6
5. The Change Request Editor 6
6. The Translator 6
7. The Scheduler 6
8. The Compiler 7
9. The Evolution Control System 7
10. The Merger 7
11. The Software Base 7
II. Getting Started 8
A. Invoking CAPS 8
1. The Designer Mode 8
2. The Manager Mode 8
3. Choosing a Design Database 9
B. A Simple and Complete CAPS Example 9
1. Create and Edit the Prototype 9
2. Translate and Schedule the Prototype 12
3. Implement Ada Modules for Atomic Operators 13
4. Compile the Prototype 13
5. Execute the Prototype 13
III. An Introduction to PSDL 14
A. Overview 14
1. Operators 14
2. Data Streams 14
3. Types 15
B. Timing Constraints 15
1. Periodic Operators 16
2. Sporadic Operators 16
C. "BY ALL" Triggers 17
D. "BY SOME" Triggers 17
E. Execution Guards 18
F. Conditional Output 18
G. State Variables 18
H. Exceptions 18
I. Timers 18
IV. PSDL Editing 19
A. Overview 19
B. A Quick Review of PSDL Operators and Data Streams 19
C. Graphic Editor and Syntax Directed Editor Interactions 20

1. Graphic Editor / User Interaction 21
a. Drawing Streams 24
b. Making External Streams 24
c. Moving Objects 24
d. Deleting and Renaming Operators 24
e. Undeleting Operators 25
f. Decomposing Operators 25
2. Syntax Directed Editor / User Interaction 26
a. The "CAPS-Cmds" Pull-Down Menu 28
b. Traversing the PSDL Parse Tree 29
c. Warnings, Errors and Alerts in the Syntax Directed Editor 30
D. The Complete PSDL Program 32
V. PSDL Translation 33
A. Overview 33
B. Translator / User Interaction 34
C. Results of Translation 36
VI. Prototype Scheduling 37
A. Overview 37
B. The Static and Dynamic Schedules 38
C. Scheduling Feasibility Factors 38
D. Decomposition Considerations 43
1. Periods 43
2. Maximum Execution Times 43
3. Other Timing Constraints 43
E. Schedule Length 44
F. Equivalent Periods 46
G. Results of Scheduling 46
VII. Interface Integration 47
A. Overview 47
B. Text Interface 47
C. Graphic Interface 48
1. Time-Critical Operator Approach 48
2. High Priority Task Approach 55
D. Summary 55
VIII. Prototype Execution and Diagnosing Errors 56
A. Overview 56
B. Timing Errors 56
C. CAPS CPU Speed Ratio 57
D. Buffer Overflow and Underflow 57
E. Prototype Modification 58
IX. Summary 59
X. References 60
XI. CAPS Glossary 61
XII. Appendix A 67
XIII. Appendix B 84
XIV. Appendix C 88
XV. Appendix D 89
XVI. Appendix E 90

Disclaimer

CAPS is an ongoing research project that consists of many individual research efforts. Consequently, CAPS is a dynamic and diverse system. Many CAPS components have been developed in near isolation and incorporated into the system as a whole. Every reasonable effort has been made to make CAPS robust and user-friendly, however, due to the very nature of its development environment, it is not perfect. There are many aspects of CAPS which could be ergonomically, semantically, procedurally or otherwise improved. The CAPS development team is well aware of this. Our intent has not been to create an industrial strength product, but rather, to put to the test (and to the silicon), ideas, concepts and philosophies that have come from an academic environment.

I. Introduction

A. Overview of CAPS

The Computer-Aided Prototyping System (CAPS) [LK88] is a software engineering tool for developing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL) [LBY88], which provides facilities for modeling timing and control constraints within a software system. CAPS is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface. CAPS provides the following kinds of support to the prototype designer:

1) timing feasibility checking via the scheduler,
2) consistency checking and some automated assistance for project planning, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
3) design completion via the editors, and
4) computer-aided software reuse via the software base.

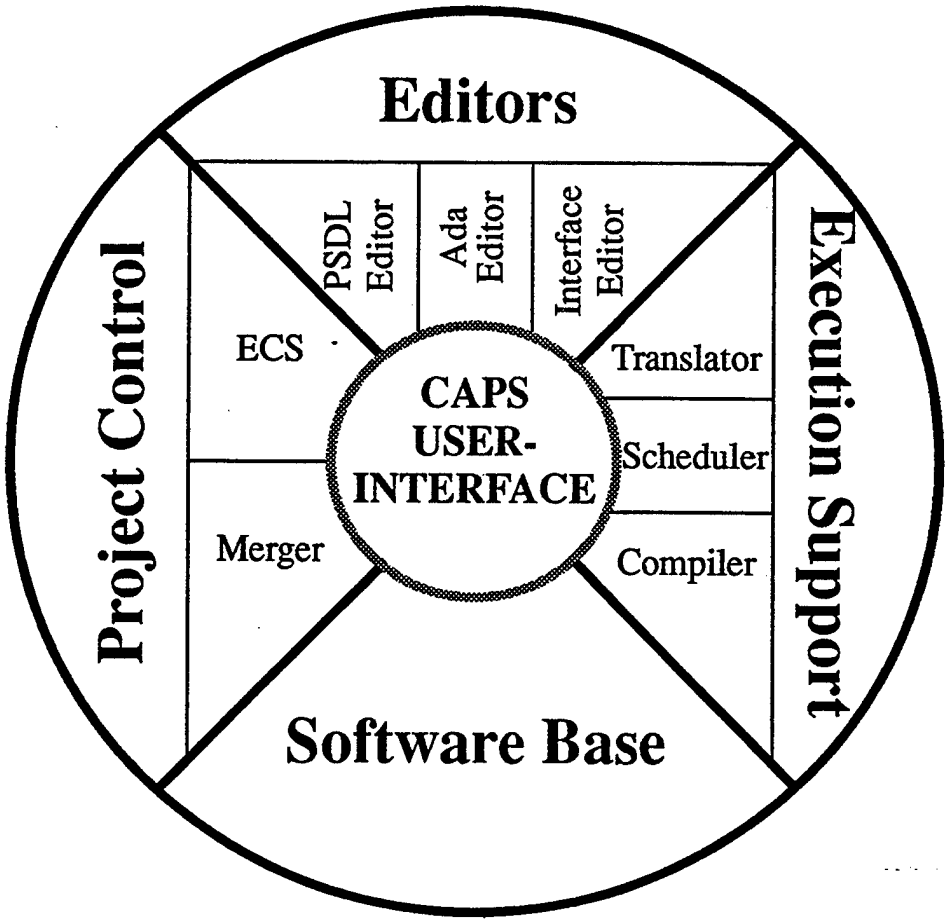


Figure 1. The CAPS Development Environment

A CAPS prototype is initially built as an augmented data flow diagram and a corresponding PSDL program. The CAPS data flow diagram and PSDL program are augmented with timing and control constraint information. This timing and control constraint information is used to model the functional and real-time aspects of the prototype. The CAPS environment provides all of the necessary tools for engineers to quickly develop, analyze and refine real-time software systems. The general structure of CAPS is shown in Figure 1.

As Figure 1 indicates, CAPS is a collection of tools, integrated by a user-interface. The CAPS User-Interface provides access to all of the CAPS tools and facilitates communication between tools when necessary. The tools in the figure are grouped into four sections, *Editors*, *Execution Support*, *Project Control* and *Software Base*. Each CAPS tool is associated with a different aspect of the CAPS prototyping process and will be discussed in further detail later in this document.

This document addresses basic and fundamental issues which arise when real-time prototypes are developed using CAPS and PSDL, providing guidelines and helpful hints along the way. These issues arise from the semantics of PSDL and from implementation specific details of CAPS. All of the information required to build a real-time prototype with CAPS is provided herein, including some fundamental real-time design concepts and CAPS- and PSDL-specific design requirements. It is intended that this document be used in conjunction with the CAPS User's Manual. This document outlines the CAPS prototyping process and concepts, and the CAPS User's Manual provides a detailed description of CAPS "buttonology".

B. Organization of Chapters

Chapter I of this document provides an overview of CAPS, the general prototyping process and the more specific CAPS prototyping process, and CAPS tools. Chapter II provides introductory system information required to run CAPS and a simple example of a complete CAPS prototype. For readers unfamiliar with PSDL, an introduction to PSDL is provided in Chapter III. Chapters IV-VIII provide in-depth descriptions of the phases of CAPS prototype development. A more detailed example of an autopilot system is provided throughout Chapters IV-VII, and real-time system design concepts are introduced during its development. Chapter VIII describes prototype evaluation and diagnosis procedures. Chapter IX is a summary that re-emphasizes the most important CAPS design concepts.

Throughout this document there are emphasis boxes. These boxes contain information that is very important for successful use of the current release of CAPS. The information addresses such issues as current CAPS implementation restrictions and pertinent aspects of the CAPS prototype generation process. In any event, they appear wherever vital information is presented.

This is an emphasis box. Be sure that you understand the comments in any such box or anything from great frustration to mild catastrophe may ensue!

If the meaning of the text in these emphasis boxes is not understood, the result can be anything from frustration to catastrophe!

C. The Prototyping Process

Rapid prototyping has been presented as an alternative paradigm for software development and evolution. The purpose of prototyping is ensure that proposed requirements and system concepts adequately match the needs of the prospective clients before detailed optimization and implementation efforts begin. "Computer-aided rapid prototyping improves the efficiency and accuracy of evolutionary development by introducing software tools that assist the designer in constructing and executing the prototype quickly and systematically" [Lu89b]. The general software development cycle with respect to prototyping is shown in Figure 2.

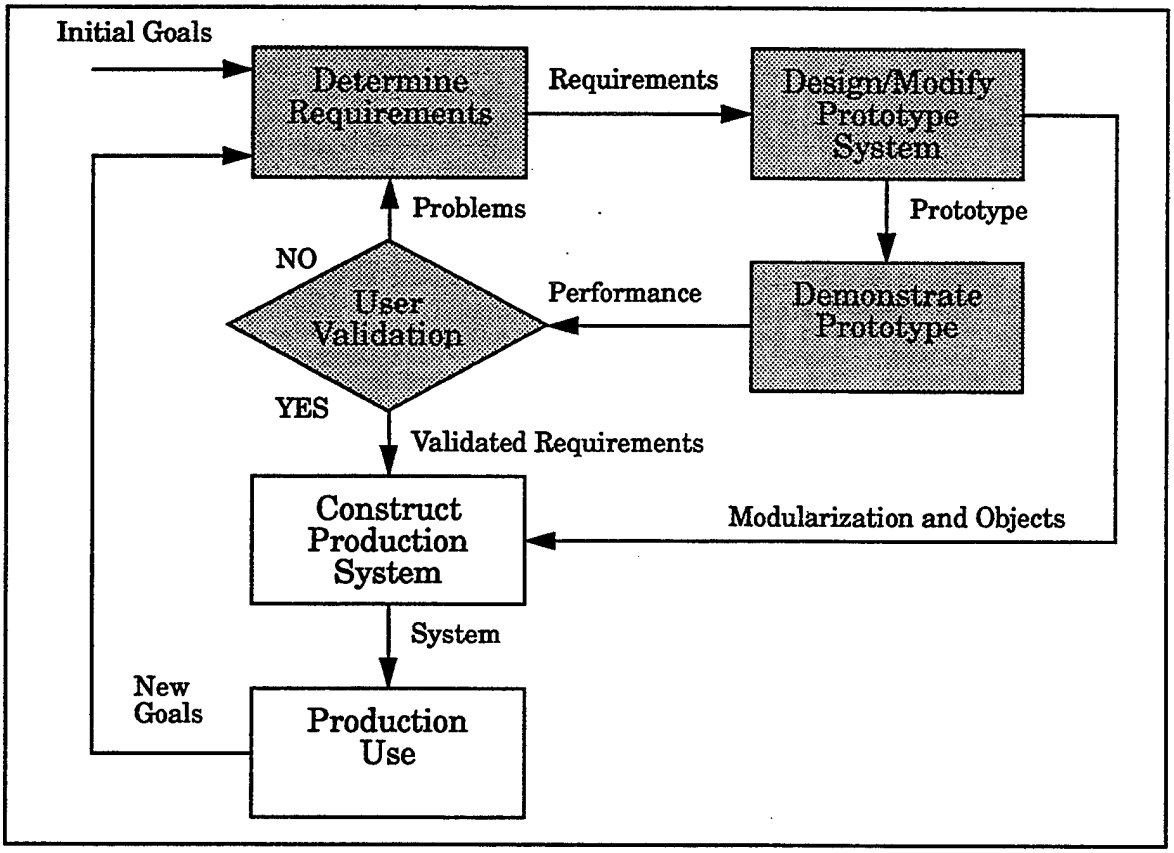


Figure 2. The Prototyping Process

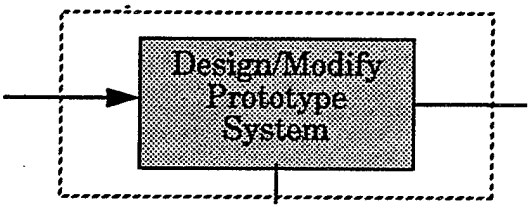
CAPS is specifically designed to assist and partially automate the development efforts which lie in the shaded regions of the prototyping process figure. Specifically, based on a set of initial requirements, CAPS allows the engineer to design, modify, demonstrate and validate a software system. Through this process, system requirements can be refined and modified as necessary.

The CAPS prototyping process is more specific than Figure 2, and is outlined below. The 10 enumerated steps are accomplished through use of the CAPS tools. Although the exact meaning of each step may not be clear at this point, specific details of each step are described in more detail in the indicated sections of this document. When developing a prototype, it will be helpful to refer back to this list, and consult the appropriate sections of this document and the CAPS User's Manual.

Basic CAPS Prototyping Process

1) Based on requirements, design (or modify) the data flow diagram for the system [Chapter IV, Section C].
2) Assign all appropriate timing and control constraints to the prototype operators. Assign latencies to data streams (if required) [Chapter IV, Section C].
3) Assign data types to all data streams [Chapter IV, Section C].
4) Find (in the software base) or build an implementation module for each user-defined data type and each atomic operator. Modules taken from the software base can be modified after retrieval to suit individual needs [Chapter V, Section B].
5) Build the prototype's user-interface (if required) [Chapter VII].
6) Translate the CAPS-generated (and user-augmented) PSDL program into (a portion of) the Ada supervisor module [Chapter V].
7) Run the CAPS scheduler to generate the static and dynamic schedules. This completes the prototype's Ada supervisor module [Chapter VI].
8) Compile the prototype. Note: for successful compilation, particular attention must be paid to the formal parameters of atomic operator implementation procedures created in step 4 [Chapter V, Section B].
9) Execute, evaluate and modify (if appropriate) the prototype and/or the requirements [Chapter VIII].
10) Return to Step 1 if prototype modification is required.

The correlation between these 10 steps and the shaded loop in the prototyping process is obvious, with the bulk of CAPS work being done in the portion of the diagram shown below.



Note that the basic 10 steps are a bit more detailed than the preceding prototyping process diagram. This highlights the real-time requirements, and associated design considerations of typical CAPS prototypes. The specific details of how to perform each of the 10 steps are covered in later sections of this document.

The remainder of this introduction briefly introduces the CAPS tools used to perform the basic 10 steps. Note, also, that two of the CAPS tools are outside the purview of the prototyping process diagram. These tools perform ancillary functions which are not seen in either the prototyping process diagram or the 10 basic CAPS steps. These advanced feature tools are the Evolution Control System and the Merger.

The purpose of the Evolution Control System is to provide automated support for coordinating the concurrent efforts of a team of prototype designers and to manage multiple versions of the designs they produce. The purpose of the Merger is to combine the effects of two or more enhancements to a prototype that have been independently developed. Details of these two tools are not presented in this document (see the CAPS User's Manual, [Ba93] and [Da94] for details).

CAPS can be executed in either the designer mode or the manager mode. The manager mode

provides access to CAPS advanced features, including modification of the designer pool, creation of project work steps, and prototype change-merging. CAPS supports distributed prototype development, and the manager interface provides facilities for such efforts. For simple, single-designer prototype building, the designer mode should be used. The next two figures show the CAPS designer and manager user-interfaces.

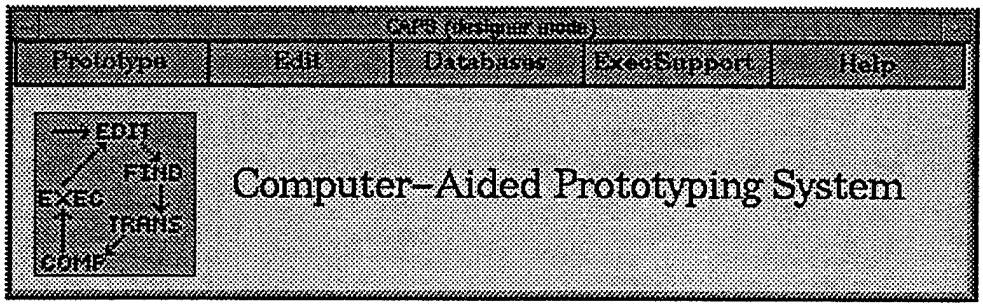


Figure 3. The CAPS User-Interface (Designer Mode)

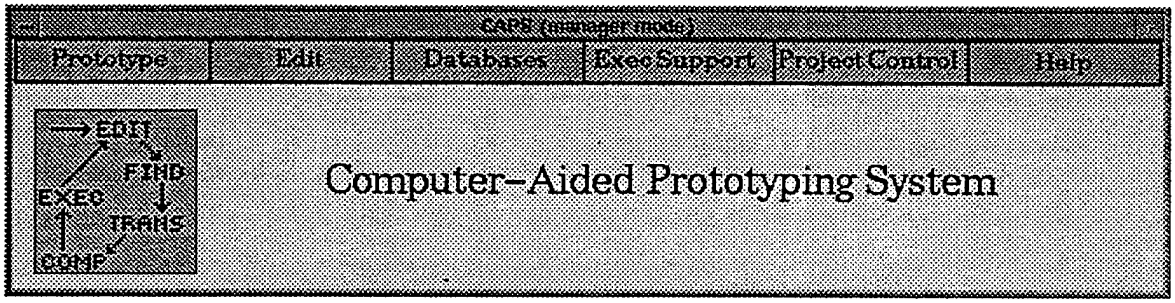


Figure 4. The CAPS User-Interface (Manager Mode)

D. CAPS Tools

This section provides a brief description of each CAPS tool. The details regarding the use of these tools can be found in the appropriate sections of this document and in the CAPS User's Manual.

1. The PSDL Editor

The PSDL Editor is the heart of CAPS prototype design. This editor consists of 3 separate parts: the Syntax Directed Editor, the Graph Viewer, and the Graphic Editor. This tool allows the designer to create the CAPS data flow diagram and PSDL program, and assign all timing and control constraints to prototype components (operators and data streams).

2. The Text Editor

Although the text editor is not exclusively a CAPS tool, CAPS does provide fluid integration of text editing facilities. Designers can select from vi, emacs and the Verdix Ada Syntax Directed

Editor (if available) for editing Ada programs. Use the "CAPS Defaults" selection under the CAPS "Edit" pull-down menu to make this selection. The CAPS User-Interface provides convenient file selection lists, based on the currently selected prototype.

3. The Interface Editor

CAPS integrates TAE+ [TAE93] for creation of window-based user-interfaces for prototypes. When using the TAE Workbench for creation of such user-interfaces, the designer must use the "single file", Ada code generation option from within TAE+. The automatically generated TAE code is placed in the prototype directory in a file called

<prototype_name>.RAW_TAE_INTERFACE.a.

For details about how to integrate this file into a prototype, see Chapter VII, Interface Integration. For details about the use of TAE+, consult [TAE93].

4. The Requirements Editor

The current version of CAPS does not have a sophisticated requirements tracking or editing tool. Simple text editor integration is provided for editing requirements documents associated with a prototype. CAPS will automatically present the user with a list of all files with a ".req" suffix when "Requirements" is selected from the "Edit" pull-down menu. After a file is selected, the default text editor will be invoked on that file.

5. The Change Request Editor

As with requirements, the current version of CAPS does not have a sophisticated change request tracking or editing tool. Simple text editor integration is provided for editing change request documents associated with a prototype. CAPS will automatically present the user with a list of all files with a ".cr" suffix when "Change Request" is selected from the "Edit" pull-down menu. After a file is selected, the default text editor will be invoked on that file.

6. The Translator

The CAPS translator converts a PSDL program into compilable Ada packages which implement supervisory aspects of the prototype. The translator expects a complete PSDL program as input, and creates several packages which make up, in part, the *supervisor module* of the prototype. It is important to note that the translator DOES NOT create Ada implementation packages for atomic operators or user-defined data types. These must be either extracted from the software base, or custom-made by the designer.

7. The Scheduler

The scheduler determines schedule feasibility for CAPS prototypes. Information is provided to the scheduler via timing constraints from the prototype's PSDL program. A prototype must be

translated before it can be scheduled, and scheduled before it can be compiled. Upon scheduling a prototype, CAPS provides schedule diagnostic information which can be analyzed and used to direct timing constraint modifications.

8. The Compiler

CAPS uses the SunAda Ada compiler. The compilation process is completely automated via the "Compile" command provided in the "Exec Support" pull-down menu in the CAPS User-Interface. Successful prototype compilation requires the formal parameter lists of atomic operator implementation modules to conform to CAPS interface conventions. See Chapter V for details.

9. The Evolution Control System

The CAPS Evolution Control System (ECS) [Ba93] is a system that supports distributed prototype development in a team environment. The ECS makes use of a design database (DDB) for persistent storage of prototype development data. The ECS supports maintenance of a designer pool from which to draw for prototype development tasks. Within the ECS, prototype development is modeled as a series of *steps*, which the project manager creates. These steps are automatically scheduled and assigned to available designers.

10. The Merger

The CAPS Merger [Da94] provides automated prototype *change-merging*. Based on *slicing* theory, applied to PSDL programs, the Merger automates the combination of two separate modifications to a base prototype. The Merger detects and warns of conflicts between the two changes to be merged. If no conflicts occur, or if they are overridden, the Merger creates a PSDL program for the newly created prototype which incorporates the changes of each of the modified prototypes.

11. The Software Base

The CAPS software base and its associated retrieval mechanism [Do93] provide access to a repository of reusable Ada and PSDL components. The software base allows a designer to browse as well as query its components. Queries to the software base can be in the form of keywords or PSDL specifications. In the current release of CAPS, the software base matching mechanism is based on parameter matching.

The current version of CAPS does not have a populated software base, however mechanisms are in place to conduct software base queries and to add components to the software base.

II. Getting Started

This chapter presents the basic initial steps that a user must take to use CAPS. A very simple example prototype is presented at the end of the chapter which can be recreated as an introduction to the CAPS prototyping process. Chapters IV-VII provide the details associated with all steps of prototype development with another, more complicated prototype than the one presented in this chapter. The purpose of this chapter is to allow a user to get CAPS up and running and get familiar with CAPS as quickly and painlessly as possible

CAPS requires the existence of a $HOME/.caps directory each user's workspace. If it does not exist, CAPS creates it.

CAPS uses a directory called $HOME/.caps. CAPS users CANNOT use a directory with this name for any other purpose.

To use CAPS, the following lines must be added to each CAPS user's .cshrc file:

```
setenv CAPSHOME <location_of_CAPS_software>
source $CAPSHOME/bin/CAPSsetup
setenv CAPS_DDB $user
```

Consult your CAPS administrator regarding the actual value of *<location_of_CAPS_software>*. The $CAPS_DDB environment variable can be left as $user or can be set to any constant value. The CAPSsetup file initializes $CAPS_DDB to $user, and the assignment in the individual user's .cshrc file is optional. It is this value that CAPS uses as the name of the active design database upon CAPS invocation. Note: CAPS users can modify the name of the active design database during a CAPS session.

A. Invoking CAPS

It does not matter from where in your directory structure you invoke CAPS.

1. The Designer Mode

Invoke CAPS in the designer mode by entering

caps

This command will bring up the main interface shown in Figure 3.

2. The Manager Mode

CAPS can be used in either the manager mode or the designer mode. The designer mode is

the default. To run CAPS in the manager mode, use the -m flag.

```
caps -m
```

This command will bring up the main interface shown in Figure 4. Depending on your situation, use of the manager mode may be restricted. Consult your CAPS administrator.

3. Choosing a Design Database

The name of the design database used during a CAPS session defaults to the value of the environment variable CAPS_DDB, or to the user's login name if CAPS_DDB is undefined.

To run CAPS with a design database other than the default, use the -d flag. Note that the design database being used can be changed during a CAPS session.

```
caps -d <other_ddb_name>
```

If the -d flag is used, the name of the design database to be used MUST follow the -d flag. The -m and -d flags can be used together, and the order is unimportant.

```
caps -m -d <other_ddb_name>
```

is the same as

```
caps -d <other_ddb_name> -m
```

B. A Simple and Complete CAPS Example

Consider a simple software system that interprets the input from a room temperature sensor and activates either a heating unit or a cooling unit. The complete development of this system will demonstrate the basic operation of CAPS. Many details of prototype development are skipped in this section, to be discussed in depth in Chapters IV-VII.

This discussion of the development of the TEMP_CONTROLLER prototype omits many details. See Chapters IV-VII for full development details.

Notice that in order to keep the **TEMP_CONTROLLER** example simple, many "realisms" are deleted. For example, there is no feedback from the heater or cooler to the temperature sensor. The **autopilot** prototype, presented in Chapters IV-VII of this tutorial, illustrates how such modifications could be made to the **TEMP_CONTROLLER** prototype.

1. Create and Edit the Prototype

After bringing up the CAPS User-Interface, the designer creates the first version of a new

prototype by selecting "New" from the "Prototype" pull-down menu. The designer will then be asked to provide the name of the new prototype, and the CAPS PSDL Editor will be automatically invoked. Upon initial creation of a prototype, the Syntax Directed Editor comes up with a single initial root operator (with a name the same as that of the prototype) as shown in Figure 5. To edit an existing prototype, the prototype must first be selected using the "Choose" command from the "Prototype" pull-down menu, and then the "PSDL" command in the "Edit" pull-down menu must be used explicitly (i.e. the PSDL editor IS NOT automatically invoked when an existing prototype is selected).

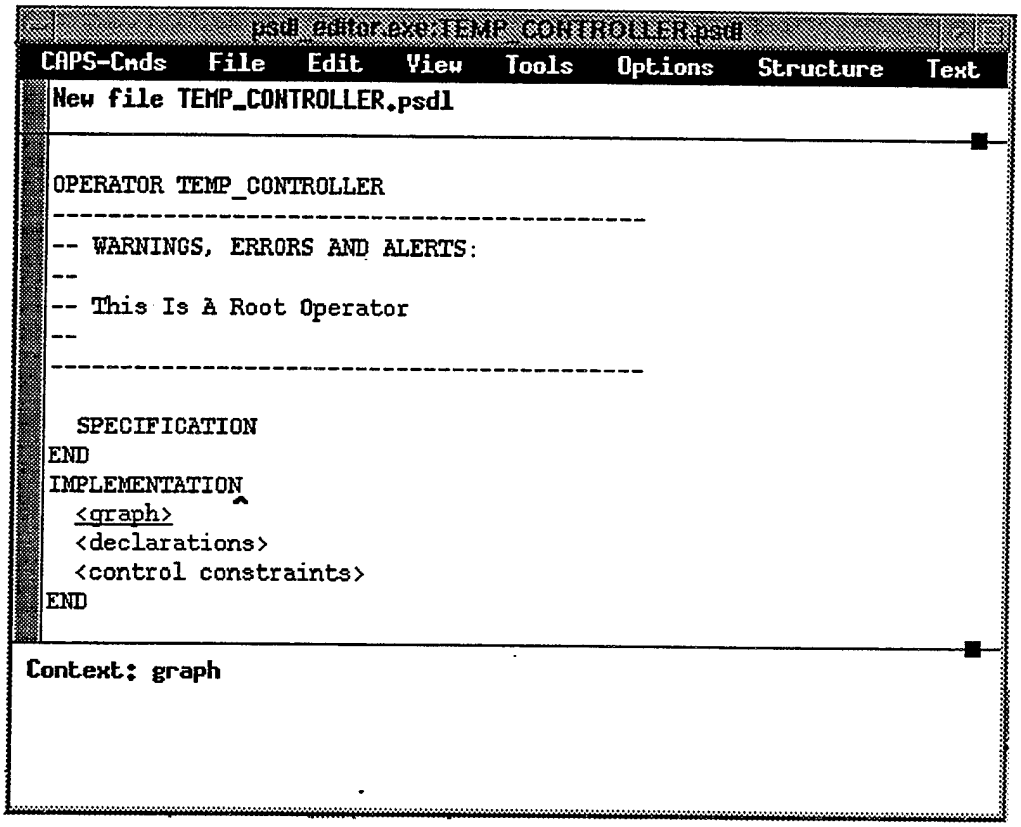


Figure 5. The CAPS Syntax Directed Editor

Invoke the CAPS Graphic Editor by using the "edit-graph" command from the "CAPS-Cmds" pull-down menu in the Syntax Directed Editor.

To invoke the CAPS Graphic Editor, the "edit-graph" command must be selected from the "CAPS-Cmds" pull-down menu in the Syntax Directed Editor.

All of the pull-down menus in the Syntax Directed Editor are active, however all of the necessary commands for PSDL editing and file saving are found in the "CAPS-Cmds" pull-down menu. DO NOT use the other pull-down menus. Use of the "File" pull-down menu in the Syntax

Directed Editor for saving PSDL programs WILL NOT save a correct PSDL program.

Use only the "CAPS-Cmds" pull-down menu in the Syntax Directed Editor. Specifically, do not use the "File" pull-down menu to save a PSDL program.

Figure 6 shows the initial appearance of the Graphic Editor for the **TEMP_CONTROLLER** prototype, which contains an empty graph.

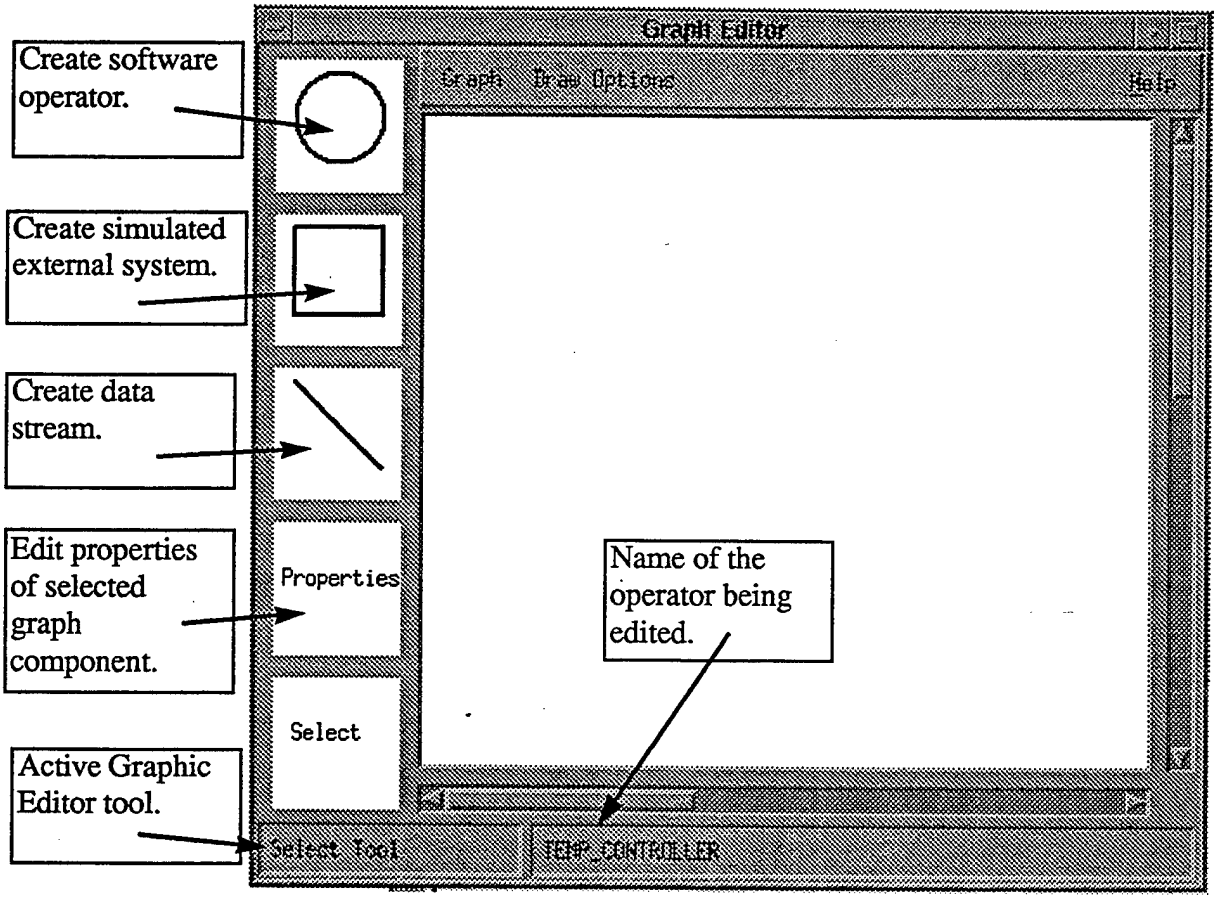


Figure 6. The CAPS Graphic Editor (with some notes)

Once in the Graphic Editor, the designer draws the data flow diagram, enters operator names, inserts input and output streams, and enters some of the prototype timing information. Timing information is completed, control constraints are entered, and implementation options are selected in the Syntax Directed Editor. See Chapter IV for details. The PSDL program for this prototype is listed in Appendix A.

Figure 7 shows the result of entering the decomposition of **TEMP_CONTROLLER**. The **TEMP_CONTROLLER** prototype has four operators and three data streams. The numbers next to **Sensor** and **Evaluate_Temp** represent maximum execution times, indicating that these two

operators are time-critical and that the other two operators (**Heater** and **Cooler**) are non-time-critical.

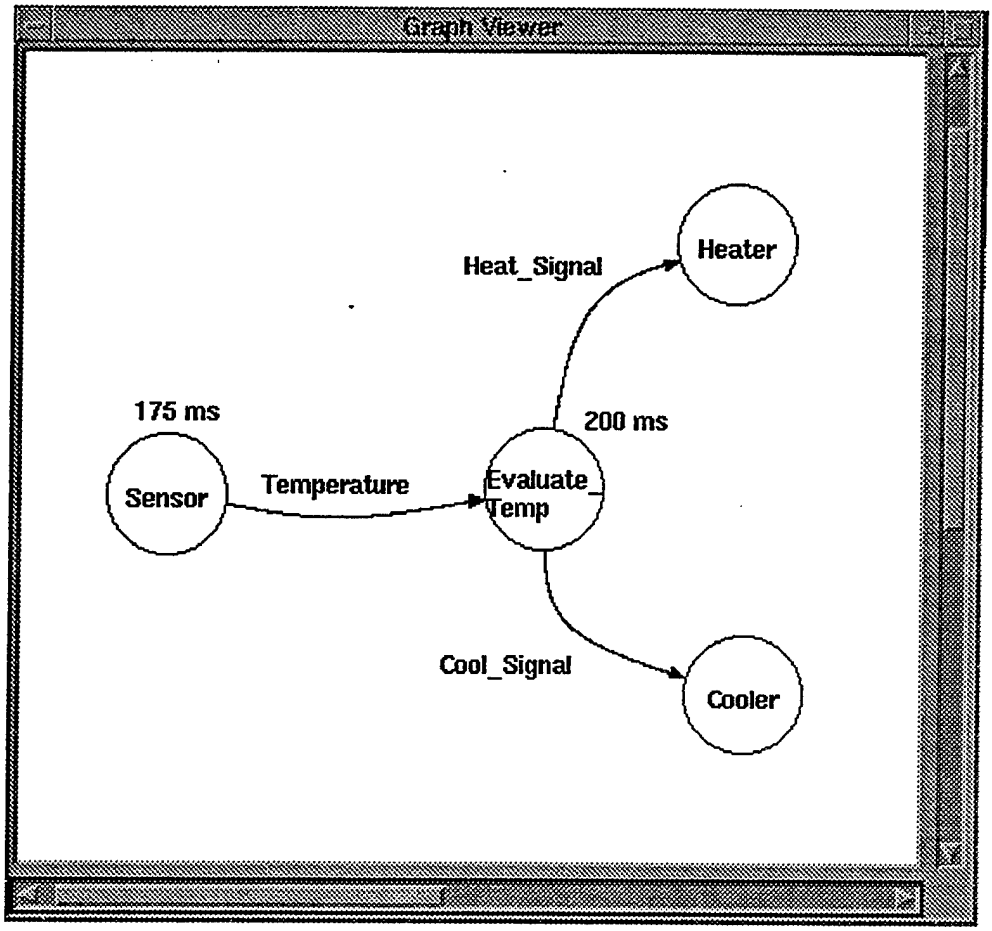


Figure 7. A Simple CAPS Example

2. Translate and Schedule the Prototype

When the designer finishes editing the prototype's graph and PSDL program, the next step is to translate and schedule the prototype, thus creating the supervisor module. For this step the designer simply selects "Translate" and then "Schedule" from the "Exec Support" pull-down menu. If there are any PSDL syntax errors, the translator will provide the designer with the location of the error. If the PSDL program was created using the Syntax Directed Editor, the only syntax errors should correspond to parts of the program that have not yet been filled in. All PSDL syntax errors must be fixed, and translation must be successful before the prototype can be scheduled.

A prototype must be successfully translated before it can be scheduled.

When the prototype has successfully been translated and scheduled, the prototype's supervisor file has been created. In this example, a file called TEMP_CONTROLLER.a is generated. This is the supervisor module. This file is listed in its entirety in Appendix A. CAPS generates schedule diagnostic information that can be useful for debugging purposes. This schedule diagnostic information is displayed after the prototype is scheduled. A complete listing of the **TEMP_CONTROLLER** prototype's schedule diagnostic information is provided in Appendix A.

3. Implement Ada Modules for Atomic Operators

In this step, the designer writes Ada programs for each atomic operator in the prototype. In this example, the following Ada modules are written:

TEMP_CONTROLLER.Cooler.a;
TEMP_CONTROLLER.Sensor.a;
TEMP_CONTROLLER.Heater.a; and
TEMP_CONTROLLER.Evaluate_Temp.a.

Note that each file name has the name of the root operator as a prefix (**TEMP_CONTROLLER** in this case). Each of these implementation files is listed in Appendix A.

4. Compile the Prototype

Once the designer has completed the coding of atomic operator implementations, the prototype can be compiled. This is accomplished by selecting the "Compile" command from the "Exec Support" pull-down menu. All new and/or modified Ada modules will be automatically compiled and linked by the system. During this step, if there are any Ada syntax errors, the designer must fix them before an executable prototype is generated. After successfully compiling the prototype, the designer can execute the prototype.

5. Execute the Prototype

The designer executes the prototype by selecting "Execute" from the "Exec Support" pull-down menu. If any problems occur during execution, alerts will be presented in the prototype execution window. The prototype execution window is labeled "<prototype_name>.exe" and serves as the standard input/output location for the prototype. If a fatal execution error occurs, details are presented in the CAPS alert window. A sample of the output from the **TEMP_CONTROLLER** prototype appears in Appendix A.

III. An Introduction to PSDL

A. Overview

The Prototype System Description Language (PSDL) is a language for describing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. PSDL provides facilities for modelling timing and control constraints within a software system and has been implemented in the Computer-Aided Prototyping System (CAPS). A PSDL prototype is built as an hierarchical decomposition of data flow diagrams, augmented with timing and control constraint information. PSDL is based on a computational model that contains operators which communicate via data streams. Each data stream carries values of a fixed type. Streams carrying exception values are of type PSDL_EXCEPTION.

1. Operators

An operator represents either a function or a state machine. When it fires, an operator reads one data object from each of its input data streams and writes at most one data object on each of its output streams. If the output depends only on the current set of input values, then the operator represents a function. If, in addition, the output of the operator depends on the current value of a state variable, then the operator represents a state machine. Operators are either *periodic* or *sporadic*. Periodic operators are explicitly assigned a frequency (period) of execution. An operator with a period of 500 milliseconds fires (executes) once every 500 milliseconds. Also associated with periodic operators are maximum execution time (MET) and finish within (FW). The maximum execution time specifies the greatest amount of CPU time that an operator can use for execution. Finish within is an upper bound on the total duration from beginning of each period to completion of execution for an operator. Operators which are assigned a maximum execution time are *time-critical*. In the current implementation of CAPS, time-critical operators are non-preemptable, thus maximum execution times correspond to uninteruptable blocks of time in the static schedule. All periodic operators are time-critical. Sporadic operators are not explicitly assigned a period, however they can be time-critical as discussed in Chapter VI. Operators that are decomposed into lower level(s) are called *composite* operators. This decomposition is always functional. An operator that is not decomposed is called *atomic*. In the current version of CAPS, atomic operators are implemented in Ada.

2. Data Streams

A data stream is a communication link that connects two sets of operators: the producers and the consumers of the stream. Data streams are represented as edges in the CAPS augmented data flow diagram. There are two types of data streams: *sampled* streams and *data flow* streams. The data trigger of a consuming operator determines the type of a data stream: if the consuming operator fires on every occurrence of data on a data stream (i.e. is in a "BY ALL" data trigger), then the stream is a data flow stream; otherwise it is a sampled stream. It is useful to think of a data flow stream as a FIFO queue of size one, and a sampled stream as a programming variable. Sampled streams can always be read (as long as they are non-empty), even if new data has not arrived on the stream. Information on data flow streams must be consumed at the same frequency it is

written (i.e every piece of data written to the stream must be used by the consuming operator). All PSDL data streams contain at most one data item at any given time.

3. Types

PSDL user-defined data types are abstract data types (ADTs) which can be used in CAPS prototypes. PSDL types, like PSDL operators, can be implemented in either PSDL or Ada. Types can have associated with them a set of operators. Types implemented in Ada are realized by an Ada package that defines a private type and a subprogram for each operator on the type.

B. Timing Constraints

PSDL allows a designer to specify timing constraints for each prototype operator [Lu89a]. Operators in PSDL can be either *periodic* or *sporadic*. Periodic operators are operators which are explicitly assigned a maximum execution time and a frequency of execution (a period). An operator with a period of 500 milliseconds fires (executes) once every 500 milliseconds. The maximum execution time specifies the greatest amount of CPU time that an operator can use for execution. Also associated with periodic operators is "finish within" (FW). Finish within is the maximum duration from the beginning of an operator's period to completion of execution. In the current implementation of CAPS, time-critical operators are not preemptable, thus maximum execution times correspond to uninterrupted blocks of time in the static schedule. All periodic operators are time-critical. Table 1 shows the allowable timing constraints for periodic and sporadic operators.

Periodic Operators	Sporadic Operators
Maximum Execution Time (MET)	Maximum Execution Time (MET)
Period (P)	Minimum Calling Period (MCP)
Finish Within (FW)	Maximum Response Time (MRT)

Table 1: Allowable operator timing constraints by operator type

Sporadic operators can be time-critical or non-time-critical. Sporadic operators which are assigned a maximum execution time are time-critical, otherwise they are non-time-critical. As indicated by the table, also associated with sporadic operators are maximum response time (MRT) and minimum calling period (MCP). The maximum response time is the maximum duration from the satisfaction of all data trigger conditions for an operator to the completion of execution. The minimum calling period is the shortest allowable duration between two successive triggerings of a sporadic operator. The minimum calling period of an operator constrains the behavior of the producers of the triggering data values rather than constraining the behavior of the operator itself. It is useful to think of minimum calling period as a "minimum trigger satisfaction period".

For any prototype, CAPS executes the set of time-critical operators and the set of non-time-critical operators in two separate Ada tasks. One (higher priority) task controls the time-critical operators and another (lower priority) task controls the non-time-critical operators. In CAPS, time-critical operators are not preemptable and execute to completion, even if they exceed their maximum execution time. Re-evaluation and modification of a prototype's timing constraints is required when an operator exceeds its maximum execution time more often than allowed by the

system requirements.

The relationships among timing constraints associated with real-time systems differ according to the target hardware architecture. The current implementation of CAPS generates prototypes for a uni-processor run-time environment and implements time-critical sporadic operators as equivalent periodic operators (see Chapter VI for details). Brief descriptions of the relations between key timing constraints in such an environment follow.

1. Periodic Operators

$$MET \leq FINISH_WITHIN \leq PERIOD$$

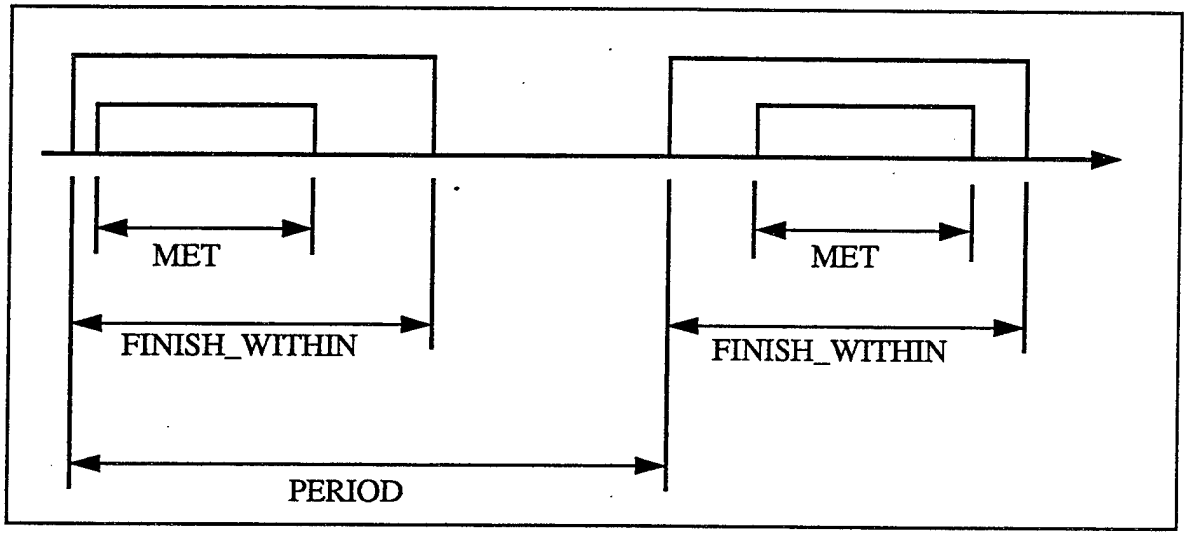


Figure 8. Timing Constraints for Periodic Operators

2. Sporadic Operators

$MET \leq MRT / 2$
$MET < MCP$
$MET < equivalent_triggering_period$
$equivalent_triggering_period \leq minimum\ [(MRT - MET), MCP]$
$equivalent_finish_within =$
$\quad minimum\ [equivalent_triggering_period, (MRT - equivalent_triggering_period)]$

Notice that for time-critical sporadic operators, equivalent triggering periods and equivalent finish within values are created. CAPS automatically generates these values in order to implement time-critical sporadic operators as equivalent periodic operators.

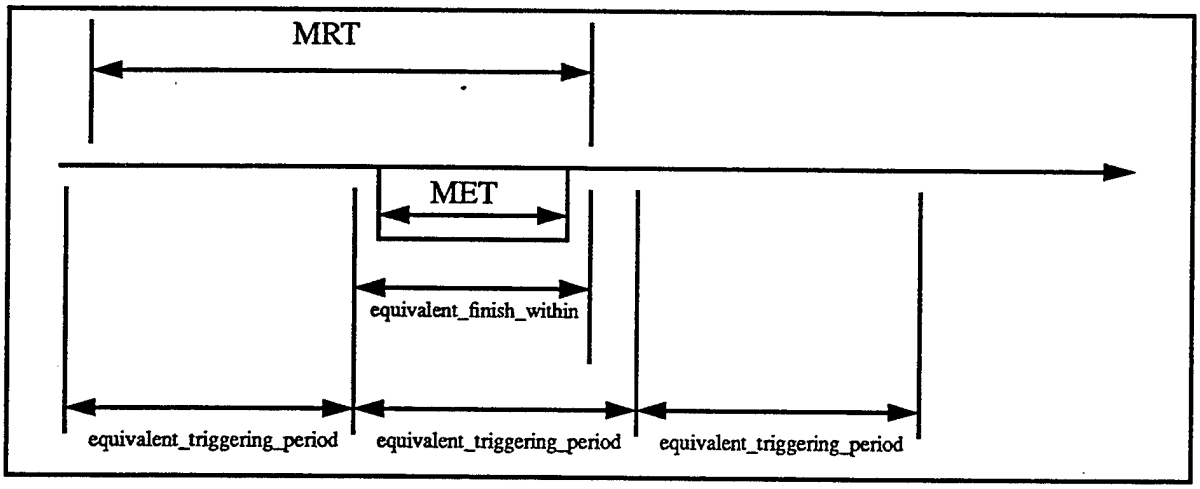


Figure 9. Timing Constraints for Sporadic Operators

C. "BY ALL" Triggers

An operator that is triggered BY ALL instances of a data stream will fire every time there is new data on ALL streams listed after "TRIGGERED BY ALL" (this is the "triggering set"). As previously mentioned, this makes the associated data streams data flow streams. This kind of trigger should be used when the items in a stream represent discrete events (e.g. transactions on a bank account) rather than samples from a continuous source of data (e.g. a temperature sensor).

The most important design consideration when "BY ALL" triggers are used is management of the firing frequencies of the producing and consuming operators. The period of the consuming operator must be at least as small as the period of the producing operator, or stream buffer overflow errors will result (i.e. the consuming operator must fire at least as often as the producing operator). This is because the data streams in CAPS can hold at most one data item. CAPS ensures that if the consuming operator's period is less than that of the producing operator, the actual firing rate of the two will be the same (i.e. "BY ALL" trigger data streams are tested for new information prior to actually firing the consuming operator).

D. "BY SOME" Triggers

"BY SOME" triggers represent sampled data streams. Operators with "BY SOME" triggers can fire whenever new data arrives on at least one of the streams listed after "TRIGGERED BY SOME". This means that old data can possibly be read from some of the streams in the "BY SOME" triggering set.

Sporadic operators with no data triggers will fire whenever there is free time in the CAPS schedule. This can result in MANY firings of untriggered sporadic operators.

E. Execution Guards

The firing of a PSDL operator can be regulated by an *execution guard.* Execution guards are conditional statements which are evaluated prior to firing the associated operator. Execution guards can depend on data from any incoming data stream. Execution guards can be combined with the "BY ALL" and "BY SOME" data triggers mentioned above. Even if an execution guard is not satisfied, and the operator does not fire, the data on each input data stream is consumed when the execution guard is evaluated.

F. Conditional Output

PSDL conditional output is implemented in CAPS as guarded execution of code which writes values to data streams. Conditional output does not affect the firing of an operator. An operator will fire in accordance with the CAPS schedules regardless of whether or not its output is written to an output data stream. The condition of an output guard may depend on the output values of the operator as well as on the values read from the input streams.

G. State Variables

A CAPS prototype is accepted by the scheduler only if its graph representation (excluding all state streams) is a directed acyclic graph (DAG). This restriction may not seem to make sense at first glance. However, when a prototype graph contains a cycle, this indicates the presence of state information. States must be declared and initialized. PSDL fully supports the integration of states in its prototypes.

When a data stream is identified as a state stream, it is, in essence, removed from the graph for scheduling purposes. Thus, when loops are introduced into prototype data flow designs, there must be a state declared. The name of the state must be the same as the name of the data stream which connects back to the beginning of the loop.

When a state is introduced into an atomic operator, it must be implemented by the Ada code for that operator. The variable **Local_Temperature** in the **Sensor** operator of the **TEMP_CONTROLLER** prototype developed in Chapter II is an example of such an implementation. Notice that the **Sensor** operator declares a state called **Local_Temperature** in the **TEMP_CONTROLLER** PSDL program (see Appendix A).

H. Exceptions

Exceptions in PSDL are values that can be transmitted on data streams of the type "PSDL_EXCEPTION". During prototype execution, unhandled Ada exceptions are transformed into PSDL exceptions. Exceptions can also be raised by "EXCEPTION" control constraints.

I. Timers

PSDL *timers* are software stopwatches that are used to measure and control durations of particular states. They are governed by the control constraints "START TIMER", "STOP TIMER" and "RESET TIMER".

IV. PSDL Editing

A. Overview

In CAPS, a prototype is built and edited through both graphic and textual manipulation of its PSDL program. The CAPS PSDL Editor incorporates two interfaces to perform these manipulations. Graphic editing is accomplished using the CAPS Graphic Editor, and text editing is accomplished using the CAPS Syntax Directed Editor. These two editors communicate information about a prototype's data flow diagram.

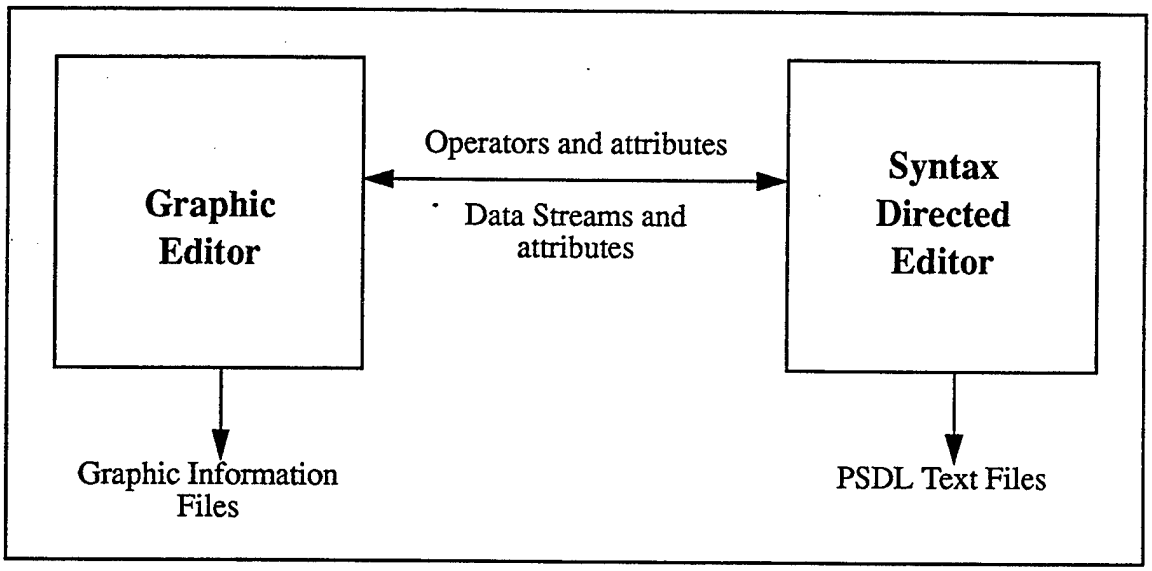


Figure 10. Simplified Graphic Editor and Syntax Directed Editor Communication

Warnings of any inconsistencies between the textual PSDL program in the Syntax Directed Editor and the graphical PSDL implementation in the Graphic Editor are presented to the designer in the Syntax Directed Editor. This chapter describes the basic operation of the CAPS PSDL Editor. A prototype of a very simple autopilot system is developed in parallel with the discussion of the PSDL Editor, and this development continues through Chapter VII.

B. A Quick Review of PSDL Operators and Data Streams

A CAPS prototype is designed as a decomposition of *operators* and its implementation is represented as an augmented data flow diagram. The operators in a prototype implementation are represented as nodes in a CAPS data flow diagram and communication between the operators is via *data streams*. There is an implied top-level node (operator) which represents the entirety of any prototype. This top-level node has no inputs or outputs and is not explicitly depicted in the data flow diagram. The top-level operator of a prototype is represented as a PSDL program. The CAPS data flow diagram represents the PSDL implementation of the top-level prototype operator.

Each operator in a prototype can, in turn, be decomposed into a PSDL (graphic) implementation. Such an operator is said to be *composite*. Operators that do not have PSDL decompositions, but are implemented in a compilable language such as Ada, are said to be *atomic*. The implemen-

tation source code for atomic operators is either extracted from the CAPS software base or custom built by the designer. In PSDL, every time-critical operator must have a period or a data trigger, or both. For time-critical operators to which no period or trigger is assigned, CAPS will generate a period. This is explained in more detail in Chapter VI, Section F entitled Equivalent Periods.

Data streams are represented as edges in the CAPS data flow diagram. The data streams in a prototype are either *sampled* streams or *data flow* streams. The data trigger of a consuming operator determines the type of a data stream. Data triggers are discussed in Chapter III. If a data stream is in a "BY ALL" trigger (i.e. the consuming operator fires upon every occurrence of data on the stream), then the stream is a data flow stream, otherwise it is a sampled stream.

The difference between data flow and sampled streams is significant and should be understood when building a prototype. Sampled streams can be written as often as desired, regardless of how often they are read. Sampled streams may also be read when they do not have new data (as long as the stream is non-empty). A data flow stream, however, must be interrogated at least as frequently as it is written. Every piece of data written to a data flow data stream must be read by the consuming operator. This is what is meant by a "BY ALL" data trigger. The data streams in CAPS cannot hold more than one data value at a time. This concept should be understood when designing a prototype, specifically when assigning periods (frequency of execution) and triggering conditions to operators.

C. Graphic Editor and Syntax Directed Editor Interactions

The first level of nodes and edges in the CAPS Graphic Editor represents the decomposition of the implied top-level node. To view the graphic representation of a PSDL implementation, the designer should use the mouse to position the Syntax Directed Editor cursor in the corresponding portion of text in the PSDL program. CAPS provides a Graph Viewer in addition to the Graphic Editor. The Graph Viewer is simply a non-editable static display of a graphic decomposition and appears with the Syntax Directed Editor on the designer's screen when a PSDL editing session is initiated. The Graphic Editor is invoked from within the Syntax Directed Editor using the "edit-graph" command from the "CAPS-Cmds" pull-down menu.

To invoke the CAPS Graphic Editor, select the "edit-graph" command from the "CAPS-Cmds" pull-down menu in the Syntax Directed Editor.

After editing the prototype with the Graphic Editor, when the designer returns to the Syntax Directed Editor, appropriate information is sent to the Syntax Directed Editor and appropriately placed in the PSDL program. Upon exiting the Graphic Editor, the Syntax Directed Editor and Graph Viewer are again presented to the designer. By its very nature, the Syntax Directed Editor ensures syntactically correct PSDL programs. To view the new graphic information in the Graph Viewer, the designer need only click the left mouse button in the appropriate portion (the GRAPH portion) of the PSDL implementation text in the Syntax Directed Editor.

Information is entered into the PSDL program both via the Graphic Editor and the Syntax Directed Editor. As CAPS evolves, the degree of automation and accuracy of PSDL program generation increases. In future versions of CAPS, a designer will be able to enter all PSDL program

information graphically, thus creating a sort of graphical version of the PSDL language. Additionally, any information entered textually via the Syntax Directed Editor in a PSDL program will be automatically reflected in the graphic display should the designer wish to directly manipulate the PSDL code. However, in the current implementation of CAPS, the PSDL Editor requires that certain information be entered in the Graphic Editor, and certain other information be entered in the Syntax Directed Editor, as summarized in Table 2.

In the current version of CAPS, certain prototype information MUST be entered using the Graphic Editor, while certain other information MUST be entered using the Syntax Directed Editor.

Graphic Editor Information Entry	Syntax Directed Editor Information Entry
vertices and edges of the graph (operators and data streams)	control constraints: execution guards, triggers, output guards and timing constraints
operator names	data stream types
data stream names	timer declarations
operator maximum execution times	state declarations and initializations
data stream latencies	user-defined types
operator color- and shape-coding	operator and type implementation selection (Ada or PSDL)

Table 2: Summary of information entered using the CAPS Syntax Directed Editor and the CAPS Graphic Editor

1. Graphic Editor / User Interaction

Figure 11 illustrates the empty CAPS Graphic Editor upon initial creation of the **autopilot** prototype.

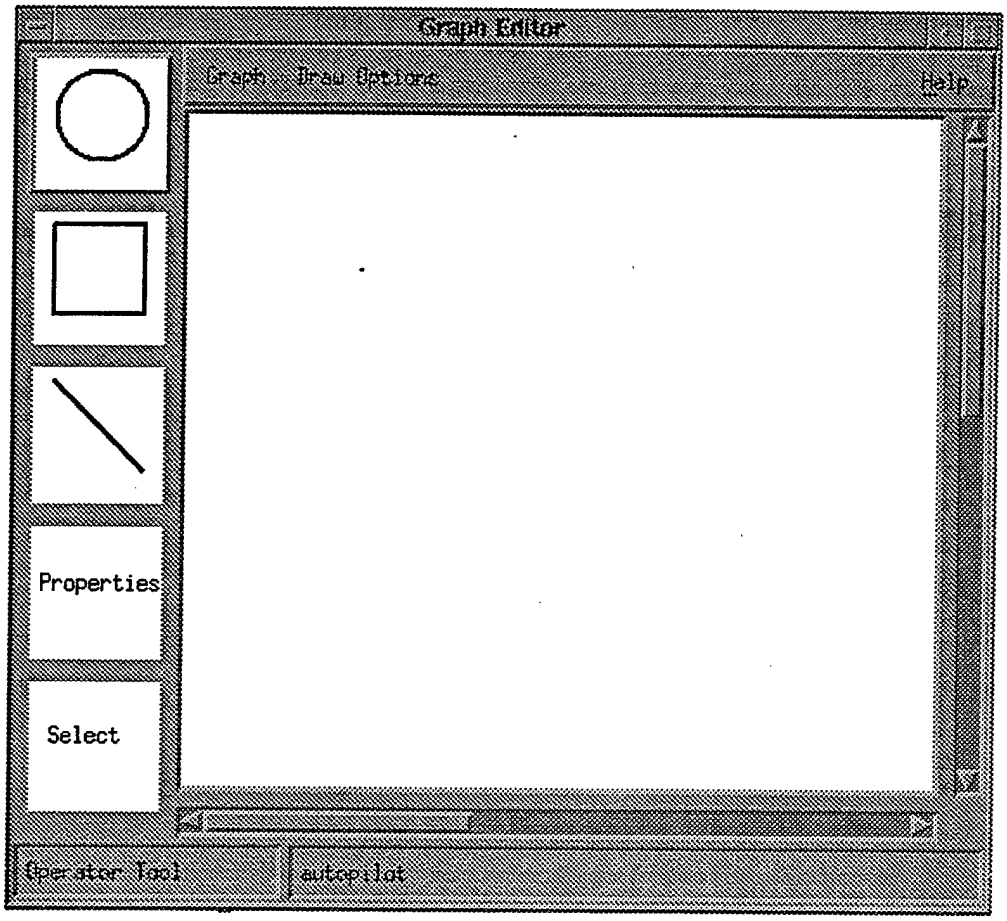


Figure 11. The CAPS Graphic Editor

The CAPS Graphic Editor is used first and foremost to lay out the data flow design of a prototype. Operators are linked together with data streams and both are given names. Context sensitive attributes are assigned to operators and data streams. These attributes are maximum execution time for operators and latency for data streams. (The latency of a data stream is a lower bound on the amount of time required for transmission of data along that stream.) The figures and words that appear on the left hand side of the Graphic Editor (the Graphic Editor palette) are the editing tools. Their functions are summarized as follows:

CIRCLE.....Draw circular operators (representing proposed software components).
SQUARE.....Draw rectangular operators (representing simulations of external systems).
LINE.....Draw data streams.
"Properties".....Assign properties to the selected operator or data stream.
"Select" .....Enable selection of an object in the graph.

The name of the active tool is displayed in the lower left portion of the Graphic Editor and the name of the operator being edited is displayed in the lower right portion.

After building the data flow diagram for the **autopilot** prototype and returning to the Syntax Directed Editor, we begin to see the formation of a PSDL program and a prototype. The picture

representing the **autopilot** prototype is transferred to the Graph Viewer, and is displayed by positioning (with the mouse) the Syntax Directed Editor's cursor in the GRAPH portion of the **autopilot** root operator's implementation. The Graph Viewer display of the **autopilot** prototype is shown in Figure 12. The complete PSDL program for the initial version of the **autopilot** prototype is listed in Appendix B.

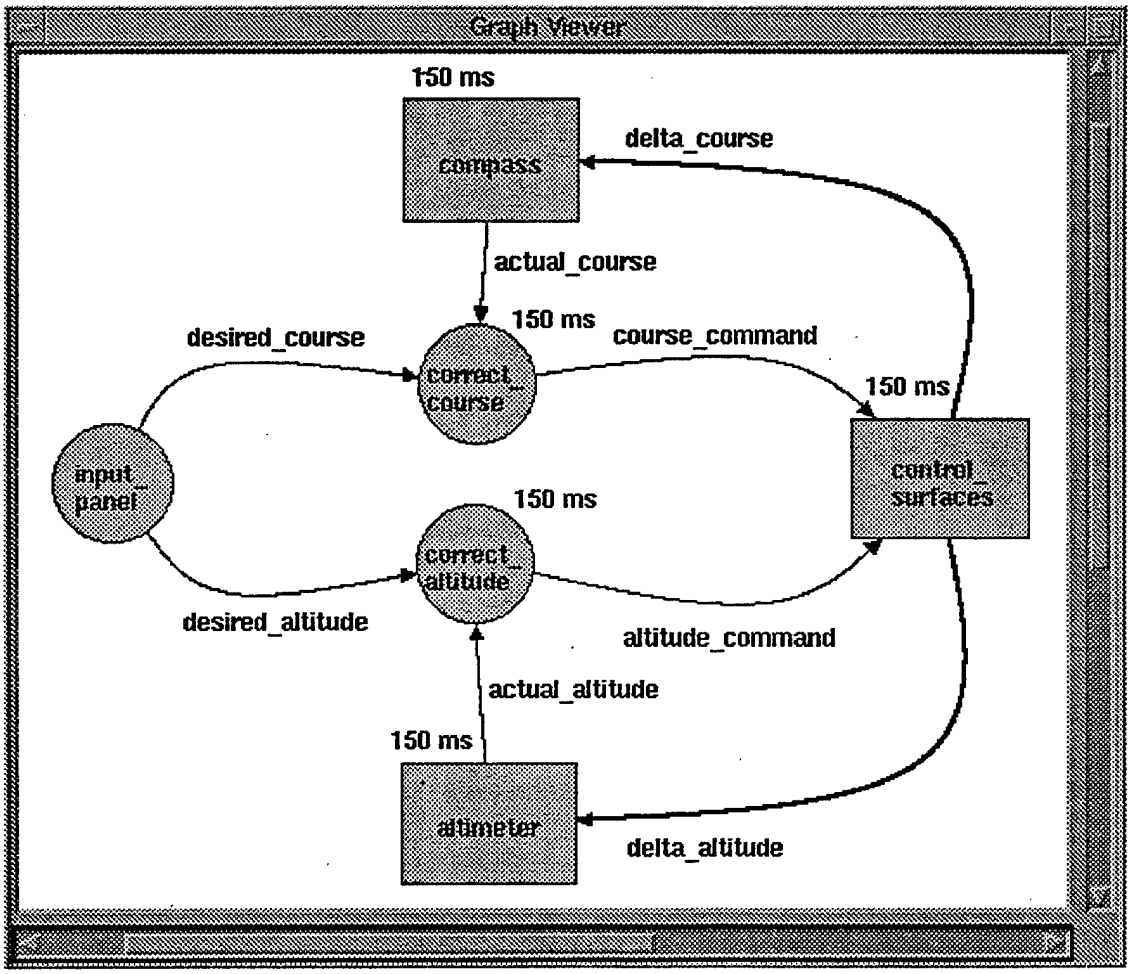


Figure 12. The CAPS Graph Viewer Showing the autopilot Prototype

Operator maximum execution times and data stream latencies (there are no data stream latencies in this prototype) are the numerical values that are displayed on the Graphic Editor (and Graph Viewer) near the associated operator or data stream. Composite operators are indicated on the CAPS Graphic Editor as double circles (there are no composite operators in the **autopilot** prototype).

The most obvious thing that has been done at this point of development of the **autopilot** prototype is that a graph has been created. This is accomplished by selecting the appropriate tool from the Graphic Editor palette and "drawing" the graph. Names are given to operators and data streams by first selecting the object, and then using the "Properties" tool in the Graphic Editor. Maximum execution times and latencies are likewise assigned to operators and data streams, respectively, using the "Properties" tool.

The darker shaded data streams in the **autopilot** prototype are state streams. Although there is a selection button for state streams in the Graphic Editor data stream "Properties" tool, to create state streams, the designer must enter the state stream information in the Syntax Directed Editor. State stream declarations and initializations appear in the specification portion of operators.

To make a state stream, you must enter the appropriate information in the Syntax Directed Editor. State streams declared in atomic operators should be implemented in the executable source code (Ada) as variables local to the Ada package.

The CAPS Graphic Editor allows a designer to color-code operators and change the fonts of displayed text. Additionally, rectangular operator representation is available to indicate hardware simulation or other operations which may be deemed "external" to the system being prototyped. In the **autopilot** prototype, the operators **compass, altimeter** and **control_surfaces** are such operators. The PSDL Editor treats operators represented as circles the same as operators represented as rectangles.

The **autopilot** prototype has no composite operators, but if it did, they would be represented as double circles. Rectangular operators can be decomposed, however there is no visual cue to indicate that they are composite. For this reason, composite operators should be represented as circular graph vertices.

a. Drawing Streams

Data streams can be drawn as straight lines or as curved lines. When drawing data streams, they are started using the left mouse button, splined using the left mouse button, and terminated using the middle or right mouse button (or double-clicking for "External" streams). Adding splines to data streams provides more "artistic" flexibility in creating a data flow diagram.

b. Making External Streams

In the decomposition of composite operators, there should be "External" data streams that correspond to the inputs and outputs of the decomposed operator. To make an external output data stream, the data stream is terminated by double-clicking the mouse at the desired termination point.

c. Moving Objects

Objects in the data flow diagram can be dragged (moved) by having the "Select" tool active and dragging the object to its new location with the left mouse button (hold the button down while moving the mouse).

d. Deleting and Renaming Operators

When operators are renamed or deleted from a PSDL graph, the corresponding PSDL specifications and implementations remain in the PSDL program. This is partly for conservative safety

reasons and partly due to current implementation restrictions. When an operator is deleted from a prototype graph, the operator's specification and implementation must be manually deleted in the Syntax Directed Editor. Similarly, if an operator is renamed, the PSDL program will have a specification and an implementation under both the new name and the old name. The operator with the old name IS NOT automatically removed from the PSDL program. In such cases, a common Syntax Directed Editor alert is:

```
-----------------------------------------
-- WARNINGS, ERRORS AND ALERTS:
--
-- This Is A Root Operator
--
-- You have multiple roots
-- Please delete the obsolete ones
--
-----------------------------------------
```

To correct this situation, the obsolete operator should be manually removed from the PSDL program. The best way to do this is by selecting the operator to be deleted (the selected structure in the Syntax Directed Editor is indicated by underlining) and using the keystroke

"Control-Shift-K".

This keystroke deletes the selected PSDL structure.

e. Undeleting Operators

When operators are deleted from a prototype graph, the deletions can be undone, but only during that graphic editing session. When a graphic editing session is completed, all deletions become permanent. To undelete an operator during a single graphic editing session, select the "Undelete Operator" command from the "Draw Options" pull-down menu in the Graphic Editor. A list of all deleted operators will be presented. Select the operator to undelete by double-clicking the mouse on the operator name. The operator, along with its associated data streams, will be re-added to the prototype graph.

f. Decomposing Operators

Individual operators in a prototype can be decomposed into a PSDL (graphical) implementation (i.e. be made composite) by selecting the "Decompose" command from the "Graph" pull-down menu in the Graphic Editor. The information of such decompositions will be transferred to the Syntax Directed Editor in a fashion similar to that used in the transfer of root operator (complete prototype) information. The **autopilot** prototype has no composite operators.

To summarize the prototype information entered by the designer in the Graphic Editor:

1) prototype graph configuration,

2) operator and data stream names,
3) operator maximum execution times,
4) data stream latencies,
5) operator color- and shape-coding as appropriate.

Operator and data stream names MUST be assigned in the Graphic Editor. If they are not, dummy names will be assigned. These dummy names can only be changed from within the Graphic Editor.

Upon exiting the Graphic Editor, information representing the prototype graph is automatically transferred to the Syntax Directed Editor. To display the graphic information in the CAPS Graph Viewer, click the left mouse button in the GRAPH portion of the prototype root operator. The Graph Viewer presents a non-modifiable representation of the prototype graph.

The CAPS Graph Viewer is non-modifiable. To modify a data flow diagram, the Graphic Editor must be invoked using the "edit-graph" command in the "CAPS-Cmds" pull-down menu.

The PSDL program generated by graphical interaction is not complete. There are still placeholders in the PSDL program. A placeholder is a piece of text in angled brackets, like "`<decl_type_name>`", that represents a required, but missing piece of the program. The Syntax Directed Editor is used to complete the PSDL program.

In the current version of CAPS, certain prototype information MUST be entered using the Graphic Editor, while certain other information MUST be entered using the Syntax Directed Editor.

2. Syntax Directed Editor / User Interaction

Completion of the PSDL program is accomplished using the Syntax Directed Editor. An important aspect of PSDL programs is that each component in the program must have a specification part and an implementation part.

Every component in a PSDL program (types and operators) must have a specification part and an implementation part or translation errors will result.

Figure 13 shows the Syntax Directed Editor upon initial creation of the **autopilot** prototype,

before the Graphic Editor has been invoked.

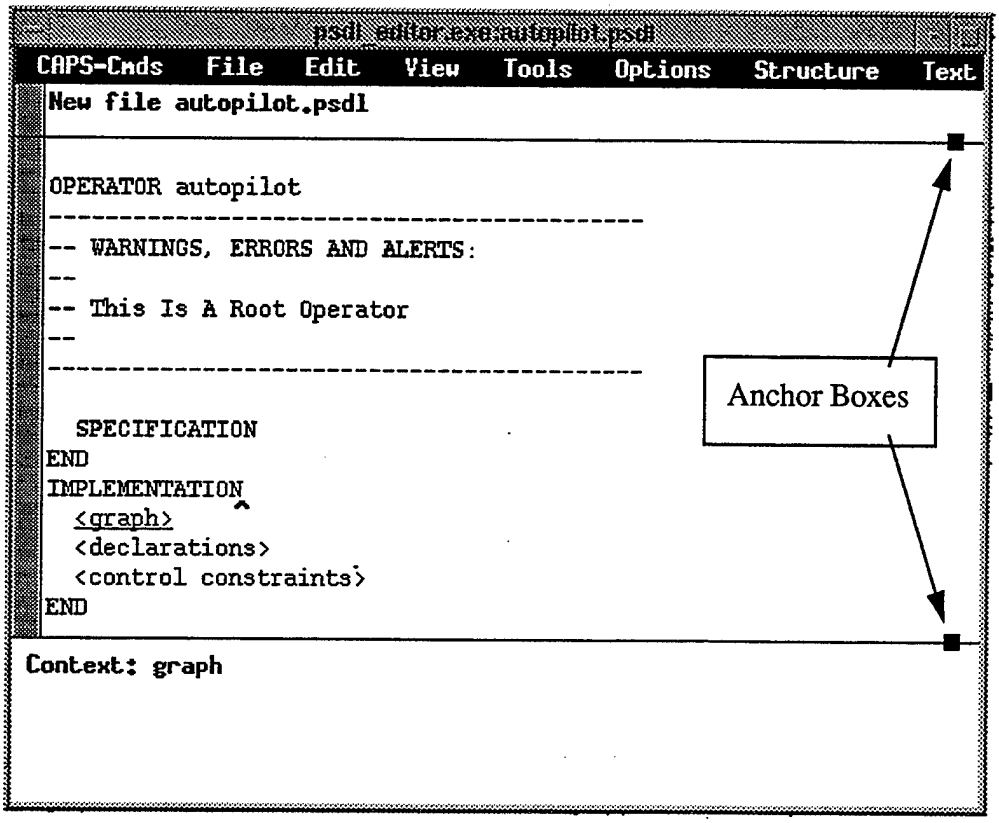


Figure 13. The CAPS Syntax Directed Editor for the Newly Created autopilot Prototype

Notice that the PSDL program stub for the **autopilot** prototype has a single operator named **autopilot**. This operator, in accordance with PSDL, has both a specification part and an implementation part and its name is the same as the name of the prototype. The specification is empty and the implementation is an empty PSDL implementation. PSDL allows operators to be implemented in either PSDL or Ada. Composite operators are implemented in PSDL and atomic operators in Ada. Since the root operator of all prototypes is composite, the implementation defaults to a PSDL implementation.

Upon return from the Graphic Editor, PSDL text that corresponds to the information entered in the Graphic Editor is automatically transferred to the Syntax Directed Editor. To transfer the graphic information to the Graph Viewer, click the left mouse button in the GRAPH portion of the **autopilot** operator. Similarly, to view the graphic representation of any composite operator, click the left mouse button in the operator's GRAPH section. Clicking the mouse in an atomic operator will cause the display of a blank graph viewer. This indicates (appropriately) that there is no PSDL (graphic) decomposition of the atomic operator.

There are three "panels" in the Syntax Directed Editor. These panels are separated by solid horizontal lines with solid black anchor boxes. The panel sizes can be changes by dragging the anchor boxes up or down with the left mouse button.

The CAPS Syntax Directed Editor is an attribute grammar-based editor and can be thought of as a convenient tool used to traverse the parse tree of a PSDL program. The designer moves about

the parse tree, entering information at the appropriate points. The attribute grammar gives the Syntax Directed Editor the ability to propagate information through the PSDL parse tree as necessary to maintain syntactic correctness and program consistency.

Full details regarding the Syntax Directed Editor functionality can be found in the CAPS User's Manual, but a brief summary of operation is provided here.

a. The "CAPS-Cmds" Pull-Down Menu

The "CAPS-Cmds" pull-down menu is the only pull-down menu in the Syntax Directed Editor that needs to be used during a PSDL editing session. The presence of the other pull-down menus is due to internal requirements of the editor synthesizer. These other pull-down menus ARE active, but their use IS NOT RECOMMENDED.

All of the pull-down menus in the Syntax Directed Editor are active. Use of any menu other than the "CAPS-Cmds" menu, however, is not recommended.

It has been mentioned before that the "edit-graph" selection from the "CAPS-Cmds" pull-down menu is the means by which the Graphic Editor is invoked.

The save commands in the "CAPS-Cmds" pull-down menu should be self-explanatory.

The "save-psdl" or "save-psdl-exit" command MUST be used at some point during a PSDL editing session in order to save clean PSDL code which can be retrieved again by the Syntax Directed Editor.

Be sure that there are no syntax errors in the PSDL code when executing "save-psdl-exit". The Syntax Directed Editor will beep and present a warning message when there are syntax errors in a PSDL program. Placeholders are not syntax errors, but rather indications of missing program text. If placeholders are present in the PSDL code upon saving the program, the Syntax Directed Editor will automatically correct or remove them. Upon saving a PSDL program, incomplete identifiers are automatically given the name "UNDEFINED_ID" and other placeholders are simply removed from the PSDL code.

Using the "save-psdl" or "save-psdl-exit" command on a PSDL program with syntax errors may corrupt the PSDL program file and cause ALL INFORMATION TO BE LOST.

When the Syntax Directed Editor is invoked, (provided that the prototype's PSDL program is syntactically correct) the types and operators in the prototype are alphabetically sorted, with types appearing before operators. At any time during a PSDL editing session, the PSDL types and

PSDL operators can be re-sorted using the "sort-psdl-components" command. The "exit" command in the "CAPS-Cmds" pull-down menu is an exit with no save, and requires verification. The "print" command will print a copy of the PSDL program to your default printer.

b. Traversing the PSDL Parse Tree

The CAPS Syntax Directed Editor displays and edits PSDL programs. PSDL programs are, of course, based on the syntax of PSDL. When using the Syntax Directed Editor, it is useful to remember that a PSDL program represents a particular configuration of a PSDL parse tree. The position of the cursor, while simply moving up, down, left and right on the display, really resides at a particular spot in the PSDL parse tree at any given time.

Depending on the location of the cursor in the parse tree, there are different allowable additions to the tree at that point. The allowable additions at any given point are displayed in the bottom panel of the Editor display. To insert a particular structure into the parse tree (the PSDL program) simply select the desired structure from the bottom of the Editor with the mouse. The selected structure will be inserted into the PSDL program. Text can be added to the PSDL program manually as well. When this "free text" method of text entry is used, it is necessary to terminate entry with a carriage return for proper results.

Free text entry into PSDL programs should be terminated with a carriage return.

To scan through the PSDL grammar from any point in the PSDL program, the return key can be used. This is a good way to get familiar with PSDL syntax.

The "Return" key on the keyboard can be used to scan through the PSDL grammar from any point in the PSDL program. This is a good way to get familiar with PSDL.

To indicate which PSDL structure is selected, the Syntax Directed Editor underlines all text associated with the selected structure. For example, if the cursor is positioned on the word SPECIFICATION, all relevant text is underlined. This indicates that selection of a structure automatically selects all substructures as well. Some useful keystrokes in the Editor include:

Control-Shift-K .....Delete structure (NO UNDO-BE CAREFUL!!!),
Control-Shift-P .....Move up one level in the parse tree,
Meta-l .....Display current line number,
Meta-g.....Go to specific line,
Control-h .....Delete previous character,
Control-v .....Advance one page,
Control-d .....Delete next character,

arrow keys .....move up, down, left, right,
delete or backspace.....Delete previous character.

The "Meta" key is either the ◊ key (on Sun Type 4 keyboards) or the "Alt" key. There is no "undo" command in the Syntax Directed Editor, so **caution is advised.** If irreversible errors occur during PSDL editing, use the "exit" command in the "CAPS-Cmds" pull-down menu and verify the exit in spite of altered editor buffers.

There is no "undo" command in the Syntax Directed Editor. Use the "exit" command from the "CAPS-Cmds" pull-down menu to exit without saving.

c. Warnings, Errors and Alerts in the Syntax Directed Editor

The Syntax Directed Editor provides the designer with some diagnostic information. This information appears both at the top of the editor and in the PSDL text. Generally, information in the top panel of the Editor will advise the designer of system-level events. The messages that appear at the top of the editor are scrolled within the upper panel. Consequently, old messages may be displayed at the top of the Editor.

Error messages that appear at the top of the Syntax Directed Editor are scrolled within the upper panel. They do not disappear.

Messages that appear in the PSDL text inform of inconsistencies, errors and alerts. For example, in the blank Syntax Directed Editor for the new **autopilot** prototype, the alert

```
-- This Is a Root Operator
```

is presented. See the CAPS User's Manual for a complete list of Editor warnings, errors and alerts.

Through manipulation via the Syntax Directed Editor, the PSDL program for the **autopilot** prototype is completed. A summary of the steps taken to complete the **autopilot** PSDL program (in no order of significance) is:

1) Data types were assigned to each data stream in the DATA STREAM portion of the implementation of root operator **autopilot**. Types are entered by positioning the cursor at the desired location, and then either selecting a predefined PSDL type from the bottom of the Syntax Directed Editor, or typing in the name of a user-defined type. After this has been done, clicking the left mouse button will cause the data type information to automatically propagate to the specifications of the atomic operators which have the

particular data stream as an input or output.

Data types for data streams must be entered in the DATA STREAM portion of the composite operator implementation. They are automatically propagated to lower level operators.

When a type is assigned to a data stream, click the left mouse button on the data stream declaration to ensure that the data stream type is propagated to lower level operators.

2) Two states were introduced to break the cycles in the **autopilot** data flow diagram. Data types and initial values were entered for each state. The state declarations appear in the specification part of the root operator.

Break all cycles in prototype data flow diagrams with state streams.

3) Control constraints were assigned to each operator. This includes periods and triggers in the **autopilot** prototype. Although no execution or output guards are used in this example, they are entered in a similar fashion in the CONTROL CONSTRAINTS portion of the operator implementation.
4) The user-defined PSDL data types **course_command_type** and **altitude_command_type** were created. Note that the implementations of these two components are in Ada and that the name of each implementation is the same as the type name. This convention is highly recommended.

User-defined PSDL data types must be entered as new PSDL components in the Syntax Directed Editor before they can be used in the DATA STREAMS portion of composite operator implementations.

A complete listing of the initial version of the **autopilot** PSDL program can be found in Appendix B. Notice that in the GRAPH portion of the PSDL program in Appendix B, there is a text description of the data flow diagram. This text does not appear in the Syntax Directed Editor. While in the Syntax Directed Editor, this text is replaced by the alert

```
-- see graph viewer for details --
```

and the graphical representation of the data flow diagram in the Graph Viewer.

D. The Complete PSDL Program

The generation of a complete PSDL program is the goal of PSDL editing. Appendix B can be studied as an example of such a program. The complete PSDL program has a single root level operator which is decomposed via the Graphic Editor. In the **autopilot** example, the root operator is the only composite operator. All other operators are atomic. In addition to the atomic operators, there are two user-defined data types. The root operator, the atomic operators, and the user-defined data types are all PSDL components. Every component in a PSDL program (types and operators) must have a specification part and an implementation part.

Every component in a PSDL program (types and operators) must have a specification part and an implementation part or translation errors will result.

The implementation of a component can be in either PSDL or Ada. If an operator is implemented in PSDL, it will have a data flow graph associated with it. If an operator is implemented in Ada, it will have an Ada implementation file associated with it. If either the specification or the implementation portion of a PSDL component is missing, the CAPS translator will complain.

V. PSDL Translation

A. Overview

The complete PSDL program is the basis for generation of compilable and executable prototype code. The CAPS translator is the tool that transforms the PSDL program into Ada code that implements all supervisory aspects of the prototype. The Ada file that is ultimately generated by CAPS is called the *supervisor module*. This Ada module is in a file called <prototype_name>.a. The basic structure of the supervisor module is shown in Figure 14.

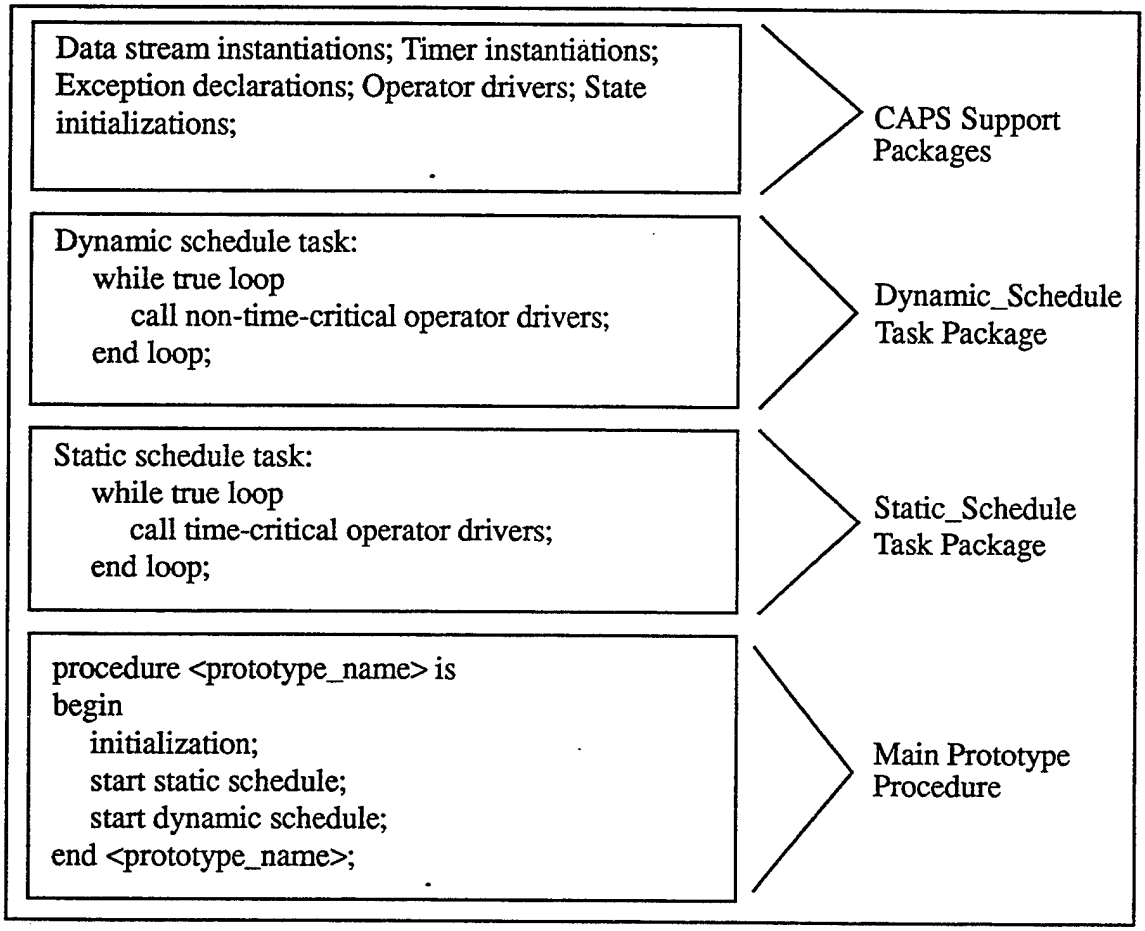


Figure 14. Basic CAPS Supervisor Module Structure

The CAPS translator generates the CAPS support packages and the main prototype procedure. The translator DOES NOT generate Ada implementation files for atomic operators or user-defined data types. The translator generates code which implements data streams, execution guards, output guards, operator triggers and timers. All of this code becomes part of the supervisor module. See Appendix A for a complete listing of the **TEMP_CONTROLLER** supervisor module.

The remaining portions of the supervisor module (the static and dynamic schedule packages) are created by the CAPS scheduler. Scheduling CAPS prototypes is discussed in the next chapter.

B. Translator / User Interaction

User interaction with the translator is ideally short and simple. Note that modifications made to the PSDL program in the Syntax Directed Editor are not reflected in the associated files until a save is performed. For this reason, even though the "Translate" command is active in the CAPS "Exec Support" pull-down menu, the designer must save the PSDL program prior to translation.

The "save-psdl" or "save-psdl-exit" command must be used in the Syntax Directed Editor prior to translation in order to provide the translator with the most recent version of the PSDL program.

The PSDL program is pre-processed prior to being sent to the translator. Occasionally errors will be detected at this point. The most frequent of which is a missing implementation portion in the PSDL program for an operator or a user-defined data type.

Every component in a PSDL program must have a specification part and an implementation part.

Creation of the Ada implementation packages for atomic operators and user-defined data types that are compatible with the automatically generated supervisor file is often the designer's most difficult task. Knowledge of five **very important** CAPS conventions will help in this matter:

Understanding the following five CAPS conventions is critical in order to create Ada implementation files, packages and procedures that are compatible with CAPS-generated code.

1) The supervisor module contains procedure calls to the Ada implementation packages. The parameters of these procedure calls are name-associated and correspond to the names of the operator's input and output streams. Because CAPS uses name association, the order of the formal parameters in the Ada implementation procedure does not matter, but the names of the parameters must match the names of the data streams precisely.
2) The mode of formal parameters in the Ada implementation files for atomic operators should match the corresponding data stream (input streams: mode "in", output streams: mode "out", streams which are both input and output: mode "in out").
3) The name of the procedure that implements an atomic operator must match the name of the operator precisely.
4) Atomic operator implementation files and user-defined data type implementation files must be named <prototype_name>.<component_name>.a. Note that <component_name>

must also be specified in the component's implementation section ("IMPLEMENTATION ADA <component_name>") of the PSDL program. CAPS uses the name given in the PSDL program to derive expected package names.

5) Each Ada implementation file for atomic operators must contain a package specification and a package body. Ada implementation files for user-defined types must contain a package specification, and need to have a package body only if the type has operators associated with it. **The name of Ada implementation packages must be <component_name>_PKG.**

Here are two examples of Ada implementation files from the **autopilot** prototype. The first is the file called autopilot.altitude_command_type.a, which implements the user-defined PSDL type called **altitude_command_type**, and the second is the file called autopilot.correct_altitude.a, which implements the operator **correct_altitude**. Note the package constructs in both files and the procedure name in autopilot.correct_altitude.a. Also note that the parameter modes in procedure **correct_altitude** correspond to the data streams in the data flow diagram (Figure 12).

```
-- brief sample code (Ada implementations of a data type and an operator --

-------------------------------------------------------------------
-- Unit : autopilot.altitude_command_type
-- Prototype : CAPS autopilot
-- Date : June '94
-- Author : Jim Brockett
-- Compiler : SunAda
-- Description : altitude_command_type Ada implementation
-------------------------------------------------------------------

package altitude_command_type_PKG is

      type altitude_command_type is (level, up, easy_up, easy_down, down);

end altitude_command_type_PKG;

-------------------------------------------------------------------
-- Unit : autopilot.correct_altitude
-- Prototype : CAPS autopilot
-- Date : June '94
-- Author : Jim Brockett
-- Compiler : SunAda
-- Description : correct_altitude Ada implementation
-------------------------------------------------------------------

with altitude_command_type_PKG; use altitude_command_type_PKG;

package correct_altitude_PKG is

procedure correct_altitude(actual_altitude : in Integer;
                           desired_altitude : in Integer;
                           altitude_command : out altitude_command_type);

end correct_altitude_PKG;
```

```
package body correct_altitude_PKG is

procedure correct_altitude(actual_altitude : in Integer;
                           desired_altitude : in Integer;
                           altitude_command : out altitude_command_type) is

      nominal_difference : Constant Integer := 5;
      small_difference : Constant Integer := 50;
      difference : Integer := actual_altitude - desired_altitude;

begin

      if difference > small_difference then altitude_command := down;
      elsif difference < -small_difference then altitude_command := up;
      elsif difference > nominal_difference then altitude_command := easy_down;
      elsif difference < -nominal_difference then altitude_command := easy_up;
      else altitude_command := level;
      end if;

end correct_altitude;

end correct_altitude_PKG;
```

These two examples, along with those provided in Appendix A for the **TEMP_CONTROLLER** prototype, should provide sufficient guidance for the development of similar Ada implementation files.

C. Results of Translation

As mentioned before, the CAPS translator generates a portion of the prototype's supervisor file. The remainder of the supervisor file is generated by the CAPS scheduler. One final reminder: the translator DOES NOT create the Ada implementation files for atomic operators or user-defined data types.

The CAPS translator DOES NOT create Ada implementation files for atomic operators or user-defined data types.

VI. Prototype Scheduling

A. Overview

The CAPS scheduler takes the timing information from the PSDL program, determines schedule feasibility, and then creates code to implement the schedule. CAPS currently generates a schedule for a single processor hardware configuration, with non-preemptable time-critical operators. Time-critical operators are executed in the CAPS *static schedule* and non-time-critical operators are executed in the CAPS *dynamic schedule*. The static and dynamic schedules are implemented as Ada tasks.

Upon scheduling a prototype, CAPS provides diagnostic information about the schedule it has created. A portion of the diagnostic information for the **autopilot** prototype is shown in Figure 15. Complete diagnostic information for the **autopilot** prototype is provided in Appendix C.

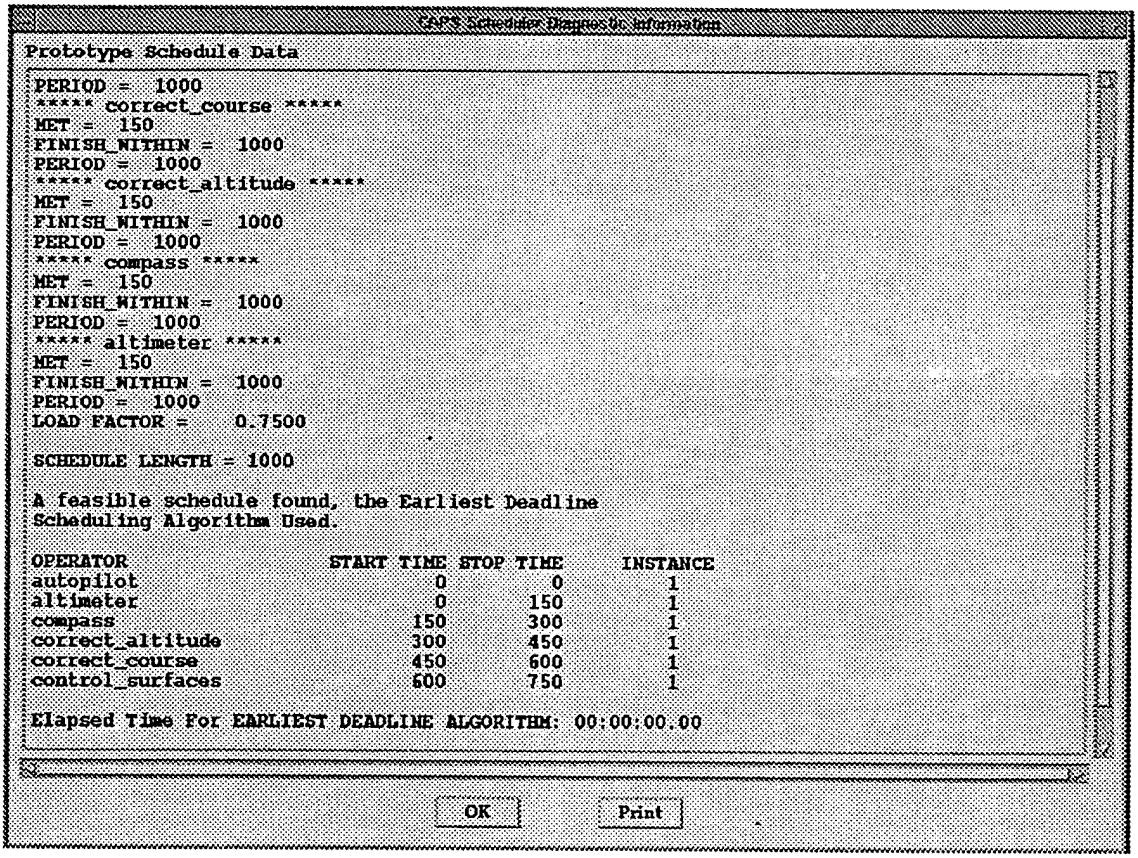


Figure 15. CAPS Scheduler Diagnostic Output

This display provides the designer with valuable raw schedule data, including each time-critical operator's start and stop times. The times indicated are in milliseconds. Also provided are timing constraints for all time-critical operators, the schedule length, and the static schedule load factor. Static schedule load factor and schedule length are discussed in Sections C and E of this chapter, respectively.

B. The Static and Dynamic Schedules

Time-critical operators are given higher priority than non-time-critical operators in CAPS by assigning a higher priority to the static schedule task than to the dynamic schedule task. The static schedule is a carefully scripted, higher priority task which invokes all time-critical operator drivers. Each time-critical operator is given a time-slice equal to its maximum execution time. If a time-critical operator does not utilize its entire maximum execution time, the remaining time is allocated to the next non-time-critical operator in the dynamic schedule. If a time-critical operator exceeds its maximum execution time, an alert is printed to the standard prototype output (text) and the schedule is re-calibrated so that the next time-critical operator gets the full amount of time allocated to it. The dynamic schedule calls all non-time-critical operator drivers in topological-sort order. The Ada packages that implement the static and dynamic schedules are added to the output of the translator to create the entire supervisor module for a prototype.

C. Scheduling Feasibility Factors

In addition to the relationships among timing constraints provided in Chapter III, a designer should be aware of other scheduling feasibility factors when designing a prototype.

One issue is the percentage of static schedule time (what portion of the schedule length) the time-critical operators will use. This value is called the *static schedule load factor* and is computed as follows:

$$\text{static schedule load factor} = \Sigma \text{ (maximum execution time / period)}$$

where the summation is over all time-critical operators. The static schedule load factor MUST be less than 1.0 for the schedule to be feasible on a single processor. If the static schedule load factor is between 0.5 and 1.0, feasibility depends on the details of the timing constraints, and a schedule may not exist in this case. A schedule always exists if the static schedule load factor is less than 0.5.

If all time-critical operators use their full maximum execution time to execute, then the static schedule load factor represents exactly the CPU utilization of the statically scheduled operators during prototype execution. If any time-critical operator uses less than its allotted maximum execution time for execution, non-time-critical, dynamically schedule operators will execute during the unused portion of time. The CAPS static schedule is a carefully scripted sequence of time-critical operator driver calls. Time-critical operators do not utilize any time "left over" time from other time-critical operators. This extra time is used exclusively by the non-time-critical, dynamically scheduled operators, in accordance with the dynamic schedule's linear list of driver calls.

static schedule load factor = Σ (maximum execution time / period)
The summation is over all time-critical operators and MUST BE LESS THAN 1.0.

The significance of these concepts becomes apparent during system design and modification. To illustrate this, suppose that we wish to modify some of the maximum execution times of the

operators in the **autopilot** prototype. We wish to change the maximum execution times of the **correct_course** and **correct_altitude** operators from 150 MS to 300 MS. This is represented in Figure 16.

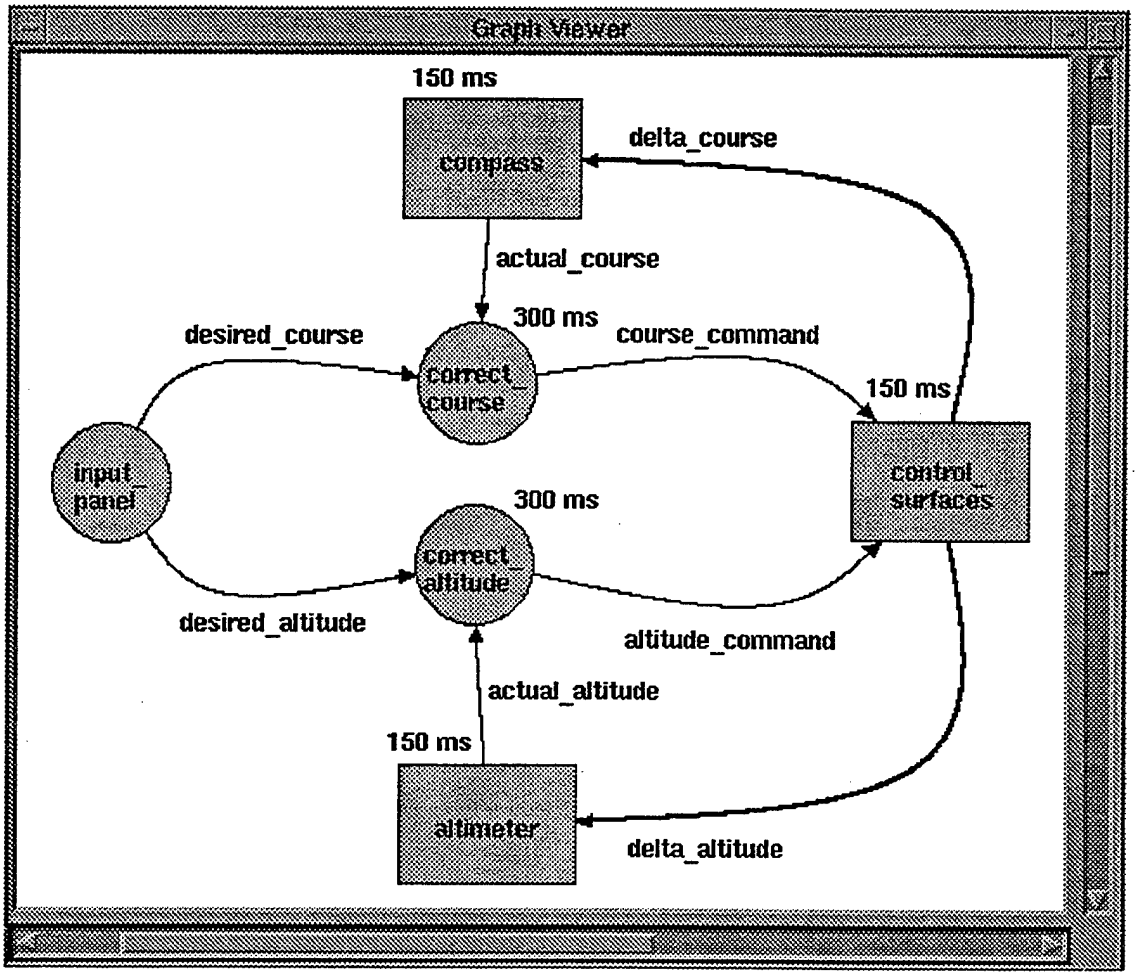


Figure 16. Modified autopilot Graph

When we try to schedule this version of **autopilot**, the scheduler diagnostic display shows:

LOAD FACTOR ERROR raised.
LOAD FACTOR = 1.0500
LOAD FACTOR must be less than 1.0.
Please modify timing specifications of the design.

The static schedule load factor of a prototype is equal to the sum of maximum execution time divided by period, for all time-critical operators. This sum must be less than 1.0. In this case, we have exceeded this value. The period of each time-critical operator in **autopilot** is 1000 MS (see Appendix B for the **autopilot** PSDL program), and the static schedule load factor has become 1050/1000 = 1.05. The **autopilot** example is simple and straightforward. This is partly due to the identical periods of all time-critical operators.

Suppose, now, that instead of doubling the maximum execution times of the two operators (**correct_course** and **correct_altitude**), we halve their periods from 1000 MS to 500 MS (while returning their maximum execution times from 300 MS to 150 MS). The implementation portion of the root operator (**autopilot**) becomes:

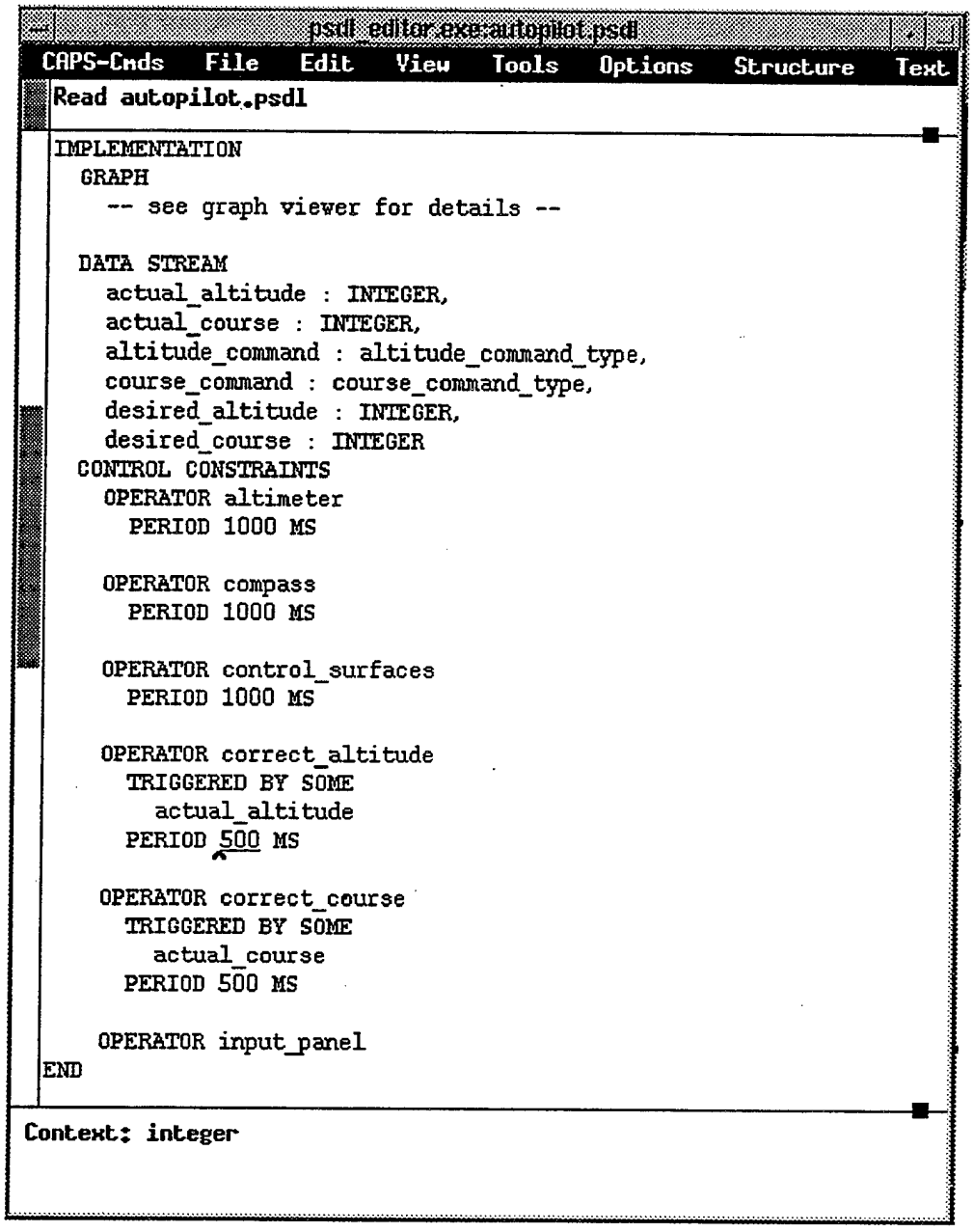


Figure 17. Modified autopilot Implementation

Again, in this configuration, when we schedule the **autopilot** prototype, we get: . . .

LOAD FACTOR ERROR raised.
LOAD FACTOR = 1.0500

LOAD FACTOR must be less than 1.0.
Please modify timing specifications of the design.

This is because in spite of halving the periods of the two operators, their contribution to the static schedule load factor is still 0.3 each.

300/1000 = 0.3 in the first case, and 150/500 = 0.3 in the second case.

Modifications made to operator periods and maximum execution times directly impact the static schedule load factor.

If we modify further, and make the maximum execution times of the two operators 100 MS rather than 150 MS (their periods are still 500 MS), we get the following Graph Viewer display:

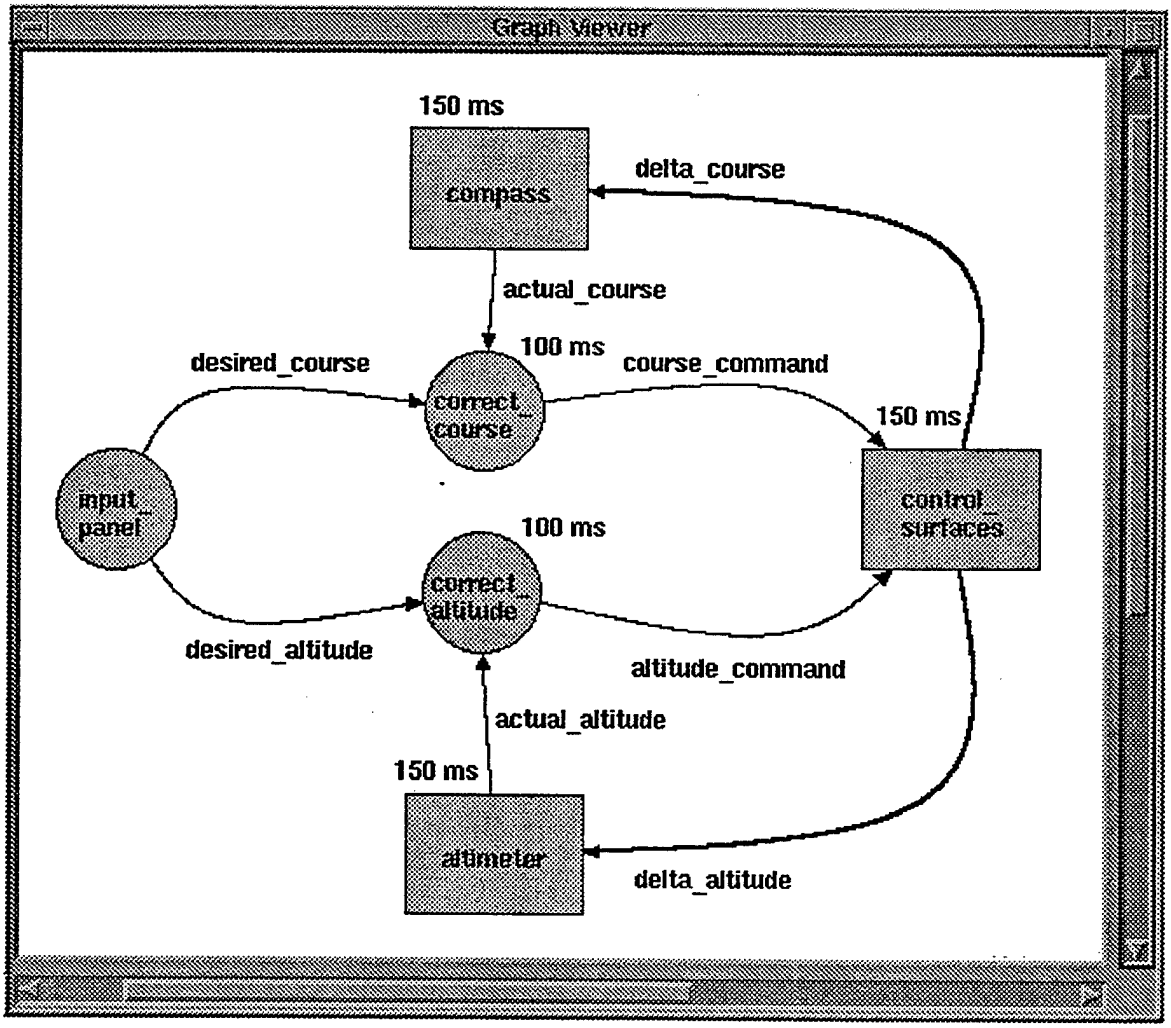


Figure 18. Modified autopilot Graph

The static schedule load factor returns to below 1.0, and scheduling is possible. The scheduler diagnostic output is shown in Figure 19.

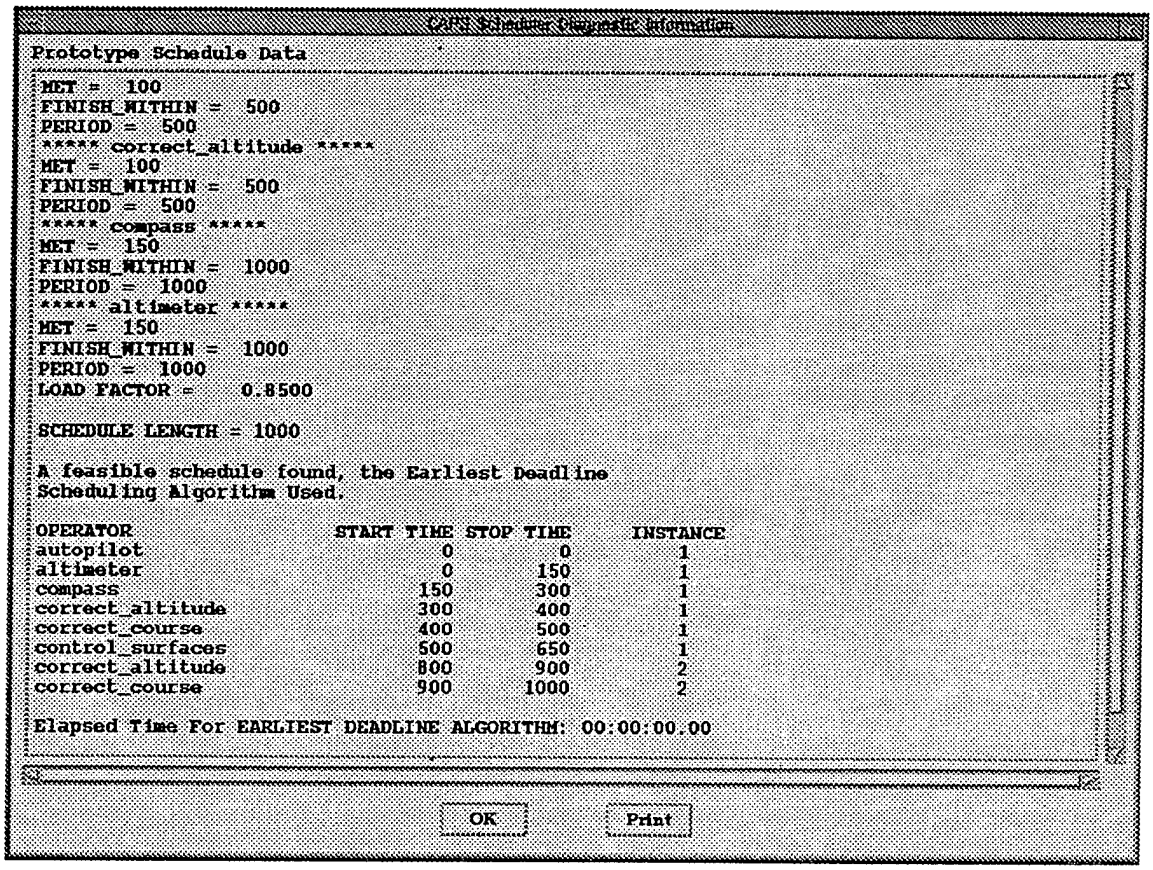


Figure 19. Modified autopilot Scheduler Diagnostic Output

Note that, as a result of their altered period, the two operators (**correct_course** and **correct_altitude**) are now called twice during each iteration of the static schedule. This reflects their period of 500 milliseconds. Modifications to prototype timing constraints, similar to those presented here, form the basis of the CAPS prototype modification and evaluation cycle.

Another timing constraint which can impact the feasibility of scheduling a prototype is data stream latency. The latency of a data stream is a lower bound on the amount of time required for transmission of data along that stream. This allows a designer to model a multi-processor target environment where data is transmitted along a time-consuming network path. The impact of data stream latencies is a delay (equal to the latency value of a data stream) in the earliest allowable start time of the consuming operator. The effects of this delay on schedule feasibility are clear. When latencies are introduced into prototypes, the scheduling diagnostic information provided by CAPS becomes even more valuable.

Data stream latencies are entered using the Graphic Editor in a manner similar to that used for entering operator maximum execution times. Use the "Properties" tool in the Graphic Editor to enter data stream latencies.

D. Decomposition Considerations

When an operator is decomposed into a PSDL implementation, it is called a composite operator. If a composite operator has timing constraints, the decomposition (implementation) of the operator must consider these constraints. This section addresses some very basic concepts associated with the decomposition of time-critical operators in CAPS.

1. Periods

In CAPS, assigning a period to a composite operator has essentially no effect. This is because it is the expanded, single-level prototype graph that is evaluated by the CAPS scheduling algorithms. However, if a designer wishes to assign a period to a composite operator, it is reasonable to expect that the period of each operator in the decomposition must be equal to the period of the parent operator. If the period of a sub-operator is greater (less frequent sub-operator firing) than that of its parent, that would possibly cause violation of the sub-operator's period. The sub-operator may fire too often. Similarly, if the period of a sub-operator is less than that of its parent (more frequent execution), violation of the parent operator period occurs. Thus, though not enforced by CAPS, it is recommended that periods not be assigned to composite operators.

2. Maximum Execution Times

The sum of maximum execution times of the sub-operators cannot exceed the maximum execution time of the parent operator. If this condition is violated, the parent operator will fail to execute within its maximum execution time in a single processor environment. CAPS currently provides default maximum execution time decomposition appropriate to single processor hardware architectures.

> **The sum of maximum execution times of operators in the decomposition of a composite operator cannot exceed the maximum execution time of the composite operator itself.**

In the absence of designer-provided information, the maximum execution time of a composite operator is equally divided among its children. For example, if a composite operator has a maximum execution time of 300 MS, and is decomposed into three atomic operators, the maximum execution time of each of the three atomic operators defaults to 100 MS. This decomposition convention is used regardless of the configuration of the decomposition.

3. Other Timing Constraints

As mentioned above, when a schedule is generated for a prototype, the information dictating schedule feasibility is taken from an equivalent expanded, single level PSDL graph. Prior to scheduling, all composite operators are expanded until only atomic operators remain. The prototype is thereby represented as a tree of only two levels. The top level consists only of the root

operator, and the second level consists only of leaves which are the prototype's atomic operators. It is these atomic operators whose timing constraints are evaluated. Therefore, it may be desirable to assign timing constraints only to the atomic operators in a prototype. If a designer wishes to assign timing constraints to composite operators, conventions similar to those outlined above should be followed.

E. Schedule Length

Computation of the static schedule for a prototype is a non-trivial task. The problem in general is NP-hard. However, experimentation has shown that simple heuristic scheduling algorithms (such as earliest deadline first) perform very well for typical CAPS prototypes. Thus, the scheduling problem is manageable. A basic understanding of some scheduling concepts used in CAPS will enable a designer to build more efficient and effective prototypes.

In the **autopilot** example, the schedule length is 1000 MS. This is rather obvious, because in the initial version, all assigned periods were 1000 MS. In the revised version, two operators had periods of 500 MS, and fired twice as often as those operators whose periods remained 1000 MS. In the general case, a prototype's schedule length is computed as the least common multiple (LCM) of the periods of all time-critical operators.

The schedule length of a CAPS prototype is computed as the LCM of the periods of all time-critical operators.

This can be demonstrated by changing the periods of the two modified operators (**correct_altitude** and **correct_course**) in the **autopilot** prototype from 500 MS to 750 MS. When we do so, the following schedule diagnostic information is generated.

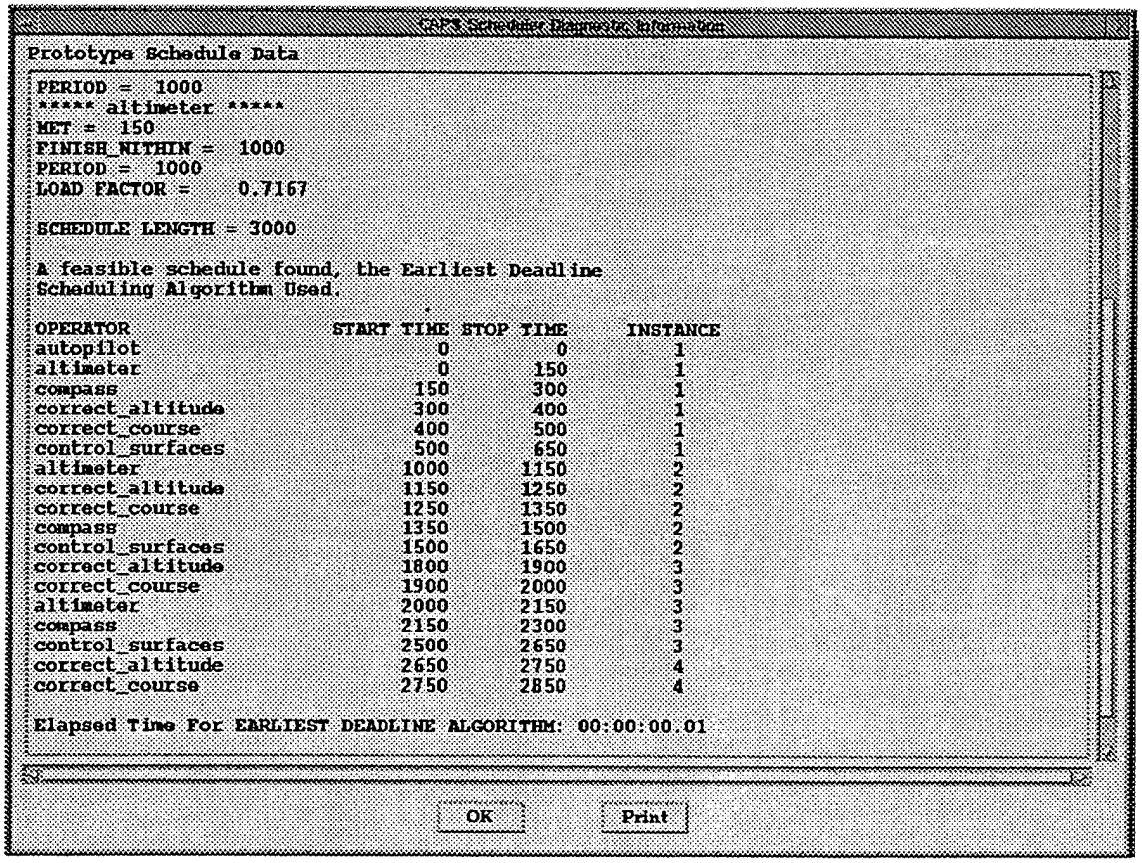


Figure 20. Modified (again) autopilot Scheduler Diagnostic Output

Notice that the schedule length is now 3000 MS (because LCM(750,1000)=3000). Also note that the two operators whose period is 750 MS (**correct_course** and **correct_altitude**) now fire four times to every three times that the operators with period=1000 MS fire. This is how CAPS implements relative period differences in time-critical operators.

The CAPS scheduler will generate schedules for any combination of operator periods, but prototypes with relatively compatible periods (lower LCMs) are easier to schedule and compile more quickly because the length of the static schedule code is proportional to the schedule length, given by the LCM of periods. Using several different periods that are not multiples of a sizable common factor can make the schedule length grow rapidly. For example,

LCM(370,630,512)=5,967,360; while LCM(400,600,600)=1200.

Keeping the LCM of the periods of time-critical operators LOW makes prototype scheduling easier and reduces compile time.

F. Equivalent Periods

Every time-critical operator in a CAPS prototype must have a period, a data trigger or both. In order for CAPS to generate a static schedule via the LCM principle described above, all time-critical operators must have periods. For time-critical sporadic operators which are not explicitly assigned periods, CAPS generates periods called *equivalent periods*. The equivalent period of a sporadic operator is not greater than:

minimum [(MRT - MET), MCP].

Recall that MRT is maximum response time and MCP is minimum calling period. The current CAPS scheduler requires that maximum execution time be assigned to time-critical sporadic operators. If maximum response time and/or minimum calling period are not provided, the scheduler will automatically generate these values. If they are provided, of course, the scheduler will use them. Details regarding the generation of these values are beyond the scope of this tutorial. Heuristics in the CAPS scheduler attempt to make all generated values as reasonable as possible. DO NOT declare maximum execution times for non-time-critical operators, as they will be classified by the scheduler as time-critical and the schedule can become very long.

A designer can take maximum advantage of the calculation of equivalent periods when designing a prototype. Remembering that the static schedule length is the LCM of the periods of all time-critical operators, it is wise to assign values for operator periods, maximum response times, maximum execution times and minimum calling periods which result in a nice tight schedule. Additionally, keeping these concepts in mind will save a designer hours of aggravation and frustration when "tweaking" a prototype's timing constraints.

Keeping the concepts described above in mind will save a designer hours of aggravation and frustration when "tweaking" a prototype's timing constraints.

G. Results of Scheduling

Upon successful scheduling of a prototype, generation of the supervisor module is completed. The dynamic schedule package and the static schedule package are added to the supervisor file. This file is Ada compilable, and expects to have available to it all of the implementation packages for the prototype's atomic operators and user-defined data types. Remember to adhere to the conventions listed in Chapter V, Section B when building the Ada implementation packages, or compile errors will result.

The prototype supervisor module will not compile until all of the Ada implementation files for atomic operators and user-defined data types are available.

VII. Interface Integration

A. Overview

In order for a prototype to be of any use, a designer must be able to observe the system's execution. If the system is an embedded system, the end product may not have any "user-interface" of its own. Despite this fact, an interface of some sort must be integrated into the prototype in order to evaluate the prototype's performance and suitability.

CAPS has built-in diagnostics which monitor prototype execution and advise the designer when maximum execution times have not been met, when data stream I/O constraints have been violated (overflow and underflow) and when (and where) exceptions occur.

In addition to such fundamental system timing constraint monitoring, a designer may wish to observe internal system values, run simulations, provide initial input values, validate hardware simulation procedures, etc. In order to facilitate such interaction, a suitable interface must be integrated into the prototype. Since such an interface may not be part of the final system, and will require CPU resources, its introduction to a real-time system prototyping environment is somewhat cumbersome.

CAPS currently uses TAE+ [TAE93] as a graphical user-interface generator. TAE+ allows a designer to quickly build high quality window-based user-interfaces for prototypes. Window management code is automatically generated by TAE+ and modified by a designer to suit the purposes of a particular prototype. Critical prototype values can be elegantly displayed in a variety of formats. List selections, push button activators and window-to-window connections can be easily programmed. Graphic and animation features provide sufficient power for system simulation purposes and visual enhancement.

TAE+ generated code can be integrated into CAPS code in at least two ways. These two methods, as well as a Text Interface option are discussed below.

B. Text Interface

If the critical elements of a prototype relate primarily to the feasibility of procedure executions under a given set of constraints, or if the required display of internal values is limited, it may be possible to evaluate the prototype's performance without an elaborate interface. Simple text output commands from within the implementation procedures of the atomic operators may be sufficient for prototype evaluation. In such a case, there is no need for development of an extensive window-based user-interface. It is also possible to have simple terminal keyboard interactions built into the implementation procedures of atomic operators. Such interaction could be suitable for data entry, menu selection and possibly simulation control.

The **TEMP_CONTROLLER** prototype presented in Chapter II is an example of a prototype with a simple text interface. In that example, the status of the heater and cooler (ON or OFF), along with the current temperature are continuously displayed in the CAPS execution window. There is no user input to this prototype. The input to the temperature sensor is simulated in the implementation of the **Sensor** operator.

The initial version of the **autopilot** prototype is another example of a prototype with a simple text interface, and allows input of initial desired course and altitude. Unfortunately, there is no easy way to provide text input while the prototype is running. This is because an atomic operator

with an Ada "get_line" statement will wait until input has been provided before continuing its execution. This results in prototype halting and timing errors if the operator with the "get_line" statement is time-critical. Thus, it is reasonable to provide real-time prototypes with INITIAL input information via simple text interfaces (i.e input before prototype execution begins). Text input of information while the prototype is running is less elegant and must be done in non-time-critical operators. Note, however, that if a non-time-critical operator is waiting for user input, other non-time-critical operators CANNOT fire. Input of initial values can be accomplished by creating a main execution block in the package body of the appropriate operator.

When text interfaces are used for prototype interaction, it is recommended that user INPUT be limited to INITIAL values only.

The file autopilot.input_panel.a is listed in Appendix D, and is an example of the use of text input for initial values. In this example, initial values are provided to the **autopilot** prototype for course and altitude.

The obvious drawback to this approach is its inherent incompatibility with real-time systems. By their very nature, real-time systems operate continuously. As a result, the text output from prototypes often results in difficult to read scrolling. It is very hard to watch this scrolling text and garner much meaningful information. Thus, having a window-based interface, where prototype information is updated in a portion of a window, is more appropriate for the real-time paradigm.

C. Graphic Interface

1. Time-Critical Operator Approach

In the event that more elaborate interaction with a prototype is desired, facilities for that interaction must be integrated into the prototype. One method that has been used with success in CAPS is assigning display and interaction procedures to a vertex in the PSDL graph, and then treating it just like any other operator. When using this method, and TAE+ for interface code generation, the operators with TAE+ code must be non-preemptable time-critical operators. This is due to the fact that if TAE+ low-level window management commands are being executed in an Ada task and are preempted by operators from a higher priority Ada task, the TAE+ procedure occasionally freezes execution. Since this is undesirable, such operators should be time-critical.

In the current version of CAPS, window-based interface operators must be non-preemptable, and thus, time-critical.

This requirement affects the length of the static schedule, but not the general prototype evaluation process. If static schedule length is critical, then the time used for display and other user-interface purposes can be subtracted to get the actual required static schedule length for the real-

time system being built. This method of interface integration allows a designer to treat the interface as a PSDL operator. Values are passed to and from the interface via PSDL data streams. A problem with this method is the relatively large maximum execution time required for the window-based display operations. Observation and experimentation has shown that operators with TAE+ code in their implementation require maximum execution times in the vicinity of 200 milliseconds.

Consider the following modification to the **autopilot** prototype.

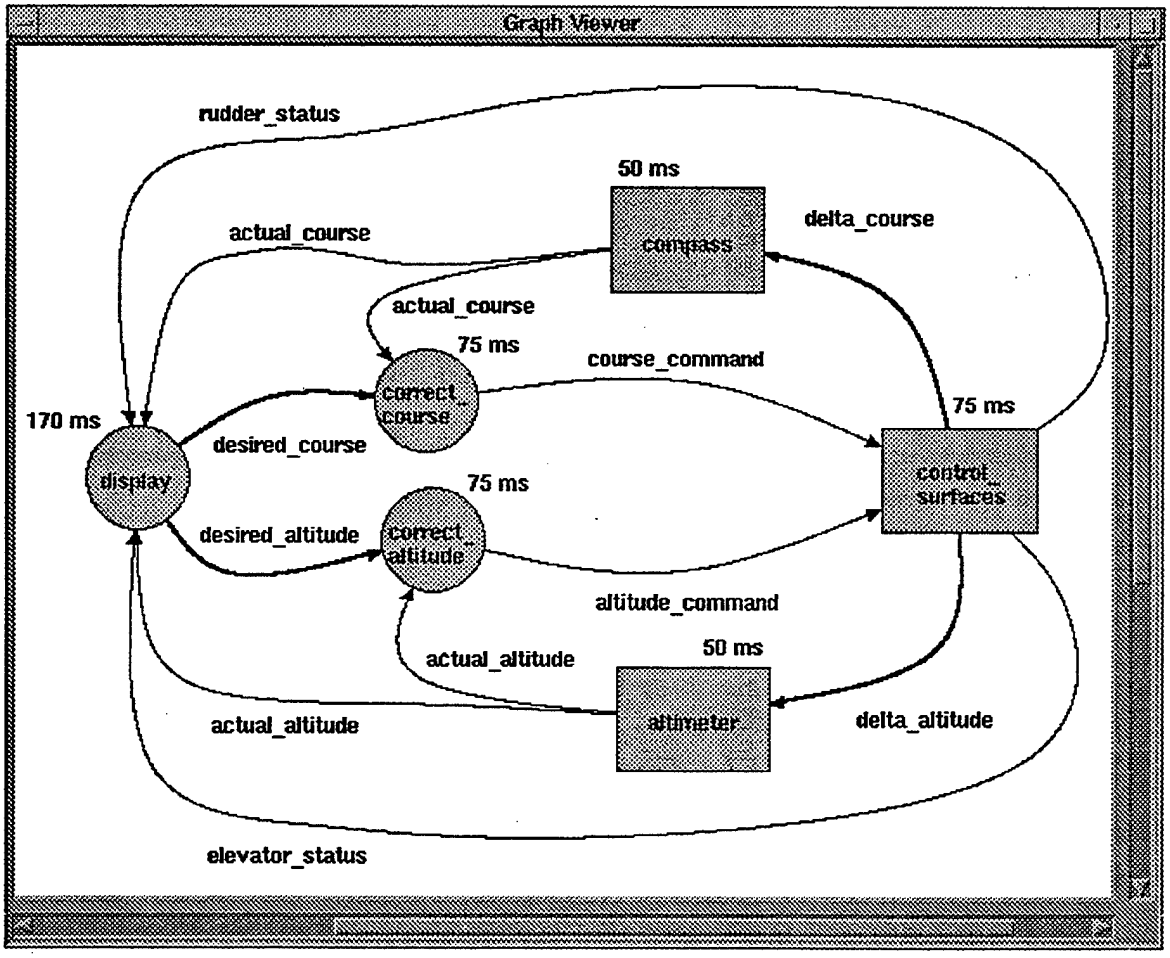


Figure 21. The autopilot Prototype With Built-in Graphic User-Interface

We have introduced a new operator called **display**. This operator will serve as both the input and the output mechanism for the **autopilot** prototype. The PSDL specification for this operator is generated just like any other CAPS operator. The difference is only in the Ada implementation. Notice that many of the timing constraints have been changed. Also, the period of all operators in this version of **autopilot** is 500 MS, resulting in a static schedule load factor of 495/500=0.99.

At this point, we will briefly digress and explain some basic properties of TAE+ generated code. Details about the use of TAE+ are beyond the scope of this tutorial, and the reader is referred to [TAE93]. This tutorial makes no attempt to describe the use of TAE+ except where it directly impacts the generation of code for operator implementations.

TAE+ applications are intended to be stand-alone window-based interface systems. As such, they are implemented as continuous looping programs. The main TAE+ loop is a loop which continuously monitors user input to the interface, such as mouse clicks or text entry. Upon detection of input, the TAE+ code calls an appropriate event handler. It is the event handlers in the TAE+ code which are modified by an application developer to do useful work (e.g. open a new window, send some information to a database, activate a procedure, display the contents of a file, etc.). CAPS uses the TAE+ event handlers similarly. However, to use TAE+ code as a CAPS operator, basic structural changes must be made.

To use TAE+ generated code as a CAPS operator, basic structural changes must be made.

1) First and foremost among these changes is the removal of the continuous event loop. The code in the TAE+ event loop becomes a linear procedure which will be called by the prototype's static schedule. It becomes, in essence, a small part of a larger loop.
2) Second, the TAE+ code needs to be converted into a callable procedure with a name corresponding to that specified in the PSDL program (the name of the operator).
3) Third, the entirety of code generated by TAE+ needs to be encapsulated in an Ada package conforming to CAPS integration conventions (named <operator_name>_PKG, see Chapter V, Section B).
4) Finally, the file name of the generated code needs to be changed to satisfy CAPS conventions (<prototype_name>.<operator_name>.a). CAPS automatically saves the raw TAE+ code as <prototype_name>.RAW_TAE_INTERFACE.a.

As a direct result of the third and fourth steps listed above, it is incumbent upon the designer to use the TAE+ code generation option which generates a single Ada file.

The "single file" Ada code generation option must be used in TAE+ to ensure compatibility with CAPS.

Careful examination of the code generated by TAE+ is **highly recommended.**

In addition to the basic structural changes listed above, there are a number of very detailed, yet very important changes that must be made to the TAE+ code to make it usable by CAPS. The specific changes made to the **autopilot display** code are described in detail here. The order of these steps is not important, as they must ALL be made in order for the code to work. The steps are presented here in roughly the order in which they appear in the source listing of the modified TAE+ file which is now called "autopilot.display.a". This file is listed in its entirety in Appendix E, and should be consulted in association with the following steps. Additionally, in the listing in Appendix E, step numbers appear (as "`-- STEP X`") in the code at locations that correspond to

the following 31 steps. While this list may appear a bit daunting, **the end results are well worth the effort of understanding these 31 steps.**

1) Rename the file generated by TAE+ from <prototype_name>.RAW_TAE_INTERFACE.a to <prototype_name>.<component_name>.a

2) Add an appropriate header to the file. In autopilot.display.a, in addition to adding a header, all of the TAE generated comments have been removed from the beginning of the file.

3) Add "`with`" and "`use`" statements as necessary for user-defined PSDL type packages.

4) Comment out the "`procedure <prototype_name> is`" statement. Remember that TAE+ generates code that is intended to be used as a stand-alone application. In CAPS, we must put the TAE+ generated main procedure inside of an Ada package.

5) Change the name of the TAE+ generated package to <component_name>_PKG. This change must be made in four places: the beginning and end of the package specification, and the beginning and end of the package body.

6) Inside of the newly named package specification, the main procedure must be declared. In the **autopilot** prototype the added code is:

```
procedure display(rudder_status: in rudder_status_type;
                  actual_course: in INTEGER;
                  desired_course: out INTEGER;
                  desired_altitude: out INTEGER;
                  actual_altitude: in INTEGER;
                  elevator_status: in elevator_status_type);
```

Notice that the parameter modes correspond to the data streams; mode "`in`" for input streams, mode "`out`" for output streams. If a stream is both an input and an output of the operator, the parameter mode should be "`in out`".

7) In the declarations of the event handler procedures, add formal parameters as necessary to pass input and display information to and from the rest of the prototype.

8) Change the name of the package in the package specification's "`end`" statement (see step 5).

9) Add "`with tae;`" before the newly named package body. Notice that in autopilot.display.a, we have also added "`with text_io`".

10) Comment out the TAE+ generated "`use <prototype_name>_support;`" statement.

11) Move the renamed "`package body`" statement above the declarations of TAE-specific variables and above the "`use tae.tae_misc;`" statement. This makes these TAE-

specific variables visible to the new "display" procedure.

12) After the TAE-specific variable declarations, add any application-specific variable declarations. In the case of the **autopilot** prototype, we have added

```
local_desired_course: INTEGER;
local_desired_altitude: INTEGER;
```

These two variables implement atomic operator state information for the **display** operator.

13) Comment out (or delete) the original "`package body`" statement (the one moved up in step 11).

14) The `initializePanels` procedure requires no modification.

15) In the **autopilot** prototype, there are two event handlers for user input. These procedures assign values to the **desired_course** and **desired_altitude** data streams. For each of the two event handlers, we have made three modifications:

a) add a formal parameter to send the input information to the rest of the prototype,
b) comment out the TAE+ generated `put` and `put_line` commands,
c) assign the added parameter the value of the TAE+ input variable (value(1)), and check this value for validity.

This is the critical step for getting information **FROM** a TAE+ window. Step 24 in this list documents the critical step for getting information **TO** a TAE+ window. Appropriate modifications to all event handlers must be made. Note that TAE+ generates event handler procedures only for those items declared (from within TAE+) to generate events (see [TAE93]).

16) Move the package "`end`" statement (with the package appropriately renamed) to the end of the file. In the **autopilot** example, the renamed "`end`" statement has been commented out in its original location and copied to the end of the file.

17) Insert the heading of the implementation procedure:

```
procedure display(rudder_status: in rudder_status_type;
                  actual_course: in INTEGER;
                  desired_course: out INTEGER;
                  desired_altitude: out INTEGER;
                  actual_altitude: in INTEGER;
                  elevator_status: in elevator_status_type) is
```

18) Comment out all of the initialization code and move it to the package body's main execution block (see step 29). The first line of the new procedure should be:

```
tae_wpt.Wpt_NextEvent (wptEvent, etype); -- get next event
```

Notice that this step has also commented out the original "`while`" loop generated by TAE+. DO NOT copy this loop to the package body's main execution block in step 29.

19) We now need to modify what used to be called the TAE+ EVENT_LOOP. Comment out the following line:

```
text_io.put ("Event: WPT_PARM_EVENT, ");
```

20) In the calls to the event handler procedures, actual parameters need to be added to match the formal parameters added in steps 7 and 15. This is done in two places for the **autopilot** prototype because there are two event handlers.

21) Comment out the "`exit`" statements in the (now defunct) EVENT_LOOP. There are two places where this needs to be done (refer to Appendix E).

22) Since we are now using the TAE+ code as a part of a larger system, we create what TAE+ calls "time-out" events (see step 30) to ensure that the TAE+ code does not "spin its wheels" when there is no user interaction (no window events). As a result, the "`put_line`" command in the "`when tae_wpt.WPT_TIMEOUT_EVENT =>`" portion of the main "`case`" structure is commented out and replaced with "`null;`".

23) Comment out the "`end loop EVENT_LOOP;`" statement.

24) Add all of the required statements that put information in the TAE+ windows. Consult [TAE93] for details and Appendix E for examples. The **autopilot** example shows the use of `TAE_WPT.WPT_SETREAL` for updating the display of course and altitude, and `TAE_WPT.WPT_SHOWITEM` and `TAE_WPT.WPT_HIDEITEM` to display the status of the aircraft's elevator and rudder.

25) Assign values to output data stream variables as required. In the **autopilot display** example the statements,

```
desired_course:= local_desired_course;
desired_altitude:= local_desired_altitude;
```

are used.

26) Comment out the following statement:

```
tae_wpt.Wpt_Finish;
```

27) Change the name of the procedure in the "`end`" statement.

28) Add a "`begin`" statement. This "`begin`" statement is the beginning of the main execution block for the package body. It will only be executed once, and is therefore, a nice place to put initialization code.

29) Move the TAE+ initialization code from step 18 to the newly created package body main execution block.

30) Add a "time-out" setting so that when there is no user-interaction with the TAE+ windows, the procedure does not wait indefinitely for a window event to occur.

```
tae_wpt.wpt_settimeout(1);
```

31) Put the package body "end". statement at the end of the file. Remember to change the package body's name.

After the required changes have been made to the TAE+ generated code, the file can be used just like any other Ada implementation file. The results are quite excellent and significantly add to the overall aesthetics of the prototype. The interface generated for the **autopilot** prototype is shown in Figure 22.

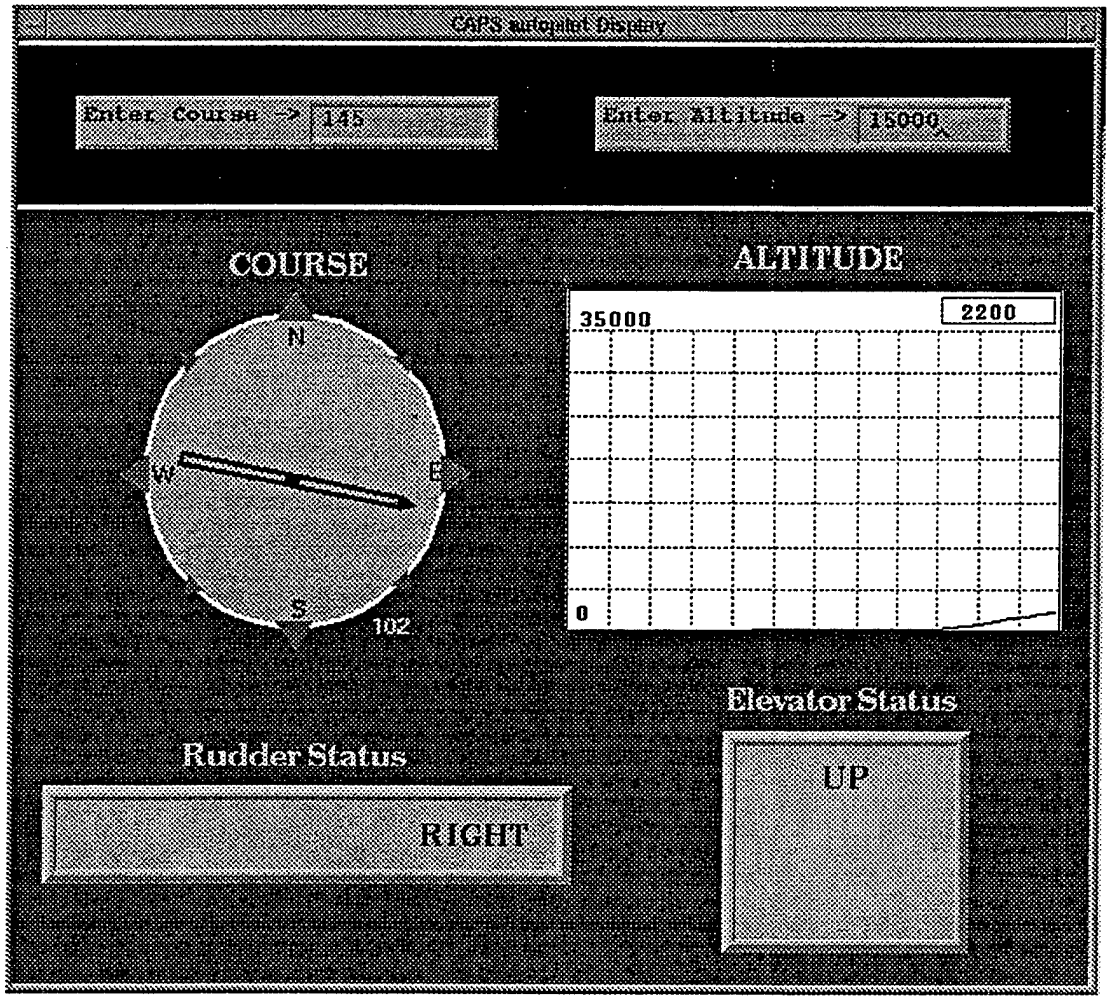


Figure 22. The autopilot Prototype's Graphic User-Interface

The use of such a graphical interface allows real-time input of information as well as real-time display of critical prototype values. There is no scrolling text to contend with as in the text interface method of prototype interaction. TAE+ is quite flexible, and many useful effects can be generated, including rudimentary animation.

2. High Priority Task Approach

An alternative method that has been successfully used for CAPS prototype interface integration is an Ada task approach. The prototype user-interface is implemented as a completely separate task, apart from the static schedule task and the dynamic schedule task. Because of preemption problems at the low level window level, the user-interface task must have a higher priority than not only the static and dynamic schedules, but also the CAPS support routines. The details of creating such an interface are not discussed here.

D. Summary

CAPS provides facilities for generation of high quality prototype user-interfaces through integration of TAE+. Other tools could, no doubt, be used to generate graphical user-interfaces for CAPS prototypes, but none have been explored by the CAPS Development Team at the current time. It is the designer's decision whether or not an elaborate interface is required for any given prototype. Three possible methods for interface integration have been described here. There may be other possibilities.

VIII. Prototype Execution and Diagnosing Errors

A. Overview

When a CAPS prototype is executed, an execution window is created. This window is called the *prototype execution window*, is labeled "<prototype_name>.exe" in the window title bar, and serves as the standard input and standard output for the prototype. Any input or output commands (e.g. get_line, put_line, etc.) in the prototype's operator implementation modules will read and write their information from this window. If a prototype uses a graphic interface as described in the previous chapter, input and output routines are programmed appropriately. CAPS diagnostic output will always appear in the prototype execution window, regardless of the type of prototype user-interface used. Control-C is used to terminate prototype execution.

To terminate prototype execution, enter Control-C in the prototype execution window.

Occasionally, prototypes will generate fatal execution errors, resulting in abnormal termination of execution. When this occurs, the message "EXECUTION ERROR" will appear in the prototype execution window. Diagnostic information will appear in the CAPS alert window after Control-C is entered in the prototype execution window.

When debugging and diagnostic output is programmed into operator implementation modules, this information will appear in the prototype execution window. An example of such output in the **autopilot** prototype is the alert "Maximum altitude exceeded!!!" whenever the altitude goes above 35,000 feet. This output is programmed in the **altimeter** operator.

B. Timing Errors

The CAPS scheduler interprets all timing information in a PSDL program and encodes the consequences of this information into the static and dynamic schedules. The static schedule is an implementation which realizes the schedule and monitors maximum execution time violations at run-time. The only run-time timing errors that are generated during prototype execution are maximum execution time violations. In the static schedule, each operator is allocated an amount of execution time in accordance with its maximum execution time. CAPS will alert the designer whenever an operator fails to execute within its allotted time.

CAPS run-time timing error diagnostic information provides alerts whenever an operator fails to execute within its maximum execution time.

When an operator fails to meet its maximum execution time an excessive number of times

(as determined by system requirements), the prototype's timing constraints should be modified.

CAPS generates prototype schedules for single-processor hardware architectures. Additionally, the schedules generated by CAPS assume that the processor on which the prototype executes is dedicated to the prototype. As a result, if the processor on which the prototype is executed is supporting a large number of other processes, the timing error statistics provided by CAPS will not be completely accurate.

Timing errors generated by CAPS are based on the assumption that the host processor is dedicated to prototype execution.

C. CAPS CPU Speed Ratio

In order to model target hardware that is different than the host machine, CAPS utilizes a CPU speed ratio. The CPU speed ratio is a ratio of target processor speed to host processor speed. Thus, if the CAPS CPU speed ratio is set to 1.5, then CAPS will simulate a processor with a speed 1.5 times greater than that of the host machine by proportionally scaling the actual deadlines. This will result in fewer timing errors (if there are any) due to the simulated higher speed processor. Future versions of CAPS will have more powerful target architecture modeling capabilities.

To modify the CAPS CPU speed ratio, select the "Hardware Model" option from the "Edit" pull-down menu in the main CAPS interface, enter the desired value, and then click the "Apply" button.

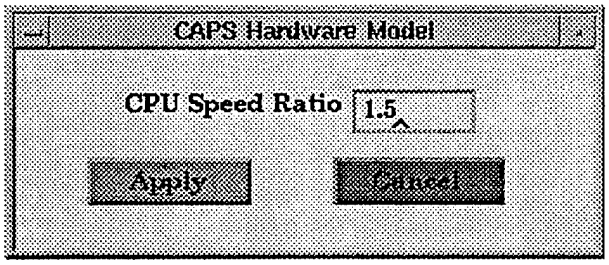


Figure 23. CAPS Hardware Model Modification Window

The active CPU speed ratio is displayed in the CAPS execution window each time a prototype is executed. The default value is 1.0. The CPU speed ratio can be modified between (but not during) prototype executions. Prototypes do not need to be retranslated, rescheduled or recompiled to observe the effects of CPU speed ratio modifications.

D. Buffer Overflow and Underflow

PSDL data streams may contain at most one item of data at any given time. In sampled streams, a single data item can be read from the stream more than once. In data flow steams, however, each item of data must be consumed before a new item is written. If a data flow stream is read by more than one operator, the values on the stream are replicated by an implicit fan-out

operator. In this case, the value on the stream must be consumed by **every** operator that reads from the data stream before the next value is written to the stream. When a producing operator attempts to write to a data flow stream before the data on the stream has been consumed, a buffer overflow occurs. If a data flow stream is read from when no data exists, a buffer underflow occurs. Buffer overflow and underflow are fatal errors, and will terminate prototype execution.

E. Prototype Modification

Modifications to prototypes can be to correct errors or enhance performance. In any case, prototype modification involves invoking the PSDL editor and changing the prototype characteristics. Ada implementation modules, as well as the PSDL program, may need modification. Modifications made to the PSDL program will not take effect until the prototype has been translated, scheduled and compiled. Modification of Ada implementation code only requires that compilation be performed.

IX. Summary

This document has provided a brief overview of basic real-time design concepts and how they can be used in CAPS. When properly used, these concepts enable designers to quickly and iteratively build efficient real-time system prototypes, integrate them with high quality interfaces, and finally execute, evaluate and modify them. Many design issues have been skipped. However, by using the information provided in this tutorial, and building some simple prototypes, designers will quickly develop the skills required to use CAPS to tackle more and more difficult real-time system design challenges.

The PSDL language has been introduced, and its relationship with the CAPS development environment has been described. Specific attention has been given to the basic prototype building tools including the PSDL Editor (which consists of the Graphic Editor, the Syntax Directed Editor and the Graph Viewer), the translator and the scheduler. PSDL is the heart of CAPS and all CAPS prototypes are specified using PSDL. Advanced CAPS features such as the Evolution Control System (ECS), software base retrieval, and prototype merging have been alluded to, but details are not provided in this document. These features are described in detail in the CAPS User's Manual. Some PSDL features have been only briefly mentioned, including *timers* and *exceptions*. Refer to [LBY88] for details.

As a quick summary of concepts described earlier in this document, the following rules of thumb are provided for designers of CAPS real-time prototypes.

1) Break cycles in a prototype graph with state variables. If the prototype data flow diagram (less state streams) is not a directed acyclic graph, scheduling is not possible.

2) Be aware of the equivalent periods being generated for time-critical sporadic operators and use them to maximum benefit.

3) Keep the "LCM of periods" concept of schedule length in mind when assigning timing constraints. The schedule length for a prototype equals the least common multiple of all time-critical operator periods.

4) Aggressively monitor and manage the static schedule load factor. It must be less than 1.0 or scheduling is not possible.

5) Ensure proper periods for producers and consumers of "BY ALL" trigger data streams. Consumers must fire at least as frequently as producers.

6) Understand and follow the CAPS conventions listed in Chapter V, Section B.

Adherence to all of the above rules of thumb will enhance a designer's probability of generating a quality prototype that accurately models the requirements to which it is being built.

X. References

[Ba93] Badr, S., "A Model and Algorithms for a Software Evolution Control System," Ph.D. Dissertation, Naval Postgraduate School, December 1993.

[Do93] Dolgoff, S., "Automated Interface for Retrieving Reusable Software Components," Masters Thesis, Naval Postgraduate School, September 1993.

[Da94] Dampier, D., "A Formal Method for Semantics-Based Change-Merging of Software Prototypes," Ph.D. Dissertation, Naval Postgraduate School, June 1994.

[Lu89a] Luqi, "Handling Timing Constraints in Rapid Prototyping," IEEE Proceedings of the 22nd Annual Hawaii International Conference on System Science, Jan. 1989, 417-424.

[Lu89b] Luqi' "Software Evolution Through Rapid Prototyping," IEEE *Computer*, May 1989, pp.13-25.

[LBY88] Luqi, Berzins, V. and Yeh, R., "A Prototyping Language for Real-Time Software," IEEE *Transactions on Software Engineering*, 14, 10 (October, 1988), 1409-1423.

[LK88] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," IEEE *Transactions on Software Engineering*, October 1988.

[TAE93] TAE+ Reference Manual, Century Computing, Inc., September 1993.

XI. CAPS Glossary

Atomic Operator ..... Lowest level operator in PSDL. In the current version of CAPS, atomic operators are implemented in Ada.

"BY ALL" Trigger..... See Triggered BY ALL.

"BY SOME" Trigger ..... See Triggered BY SOME.

CAPS ..... Computer-Aided Prototyping System. A software development environment used to create real-time software system prototypes.

CAPS Alert Window..... A window that is presented whenever important information needs to be presented to the user. To continue CAPS execution upon seeing a CAPS alert window, click the "OK" button.

CAPS Execution Window..... The window in which CAPS is initially invoked.

Change Request Editor ..... Text editing of change request documents.

Composite Operator..... An operator which is implemented in PSDL. As such, composite operators have associated with them a data flow graph, internal data, and control constraints.

CPU Speed Ratio ..... A ratio of target processor speed to host processor speed.

Data Flow Stream ..... Data flow streams are used in applications where each value in a stream represents a discrete event, and all values written to the stream must be read. A stream is a data flow stream when it appears in a "BY ALL" data trigger.

Data Trigger..... The arrival of data to a consuming operator along a data stream. PSDL supports two types of data triggers: "BY ALL" data triggers and "BY SOME" data triggers.

DDB ..... See Design Database.

Design Database ..... A database used for persistent storage of prototype development data.

Dynamic Schedule..... A schedule (linear list) of non-time-critical operators which are not subject to hard-real time constraints.

ECS ..... See Evolution Control System.

Equivalent Period..... A CAPS-generated period for time-critical sporadic operators.

Evolution Control System..... The CAPS system that provides automated support for coordinating the concurrent efforts of a team of prototype designers and manages multiple versions of the designs they produce.

Execution Guard ..... A condition which is evaluated prior to an operator firing.

Finish Within..... The length of each scheduling interval.

FW ..... See Finish Within.

GE ..... See Graphic Editor.

Graph Viewer..... A non-editable static display of a PSDL operator's graphic decomposition.

Graphic Editor..... The CAPS editor which allows creation of prototype augmented data flow diagrams.

Hardware Model Editor ..... The CAPS tool that allows a prototype designer to model a target hardware architecture that is different than the CAPS host architecture.

Interface Editor ..... The CAPS tool that facilitates development of graphic prototype user-interfaces. CAPS currently integrates TAE+ [TAE93] as its Interface Editor.

Latency..... A lower bound on the amount of time required for transmission of data along the associated data stream.

Load Factor ..... See Static Schedule Load Factor.

Maximum Execution Time ..... An upper bound on the length of time between the instant when a module begins execution and the instant when it completes.

Maximum Response Time ..... An upper bound on the time between the satisfaction of the data trigger conditions and the time when the last value is put into the output streams of the operator.

MCP ..... See Minimum Calling Period.

Merger..... The CAPS system that allows merging of two modifications of a base prototype.

MET ..... See Maximum Execution Time.

Minimum Calling Period ..... A constraint on the environment of a sporadic operator, consisting of a lower bound on the duration between two successive satisfactions of data trigger conditions of an operator.

MRT ..... See Maximum Response Time.

Period ..... A length of time between the start of any scheduling interval and the start of the next scheduling interval of a time-critical operator.

Periodic Operator ..... An operator that is scheduled to execute at approximately regular time intervals.

Prototype Execution Window .... A window created by CAPS for standard prototype I/O during prototype execution.

PSDL ..... Prototype System Description Language. The language upon which CAPS is based.

PSDL Editor ..... This editor consists of 3 separate parts: the Syntax Directed Editor, the Graph Viewer, and the Graphic Editor. The PSDL Editor allows the designer to create the CAPS data flow diagram and PSDL program, and assign all timing and control constraints to prototype components (operators and data streams).

PSDL Program ..... The PSDL text which represents an entire prototype.

Requirements Editor ..... Text editing of requirements documents.

Root Operator ..... The highest level operator in a CAPS prototype.

Sampled Stream ..... Sampled streams are used in applications where a value must be available at all times and values can be replicated without affecting their meaning. A stream is sampled stream when it appears in a "BY SOME" data trigger or is not used as a trigger at all.

Schedule Length ..... The duration (in milliseconds) required for completion of one iteration of a CAPS static schedule.

Scheduling Interval ..... The duration from the earliest time an operator can be fired to the latest time the operator must complete its execution.

Scheduler ..... The CAPS tool that assesses a prototype's scheduling feasibility,

and if feasible, creates a static and dynamic schedule for the prototype.

SDE ..... See Syntax Directed Editor.

Software Base ..... A collection of reusable PSDL and Ada components.

Sporadic Operator ..... An operator that is not explicitly assigned a period. Sporadic operators that are assigned timing constraints other than a period are implemented by CAPS as equivalent periodic operators.

Static Schedule ..... A carefully scripted schedule of a prototype's time-critical operators. The performance of these operators determines whether the system, as designed, meets specified timing requirements.

Static Schedule Load Factor ..... The sum of maximum execution time divided by period for all time-critical operators.

Supervisor File ..... The Ada file containing the packages which comprise a prototype's supervisor module.

Supervisor Module ..... A module consisting of Ada packages. These Ada packages implement a prototype's timing and control constraints via operator drivers, a dynamic schedule and a static schedule. This module is automatically created by CAPS.

Syntax Directed Editor ..... The CAPS editor which allows text editing of PSDL programs.

Timer ..... An abstract state machine whose behavior is similar to a stopwatch.

Time-Critical Operator ..... An operator which has at least one timing constraint associated with it (maximum execution time is minimally required). The operator is non-time-critical otherwise. In CAPS, time-critical sporadic operators are implemented as equivalent periodic operators.

Translator ..... The CAPS tool that converts a PSDL program into Ada source code.

Trigger ..... See Data Trigger.

Triggered BY ALL ..... Condition where an operator can fire whenever new data values have arrived on all of the input streams in the operator's triggering set.

Triggered BY SOME ..... Condition where an operator can fire whenever any one of the inputs in the operator's triggering set gets a new value.

Triggering Set ..... The set of data streams listed after "TRIGGERED BY ALL" or "TRIGGERED BY SOME" in a PSDL program.

This page is intentionally left blank.

XII. Appendix A

This appendix contains the complete source listings for the **TEMP_CONTROLLER** prototype.

PSDL program for
TEMP_CONTROLLER prototype

```
::::::::::::::
file TEMP_CONTROLLER.psdl
::::::::::::::

OPERATOR Cooler
  SPECIFICATION
    INPUT
      Cool_Signal : BOOLEAN
END
IMPLEMENTATION ADA Cooler

END

OPERATOR Evaluate_Temp
  SPECIFICATION
    INPUT
      Temperature : Float
    OUTPUT
      Cool_Signal : BOOLEAN,
      Heat_Signal : BOOLEAN
    MAXIMUM EXECUTION TIME 200 MS
END
IMPLEMENTATION ADA Evaluate_Temp

END

OPERATOR Heater
  SPECIFICATION
    INPUT
      Heat_Signal : BOOLEAN
END
IMPLEMENTATION ADA Heater

END

OPERATOR Sensor
  SPECIFICATION
    OUTPUT
      Temperature : Float
    STATES
      Local_Temperature : Float
      INITIALLY
        40.0
    MAXIMUM EXECUTION TIME 175 MS
```

```
END
IMPLEMENTATION ADA Sensor

END

OPERATOR TEMP_CONTROLLER
  SPECIFICATION
END
IMPLEMENTATION
  GRAPH
    VERTEX Cooler

    VERTEX Evaluate_Temp : 200 MS

    VERTEX Heater

    VERTEX Sensor : 175 MS

    EDGE Cool_Signal
      Evaluate_Temp ->
      Cooler

    EDGE Heat_Signal
      Evaluate_Temp ->
      Heater

    EDGE Temperature
      Sensor ->
      Evaluate_Temp
  DATA STREAM
    Cool_Signal : BOOLEAN,
    Heat_Signal : BOOLEAN,
    Temperature : Float
  CONTROL CONSTRAINTS
    OPERATOR Cooler
      TRIGGERED BY SOME
        cool_signal

    OPERATOR Evaluate_Temp
      TRIGGERED BY ALL
        Temperature
      PERIOD 500 MS

    OPERATOR Heater
      TRIGGERED BY SOME
        heat_signal

    OPERATOR Sensor
      PERIOD 500 MS
END
```

::::::::::::::
file TEMP_CONTROLLER.Cooler.a
::::::::::::::

```
-------------------------------------------------------------------------
-- UNIT          : Cooler.a
-- CSCI          : Temperature Controller
-- CSU           : Cooler
-- Date          : Aug, 2 1994
-- Author        : Osman Ibrahim
-- Compiler      : Sun/Ada
-- Description   : This Package simulates the cooler switch.
-------------------------------------------------------------------------
With Text_Io; Use Text_Io;
Package Cooler_pkg is
 Procedure Cooler(Cool_Signal: In Boolean);
End Cooler_pkg;

Package body Cooler_Pkg is
 Procedure Cooler(Cool_Signal: In Boolean) is
  Begin
    If Cool_Signal = True then
     Put_Line("  Cooler Switch is on");
    else Put_Line("  Cooler Switch is off");
  end if;
  End Cooler;
End Cooler_Pkg;
```

```
::::::::::::::
```
file TEMP_CONTROLLER.Evaluate_Temp.a
```
::::::::::::::
-------------------------------------------------------------------------
-- UNIT          : Evaluate_Temp.a
-- CSCI          : Temperature Controller
-- CSU           : Evaluate_Temp
-- Date          : Aug, 2 1994
-- Author        : Osman Ibrahim
-- Compiler      : Sun/Ada
-- Description   : This Package evaluates the temperature receiving
--                 from Sensor and set Cool_signal, Heat_signal
--                 accordingly.
-------------------------------------------------------------------------

with text_io; use text_io;

Package Evaluate_Temp_Pkg is
 Procedure Evaluate_Temp (Temperature: in Float;
                          Heat_Signal,Cool_Signal: Out Boolean);
End Evaluate_Temp_Pkg;


Package body Evaluate_Temp_Pkg is

package fl_io is new float_io(float);

 Procedure Evaluate_Temp (Temperature: in Float;
                          Heat_Signal,Cool_Signal: Out Boolean) is
  Begin
    If Temperature > 80.0 then
      Begin
        Cool_Signal := True;
        Heat_Signal := False;
      End;
    elsif Temperature < 60.0 then
      Begin
        Cool_Signal := False;
        Heat_Signal := True;
      End;
    else
      Null;
    End if;

  put("Temperature is: ");
  fl_io.put(Temperature,FORE=>3,AFT=>2,EXP=>0);
  new_line;

End Evaluate_Temp;

End Evaluate_Temp_Pkg;


```

::::::::::::::

file TEMP_CONTROLLER.Heater.a

::::::::::::::

```
-------------------------------------------------------------------------
-- UNIT          : Heater.a
-- CSCI          : Temperature Controller
-- CSU           : Heater
-- Date          : Aug, 2 1994
-- Author        : Osman Ibrahim
-- Compiler      : Sun/Ada
-- Description   : This Package simulates the heater switch.
-------------------------------------------------------------------------

With Text_Io; Use Text_Io;
Package Heater_Pkg is
 Procedure Heater(Heat_Signal: In Boolean);
End Heater_pkg;

Package body Heater_Pkg is
 Procedure Heater(Heat_Signal: In Boolean) is
  Begin
    If Heat_Signal then
     Put_Line("  Heater Switch is on");
    else Put_Line("  Heater Switch is off");
  end if;
  End Heater;
End Heater_Pkg;
```

```
::::::::::::::
```
file TEMP_CONTROLLER.sensor.a
```
::::::::::::::
--------------------------------------------------------------------------
-- UNIT: sensor.a
-- CSCI: Temperature Control System
-- CSU : sensor
-- Date: 8/2/94
-- Author: Nguyen, Doan
-- Compiler: Sun/Ada
-- Description: This package contains the operation to simulate
-- a temperator and send it to a temperature evaluator.
--------------------------------------------------------------------------

Package sensor_Pkg is
  Procedure sensor(Temperature: Out Float);
End  sensor_pkg;

Package body sensor_pkg is

  Increasing: Constant Boolean := True;
  Decreasing: Constant Boolean := False;
  Start_time: Constant Integer := 1;
  Limit_time: constant Integer := 100;

  Simulation_time : Integer := Start_time;
  Simulation_cycle : Boolean := Increasing;
  Local_Temperature : Float := 40.0;

  Procedure sensor(Temperature: Out Float) is

  Begin
    If Simulation_cycle = Increasing then
     Begin
      Local_Temperature := Local_Temperature + 1.0;
      Simulation_time := Simulation_time + 1;
      If Simulation_time = Limit_time then
        Begin
          Simulation_time := Start_time;
          Simulation_cycle := Decreasing;
        End;
      End if;
     End;
    End if;
    If Simulation_cycle = Decreasing then
     Begin
      Local_Temperature := Local_Temperature - 1.0;
      Simulation_time := Simulation_time + 1;
      If Simulation_time = Limit_time then
        Begin
          Simulation_time := Start_time;
          Simulation_cycle := Increasing;
        End;
      End if;


```

```
    End;
   End if;
   Temperature := Local_Temperature;
 End sensor;
End sensor_pkg;
```

CAPS-Generated Supervisor Module for
TEMP_CONTROLLER prototype

::::::::::::::
file TEMP_CONTROLLER.a
::::::::::::::

```
  package TEMP_CONTROLLER_EXCEPTIONS is
    -- PSDL exception type declaration
    type PSDL_EXCEPTION is (UNDECLARED_ADA_EXCEPTION);
  end TEMP_CONTROLLER_EXCEPTIONS;

  package TEMP_CONTROLLER_INSTANTIATIONS is
    -- Ada Generic package instantiations

  end TEMP_CONTROLLER_INSTANTIATIONS;

    with PSDL_TIMERS;
  package TEMP_CONTROLLER_TIMERS is
    -- Timer instantiations
  end TEMP_CONTROLLER_TIMERS;

-- with/use clauses for atomic type packages
-- with/use clauses for generated packages.
    with TEMP_CONTROLLER_EXCEPTIONS; use TEMP_CONTROLLER_EXCEPTIONS;
    with TEMP_CONTROLLER_INSTANTIATIONS; use TEMP_CONTROLLER_INSTANTIATIONS;
-- with/use clauses for CAPS library packages.
    with PSDL_STREAMS; use PSDL_STREAMS;
  package TEMP_CONTROLLER_STREAMS is
-- Local stream instantiations

    package DS_COOL_SIGNAL_COOLER is new
      PSDL_STREAMS.SAMPLED_BUFFER(BOOLEAN);

    package DS_HEAT_SIGNAL_HEATER is new
      PSDL_STREAMS.SAMPLED_BUFFER(BOOLEAN);

    package DS_TEMPERATURE_EVALUATE_TEMP is new
      PSDL_STREAMS.FIFO_BUFFER(FLOAT);

-- State stream instantiations

  end TEMP_CONTROLLER_STREAMS;

  package TEMP_CONTROLLER_DRIVERS is
    procedure COOLER_DRIVER;
    procedure EVALUATE_TEMP_DRIVER;
    procedure HEATER_DRIVER;
    procedure SENSOR_DRIVER;
  end TEMP_CONTROLLER_DRIVERS;

-- with/use clauses for atomic components.
```

```
    with COOLER_PKG; use COOLER_PKG;
    with EVALUATE_TEMP_PKG; use EVALUATE_TEMP_PKG;
    with HEATER_PKG; use HEATER_PKG;
    with SENSOR_PKG; use SENSOR_PKG;
-- with/use clauses for generated packages.
    with TEMP_CONTROLLER_EXCEPTIONS; use TEMP_CONTROLLER_EXCEPTIONS;
    with TEMP_CONTROLLER_STREAMS; use TEMP_CONTROLLER_STREAMS;
    with TEMP_CONTROLLER_TIMERS; use TEMP_CONTROLLER_TIMERS;
    with TEMP_CONTROLLER_INSTANTIATIONS; use TEMP_CONTROLLER_INSTANTIATIONS;
-- with/use clauses for CAPS library packages.
    with DS_DEBUG_PKG; use DS_DEBUG_PKG;
    with PSDL_STREAMS; use PSDL_STREAMS;
    with PSDL_TIMERS;
  package body TEMP_CONTROLLER_DRIVERS is

    procedure COOLER_DRIVER is
      LV_COOL_SIGNAL : BOOLEAN;

      EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
      EXCEPTION_ID: PSDL_EXCEPTION;
    begin
-- Data trigger checks.
      if not (DS_COOL_SIGNAL_COOLER.NEW_DATA) then
        return;
      end if;

-- Data stream reads.
      begin
        DS_COOL_SIGNAL_COOLER.BUFFER.READ(LV_COOL_SIGNAL);
      exception
        when BUFFER_UNDERFLOW =>
          DS_DEBUG.BUFFER_UNDERFLOW("COOL_SIGNAL_COOLER", "COOLER");
      end;

-- Execution trigger condition check.
      if True then
        begin
        COOLER(
          COOL_SIGNAL => LV_COOL_SIGNAL);
        exception
          when others =>
            DS_DEBUG.UNDECLARED_EXCEPTION("COOLER");
            EXCEPTION_HAS_OCCURRED := true;
            EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
      end;
      else return;
      end if;

 -- Exception Constraint translations.

 -- Other constraint option translations.

 --Unconditional output translations.
```

```
 -- PSDL Exception handler.
      if EXCEPTION_HAS_OCCURRED then
        DS_DEBUG.UNHANDLED_EXCEPTION(
          "COOLER",
          PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
      end if;
    end COOLER_DRIVER;


    procedure EVALUATE_TEMP_DRIVER is
      LV_TEMPERATURE : FLOAT;
      LV_COOL_SIGNAL : BOOLEAN;
      LV_HEAT_SIGNAL : BOOLEAN;

      EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
      EXCEPTION_ID: PSDL_EXCEPTION;
    begin
-- Data trigger checks.
      if not (DS_TEMPERATURE_EVALUATE_TEMP.NEW_DATA) then
        return;
      end if;

-- Data stream reads.
      begin
        DS_TEMPERATURE_EVALUATE_TEMP.BUFFER.READ(LV_TEMPERATURE);
      exception
        when BUFFER_UNDERFLOW =>
          DS_DEBUG.BUFFER_UNDERFLOW("TEMPERATURE_EVALUATE_TEMP",
"EVALUATE_TEMP");
      end;

-- Execution trigger condition check.
      if True then
        begin
        EVALUATE_TEMP(
          TEMPERATURE => LV_TEMPERATURE,
          COOL_SIGNAL => LV_COOL_SIGNAL,
          HEAT_SIGNAL => LV_HEAT_SIGNAL);
        exception
          when others =>
            DS_DEBUG.UNDECLARED_EXCEPTION("EVALUATE_TEMP");
            EXCEPTION_HAS_OCCURRED := true;
            EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
        end;
      else return;
      end if;

 -- Exception Constraint translations.

 -- Other constraint option translations.

 --Unconditional output translations.
      if not EXCEPTION_HAS_OCCURRED then
        begin
```

```
          DS_COOL_SIGNAL_COOLER.BUFFER.WRITE(LV_COOL_SIGNAL);
        exception
          when BUFFER_OVERFLOW =>
            DS_DEBUG.BUFFER_OVERFLOW("COOL_SIGNAL_COOLER", "EVALUATE_TEMP");
        end;
      end if;
      if not EXCEPTION_HAS_OCCURRED then
        begin
          DS_HEAT_SIGNAL_HEATER.BUFFER.WRITE(LV_HEAT_SIGNAL);
        exception
          when BUFFER_OVERFLOW =>
            DS_DEBUG.BUFFER_OVERFLOW("HEAT_SIGNAL_HEATER", "EVALUATE_TEMP");
        end;
      end if;

 -- PSDL Exception handler.
      if EXCEPTION_HAS_OCCURRED then
        DS_DEBUG.UNHANDLED_EXCEPTION(
          "EVALUATE_TEMP",
          PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
      end if;
    end EVALUATE_TEMP_DRIVER;


    procedure HEATER_DRIVER is
      LV_HEAT_SIGNAL : BOOLEAN;

      EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
      EXCEPTION_ID: PSDL_EXCEPTION;
    begin
-- Data trigger checks.
      if not (DS_HEAT_SIGNAL_HEATER.NEW_DATA) then
        return;
      end if;

-- Data stream reads.
      begin
        DS_HEAT_SIGNAL_HEATER.BUFFER.READ(LV_HEAT_SIGNAL);
      exception
        when BUFFER_UNDERFLOW =>
          DS_DEBUG.BUFFER_UNDERFLOW("HEAT_SIGNAL_HEATER", "HEATER");
      end;

-- Execution trigger condition check.
      if True then
        begin
        HEATER(
          HEAT_SIGNAL => LV_HEAT_SIGNAL);
        exception
          when others =>
            DS_DEBUG.UNDECLARED_EXCEPTION("HEATER");
            EXCEPTION_HAS_OCCURRED := true;
            EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
        end;
```

```
      else return;
      end if;

 -- Exception Constraint translations.

 -- Other constraint option translations.

 --Unconditional output translations.

 -- PSDL Exception handler.
      if EXCEPTION_HAS_OCCURRED then
        DS_DEBUG.UNHANDLED_EXCEPTION(
          "HEATER",
          PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
      end if;
    end HEATER_DRIVER;


    procedure SENSOR_DRIVER is
      LV_TEMPERATURE : FLOAT;

      EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
      EXCEPTION_ID: PSDL_EXCEPTION;
    begin
-- Data trigger checks.

-- Data stream reads.

-- Execution trigger condition check.
      if True then
        begin
        SENSOR(
          TEMPERATURE => LV_TEMPERATURE);
        exception
          when others =>
            DS_DEBUG.UNDECLARED_EXCEPTION("SENSOR");
            EXCEPTION_HAS_OCCURRED := true;
            EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
        end;
      else return;
      end if;

 -- Exception Constraint translations.

 -- Other constraint option translations.

 --Unconditional output translations.
      if not EXCEPTION_HAS_OCCURRED then
        begin
          DS_TEMPERATURE_EVALUATE_TEMP.BUFFER.WRITE(LV_TEMPERATURE);
        exception
          when BUFFER_OVERFLOW =>
            DS_DEBUG.BUFFER_OVERFLOW("TEMPERATURE_EVALUATE_TEMP", "SENSOR");
        end;
```

```
    end if;

-- PSDL Exception handler.
    if EXCEPTION_HAS_OCCURRED then
      DS_DEBUG.UNHANDLED_EXCEPTION(
        "SENSOR",
        PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
    end if;
  end SENSOR_DRIVER;
end TEMP_CONTROLLER_DRIVERS;

package TEMP_CONTROLLER_DYNAMIC_SCHEDULERS is
  procedure START_DYNAMIC_SCHEDULE;
end TEMP_CONTROLLER_DYNAMIC_SCHEDULERS;


with TEMP_CONTROLLER_DRIVERS;  use TEMP_CONTROLLER_DRIVERS;
with PRIORITY_DEFINITIONS;  use PRIORITY_DEFINITIONS;
package body TEMP_CONTROLLER_DYNAMIC_SCHEDULERS is

  task type DYNAMIC_SCHEDULE_TYPE is
    pragma priority (DYNAMIC_SCHEDULE_PRIORITY);
    entry START;
  end DYNAMIC_SCHEDULE_TYPE;
  for DYNAMIC_SCHEDULE_TYPE'STORAGE_SIZE use 100_000;
  DYNAMIC_SCHEDULE : DYNAMIC_SCHEDULE_TYPE;

  task body DYNAMIC_SCHEDULE_TYPE is
  begin
    accept START;
    loop
      Cooler_DRIVER;
      Heater_DRIVER;
    end loop;
  end DYNAMIC_SCHEDULE_TYPE;

  procedure START_DYNAMIC_SCHEDULE is
  begin
    DYNAMIC_SCHEDULE.START;
  end START_DYNAMIC_SCHEDULE;

end TEMP_CONTROLLER_DYNAMIC_SCHEDULERS;


package TEMP_CONTROLLER_STATIC_SCHEDULERS is
  procedure START_STATIC_SCHEDULE;
end TEMP_CONTROLLER_STATIC_SCHEDULERS;

with TEMP_CONTROLLER_DRIVERS;  use TEMP_CONTROLLER_DRIVERS;
with PRIORITY_DEFINITIONS;  use PRIORITY_DEFINITIONS;
with PSDL_TIMERS;  use PSDL_TIMERS;
with TEXT_IO;  use TEXT_IO;
package body TEMP_CONTROLLER_STATIC_SCHEDULERS is
```

```
    task type STATIC_SCHEDULE_TYPE is
      pragma priority (STATIC_SCHEDULE_PRIORITY);
      entry START;
    end STATIC_SCHEDULE_TYPE;
    for STATIC_SCHEDULE_TYPE'STORAGE_SIZE use 200_000;
    STATIC_SCHEDULE : STATIC_SCHEDULE_TYPE;

    task body STATIC_SCHEDULE_TYPE is
      PERIOD : duration;
      Sensor_START_TIME1 : duration;
      Sensor_STOP_TIME1 : duration;
      Evaluate_Temp_START_TIME2 : duration;
      Evaluate_Temp_STOP_TIME2 : duration;
      schedule_timer : TIMER := NEW_TIMER;
    begin
      accept START;
      PERIOD := TARGET_TO_HOST(duration( 5.00000000000000E-01));
      Sensor_START_TIME1 := TARGET_TO_HOST(duration( 0.00000000000000E+00));
      Sensor_STOP_TIME1 := TARGET_TO_HOST(duration( 1.75000000000000E-01));
      Evaluate_Temp_START_TIME2 := TARGET_TO_HOST(duration( 1.75000000000000E-
01));
      Evaluate_Temp_STOP_TIME2 := TARGET_TO_HOST(duration( 3.75000000000000E-
01));
      START(schedule_timer);
      loop
        delay(Sensor_START_TIME1 - HOST_DURATION(schedule_timer));
        Sensor_DRIVER;
        if HOST_DURATION(schedule_timer) > Sensor_STOP_TIME1 then
          PUT_LINE("timing error from operator Sensor");
          SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(HOST_DURATION(schedule_timer) -
Sensor_STOP_TIME1);
        end if;

        delay(Evaluate_Temp_START_TIME2 - HOST_DURATION(schedule_timer));
        Evaluate_Temp_DRIVER;
        if HOST_DURATION(schedule_timer) > Evaluate_Temp_STOP_TIME2 then
          PUT_LINE("timing error from operator Evaluate_Temp");
          SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(HOST_DURATION(schedule_timer) -
Evaluate_Temp_STOP_TIME2);
        end if;

        delay(PERIOD - HOST_DURATION(schedule_timer));
        RESET(schedule_timer);
      end loop;
    end STATIC_SCHEDULE_TYPE;

    procedure START_STATIC_SCHEDULE is
    begin
      STATIC_SCHEDULE.START;
    end START_STATIC_SCHEDULE;

  end TEMP_CONTROLLER_STATIC_SCHEDULERS;
```

```
    with TEMP_CONTROLLER_STATIC_SCHEDULERS; use
TEMP_CONTROLLER_STATIC_SCHEDULERS;
    with TEMP_CONTROLLER_DYNAMIC_SCHEDULERS; use
TEMP_CONTROLLER_DYNAMIC_SCHEDULERS;
    with CAPS_HARDWARE_MODEL; use CAPS_HARDWARE_MODEL;

  procedure TEMP_CONTROLLER is
  begin
    init_hardware_model;
    start_static_schedule;
    start_dynamic_schedule;
  end TEMP_CONTROLLER;
```

CAPS Scheduler Diagnostic Information
for **TEMP_CONTROLLER** prototype

::::::::::::::
file TEMP_CONTROLLER.diag
::::::::::::::

```
PERIODIC OPERATORS TIMING CONTRAINTS:
***** Evaluate_Temp *****
MET =  200
FINISH_WITHIN =  500
PERIOD =  500
***** Sensor *****
MET =  175
FINISH_WITHIN =  500
PERIOD =  500

LOAD FACTOR =    0.7500

SCHEDULE LENGTH = 500

A feasible schedule found, the Earliest Deadline
Scheduling Algorithm Used.
```

OPERATOR	START TIME	STOP TIME	INSTANCE
TEMP_CONTROLLER	0	0	1
Sensor	0	175	1
Evaluate_Temp	175	375	1

```
Elapsed Time For EARLIEST DEADLINE ALGORITHM: 00:00:00.00
```

Sample of an execution session of
TEMP_CONTROLLER prototype

```
Temperature is:  42.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  43.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  44.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  45.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  46.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  47.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  48.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  49.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  50.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  51.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  52.00
  Cooler Switch is off
  Heater Switch is on
Temperature is:  53.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  54.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  55.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  56.00
  Heater Switch is on
  Cooler Switch is off
Temperature is:  57.00
```

XIII. Appendix B

PSDL Program for initial version of **autopilot** prototype.

```
TYPE altitude_command_type
 SPECIFICATION
END
IMPLEMENTATION ADA altitude_command_type

END

TYPE course_command_type
 SPECIFICATION
END
IMPLEMENTATION ADA course_command_type

END

OPERATOR altimeter
 SPECIFICATION
    INPUT
      delta_altitude : INTEGER
    OUTPUT
      actual_altitude : INTEGER
    MAXIMUM EXECUTION TIME 150 MS
END
IMPLEMENTATION ADA altimeter

END

OPERATOR autopilot
 SPECIFICATION
    STATES
      delta_course : INTEGER
      INITIALLY
        0
    STATES
      delta_altitude : INTEGER
      INITIALLY
        0
END
IMPLEMENTATION
 GRAPH
    VERTEX altimeter : 150 MS

    VERTEX compass : 150 MS

    VERTEX control_surfaces : 150 MS

    VERTEX correct_altitude : 150 MS
```

```
    VERTEX correct_course : 150 MS

    VERTEX input_panel

    EDGE actual_altitude
      altimeter ->
      correct_altitude

    EDGE actual_course
      compass ->
      correct_course

    EDGE altitude_command
      correct_altitude ->
      control_surfaces

    EDGE course_command
      correct_course ->
      control_surfaces

    EDGE delta_altitude
      control_surfaces ->
      altimeter

    EDGE delta_course
      control_surfaces ->
      compass

    EDGE desired_altitude
      input_panel ->
      correct_altitude

    EDGE desired_course
      input_panel ->
      correct_course
DATA STREAM
    actual_altitude : INTEGER,
    actual_course : INTEGER,
    altitude_command : altitude_command_type,
    course_command : course_command_type,
    desired_altitude : INTEGER,
    desired_course : INTEGER
CONTROL CONSTRAINTS
    OPERATOR altimeter
      PERIOD 1000 MS

    OPERATOR compass
      PERIOD 1000 MS

    OPERATOR control_surfaces
      PERIOD 1000 MS

    OPERATOR correct_altitude
```

```
        TRIGGERED BY SOME
          actual_altitude
        PERIOD 1000 MS

      OPERATOR correct_course
        TRIGGERED BY SOME
          actual_course
        PERIOD 1000 MS

      OPERATOR input_panel
 END

 OPERATOR compass
  SPECIFICATION
      INPUT
        delta_course : INTEGER
      OUTPUT
        actual_course : INTEGER
      MAXIMUM EXECUTION TIME 150 MS
 END
 IMPLEMENTATION ADA compass

 END

 OPERATOR control_surfaces
  SPECIFICATION
      INPUT
        altitude_command : altitude_command_type,
        course_command : course_command_type
      OUTPUT
        delta_altitude : INTEGER,
        delta_course : INTEGER
      MAXIMUM EXECUTION TIME 150 MS
 END
 IMPLEMENTATION ADA control_surfaces

 END

 OPERATOR correct_altitude
  SPECIFICATION
      INPUT
        actual_altitude : INTEGER,
        desired_altitude : INTEGER
      OUTPUT
        altitude_command : altitude_command_type
      MAXIMUM EXECUTION TIME 150 MS
 END
 IMPLEMENTATION ADA correct_altitude

 END

 OPERATOR correct_course
  SPECIFICATION
      INPUT
```

```
      actual_course : INTEGER,
      desired_course : INTEGER
    OUTPUT
      course_command : course_command_type
    MAXIMUM EXECUTION TIME 150 MS
END
IMPLEMENTATION ADA correct_course

END

OPERATOR input_panel
 SPECIFICATION
    OUTPUT
      desired_altitude : INTEGER,
      desired_course : INTEGER
END
IMPLEMENTATION ADA input_panel

END
```

XIV. Appendix C

Complete scheduler diagnostics for initial version of **autopilot** prototype.

```
PERIODIC OPERATORS TIMING CONTRAINTS:
***** control_surfaces *****
MET =  150
FINISH_WITHIN =  1000
PERIOD =  1000
***** correct_course *****
MET =  150
FINISH_WITHIN =  1000
PERIOD =  1000
***** correct_altitude *****
MET =  150
FINISH_WITHIN =  1000
PERIOD =  1000
***** compass *****
MET =  150
FINISH_WITHIN =  1000
PERIOD =  1000
***** altimeter *****
MET =  150
FINISH_WITHIN =  1000
PERIOD =  1000

LOAD FACTOR =    0.7500

SCHEDULE LENGTH = 1000

A feasible schedule found, the Earliest Deadline
Scheduling Algorithm Used.
```

OPERATOR	START TIME	STOP TIME	INSTANCE
autopilot	0	0	1
altimeter	0	150	1
compass	150	300	1
correct_altitude	300	450	1
correct_course	450	600	1
control_surfaces	600	750	1

```
Elapsed Time For EARLIEST DEADLINE ALGORITHM: 00:00:00.00
```

XV. Appendix D

Ada code implementing the **input_panel** operator in the **autopilot** prototype.
Text interaction is used to input initial values for course and altitude.

File autopilot.display.a:

```
-------------------------------------------------------------------
-- Unit            :  autopilot.input_panel
-- Prototype       :  CAPS autopilot
-- Date            :  June '94
-- Author          :  Jim Brockett
-- Compiler        :  SunAda
-- Description     :  input_panel Ada implementation
-------------------------------------------------------------------
with text_io; use text_io;

package input_panel_PKG is
  procedure input_panel(desired_course   : out Integer;
                        desired_altitude : out Integer);
end input_panel_PKG;

package body input_panel_PKG is

  initial_desired_course   : integer;
  initial_desired_altitude : integer;
  package int_io is new integer_io(integer);

  procedure input_panel(desired_course   : out Integer;
                        desired_altitude : out Integer) is
  begin

    desired_course   := initial_desired_course;
    desired_altitude := initial_desired_altitude;

  end input_panel;

begin -- package body main execution block (executes only ONCE)
  put_line("   CAPS autopilot prototype");
  put_line("(with simple text interaction)");
  put_line("-----------------------------");
  new_line;
  put("Enter desired course => ");
  int_io.get(initial_desired_course);
  if (initial_desired_course < 0 or initial_desired_course > 360)
    then initial_desired_course := 0;
  end if;
  put("Enter desired altitude => ");
  int_io.get(initial_desired_altitude);
  if initial_desired_altitude < 0 then initial_desired_altitude := 0;
  end if;
  new_line;
end input_panel_PKG;
```

XVI. Appendix E

Modified TAE+ code which implements the **display** operator in the **autopilot** prototype.

File autopilot.display.a (from file autopilot.RAW_TAE_INTERFACE.a): **-- STEP 1**

```
-- STEP 2 (begin)
-----------------------------------------------------------------
-- Unit            :  autopilot.display
-- Prototype       :  CAPS autopilot
-- Date            :  June '94
-- Author          :  Jim Brockett
-- Compiler        :  SunAda
-- Description     :  display (and input) Ada implementation
--
-- Notes           :  This code is modified TAE+ output. All
--                    modifications and supportive comments
--                    are followed by "(jrb)".
-----------------------------------------------------------------
-- STEP 2 (end)
with tae; use tae;
with X_Windows;
with text_io;

-- Add "with" and "use" statements
-- for user-defined types used by operator "display". (jrb)
with elevator_status_type_PKG, rudder_status_type_PKG;  -- (jrb) -- STEP 3
use  elevator_status_type_PKG, rudder_status_type_PKG;  -- (jrb) -- STEP 3


-- Remove the TAE+ outer procedure called "autopilot". The scope and name of this
-- procedure will be changed. The scope becomes inside the display_PKG package,
-- and the name becomes "display" (the same as the PSDL operator name). (jrb)

-- procedure autopilot is (jrb) -- STEP 4

-- Supporting procedures for autopilot
-- Including event handling routines.

-- Change the name of the package to conform to CAPS conventions. (jrb)

package display_PKG is  -- package name changed (jrb) -- STEP 5

    package taefloat_io is new text_io.float_io (taefloat);
    procedure initializePanels (file : in string);  -- NOTE: params changed

-- Add the procedure specification to the package specification. Make sure
-- that the parameters correspond to the PSDL program. Also remember that
-- CAPS uses name association for implementation procedure calls.     (jrb)
```

```
procedure display( rudder_status     : in rudder_status_type; -- STEP 6
                   actual_course     : in INTEGER; -- STEP 6
                   desired_course    : out INTEGER; -- STEP 6
                   desired_altitude : out INTEGER; -- STEP 6
                   actual_altitude   : in INTEGER; -- STEP 6
                elevator_status  : in elevator_status_type);-- (jrb) -- STEP 6

-- BEGIN EVENT_HANDLERs

-- A parameter has been added to each event handler as appropriate
-- for the information that is modified within the event handler.    (jrb)

    procedure main_altitude_entry (info : in tae_wpt.event_context_ptr;
                                   altitude_entry : out INTEGER);-- (jrb) -- STEP 7

    procedure main_course_entry (info : in tae_wpt.event_context_ptr;
                                  course_entry : out INTEGER); -- (jrb) -- STEP 7
-- END EVENT_HANDLERs

end display_PKG; -- package name changed (jrb) -- STEP 5 -- STEP 8

------------------------------------------------------------
-- Add a "with" statement for each package "used" by the package body here. (jrb)

with tae;  -- (jrb) -- STEP 9
with text_io;  -- (jrb) -- STEP 9

-- Move the "package body" statement to here. (jrb) -- STEP 11
package body display_PKG is  -- package name changed (jrb) -- STEP 5

    -- use autopilot_support; -- commented out (jrb) -- STEP 10
    use tae.tae_misc;

    theDisplay : X_Windows.Display;
    Application_Done : boolean := false;
    user_ptr : tae_wpt.event_context_ptr;
    main_info : tae_wpt.event_context_ptr;
    etype : wpt_eventtype;
    wptEvent : tae_wpt.wpt_eventptr;

-- Declare local state variables so desired_course and desired_altitude
-- do not get reset to zero when there is no user input. (jrb)
-- These are the implementations of PSDL state variables in the atomic
-- operator "display". (jrb)

local_desired_course : INTEGER;     -- (jrb) -- STEP 12
local_desired_altitude : INTEGER;  -- (jrb) -- STEP 12
------------------------------------------------------------


-- Move the beginning of the package body to
-- encapsulate the above declarations (see above). (jrb) -- STEP 11
-- package body display_PKG is  -- package name changed (jrb) -- STEP 13
```

```
procedure initializePanels (file : in string) is -- STEP 14

            use tae.tae_co;
            use tae.tae_misc;

            tmp_info : tae_wpt.event_context_ptr;

    begin

            -- do one Co_New and Co_ReadFile per resource file
            tmp_info := new tae_wpt.event_context;
            Co_New (0, tmp_info.collection);
            -- could pass P_ABORT if you prefer
            Co_ReadFile (tmp_info.collection, file, P_CONT);

       -- pair of Co_Finds for each panel in this resource file

        main_info := new tae_wpt.event_context;
        main_info.collection := tmp_info.collection;
        Co_Find (main_info.collection, "main_v", main_info.view);
        Co_Find (main_info.collection, "main_t", main_info.target);


       -- Since there can now be MULTIPLE INITIAL PANELS defined from
       -- within the TAE WorkBench, call Wpt_NewPanel for each panel
       -- defined to be an initial panel (but not usually all the panels
       -- which appear in the resource file).


        if main_info.panel_id = NULL_PANEL_ID then
           tae_wpt.Wpt_NewPanel (theDisplay, main_info.target, main_info.view,
                X_Windows.Null_Window, main_info, tae_wpt.WPT_PREFERRED,
                main_info.panel_id);
        else
            tae_wpt.Wpt_SetPanelState (
               main_info.panel_id, tae_wpt.WPT_PREFERRED);
        end if;

    end initializePanels;



--
--          BEGIN EVENT_HANDLERs
--
```

```
-- We must add a parameter to the event handler procedures for the modified
-- piece of information; "desired_altitude" in this case.            (jrb)
procedure main_altitude_entry (info : in tae_wpt.event_context_ptr;  -- (jrb)
                               altitude_entry : out INTEGER) is -- (jrb) -- STEP 15a
    value : array (1..1) of taeint;
    count : taeint;

        begin
        --text_io.put ("Panel main, parm altitude_entry: value = "); -- STEP 15b
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
           --text_io.put_line ("none"); -- STEP 15b
        else
           tae_vm.Vm_Extract_IVAL (info.parm_ptr, 1, value(1));
           --text_io.put_line (taeint'image(value(1))); -- STEP 15b
        end if;

-- Assign the value that was input by the user to the altitude_entry parameter.

        altitude_entry := integer(value(1)); -- STEP 15c
        if integer(value(1)) < 0 then altitude_entry := 0; -- STEP 15c
        end if; -- STEP 15c

    end main_altitude_entry;

-- We must add a parameter to the event handler procedures for the modified
-- piece of information; "desired_course" in this case.              (jrb)
procedure main_course_entry (info : in tae_wpt.event_context_ptr;    -- (jrb)
                             course_entry : out INTEGER) is       -- (jrb) -- STEP 15a
    value : array (1..1) of taeint;
    count : taeint;

        begin
        --text_io.put ("Panel main, parm course_entry: value = "); -- STEP 15b
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
           --text_io.put_line ("none"); -- STEP 15b
        else
           tae_vm.Vm_Extract_IVAL (info.parm_ptr, 1, value(1));
           --text_io.put_line (taeint'image(value(1))); -- STEP 15b
        end if;

-- Assign the value that was input by the user to the course_entry parameter.

        course_entry := integer(value(1)); -- STEP 15c (begin)
            if(integer(value(1)) < 0 or integer(value(1)) > 360)
            then
             course_entry:=0;
             text_io.put_line("ENTRY ERROR: course must be between 0 and 360.");
            endif; -- STEP 15c (end)

    end main_course_entry;

-- END EVENT_HANDLERs
```

```
-- The package body now needs to include the following procedure, as
-- it becomes the procedure that implements the "display" operator and
-- that we call from the CAPS supervisor module. Therefore, the package "end"
-- statement has been commented out here, and put at the end of this file. (jrb)

-- end display_PKG; -- package name changed (jrb) -- STEP 16 -- STEP 5

------------------------------------------------------------------------

--
-- Main Program (This is now the main procedure for implementation
--               of the "display" operator rather than an infinite
--               outer TAE event_loop)                                    (jrb)
--

-- add the "procedure" statement (jrb)
procedure display( rudder_status     : in rudder_status_type; -- STEP 17
                   actual_course     : in INTEGER; -- STEP 17
                   desired_course    : out INTEGER; -- STEP 17
                   desired_altitude  : out INTEGER; -- STEP 17
                   actual_altitude   : in INTEGER; -- STEP 17
              elevator_status  : in elevator_status_type) is -- (jrb) -- STEP 17

begin

-- Now, we must move this initialization stuff out of the "display" procedure.
-- It is all commented out here, and moved
-- to the end of the package body, where it
-- will be executed only once. (jrb)

-- (begin jrb)
--    f_force_lower (FALSE);    -- permit upper/lowercase file names -- STEP 18
--    tae_wpt.Wpt_Init ("",theDisplay); -- STEP 18

-- PROGRAMMER NOTE:
-- To enable scripting, uncomment the following line.  See the
-- taerecord man page.
--    tae_wpt.Wpt_ScriptInit ("autopilot"); -- STEP 18
--    tae_wpt.Wpt_NewEvent (wptEvent); -- STEP 18

--    initializePanels ("autopilot.res");   -- single call -- STEP 18

--      main event loop -- STEP 18

-- Comment out the outer loop. (jrb)
--EVENT_LOOP: -- STEP 18
--    while not Application_Done loop -- STEP 18
-- (end jrb)


tae_wpt.Wpt_NextEvent (wptEvent, etype);     -- get next event
```

```
-- NOTE: This case statement includes STUBs for non-WPT_PARM_EVENT events.

        case etype is

            when wpt_eventtype'first .. -1 => null;
                -- iterate loop on Wpt_NextEvent error

-- TYPICAL CASE: Panel Event (WPT_PARM_EVENT)

            when tae_wpt.WPT_PARM_EVENT =>
            -- You can comment out the following "put" call.
            -- The appropriate EVENT_HANDLER finishes the message.
                --text_io.put ( "Event: WPT_PARM_EVENT, " ); -- (jrb) -- STEP 19

            --      Panel event has occurred.
            --      Get parm name and then call appropriate EVENT_HANDLER.
            --
            --                              CAUTION:
            --      DO NOT call Wpt_Extract_Parm_xEvent from any other branch
            --      of this "case" statement or you'll get "storage_error".
            --
                    tae_wpt.Wpt_Extract_Context (wptEvent, user_ptr);
                    tae_wpt.Wpt_Extract_Parm (wptEvent, user_ptr.parm_name);
                    tae_wpt.Wpt_Extract_Data (wptEvent, user_ptr.datavm_ptr);
                    tae_vm.Vm_Find (user_ptr.datavm_ptr, user_ptr.parm_name,
                                    user_ptr.parm_ptr);

                -- WPT_PARM_EVENT, BEGIN panel main

                if tae_wpt."=" (user_ptr, main_info)  then
                    -- determine appropriate EVENT_HANDLER for this item
                    if s_equal ("altitude_entry", user_ptr.parm_name) then
                        main_altitude_entry (user_ptr,local_desired_altitude);
-- Added parameter to -- STEP 20 (previous line)
-- event handler call. (jrb)
                    elsif s_equal ("course_entry", user_ptr.parm_name) then
                        main_course_entry (user_ptr,local_desired_course);
-- Added parameter to -- STEP 20 (previous line)
-- event handler call. (jrb)
                    end if;     -- END panel main

                else
                    text_io.put_line ("unexpected event from wpt!");
-- Comment out the "exit" statement. (jrb) -- STEP 21 (next line)
                -- exit; -- or raise an exception, but compiler warns if no exit
                end if;

            when tae_wpt.WPT_FILE_EVENT =>
                        text_io.put_line ("STUB: Event WPT_FILE_EVENT");

                        -- Use Wpt_AddEvent and Wpt_RemoveEvent and
                        -- Wpt_Extract_EventSource and Wpt_Extract_EventMask
```

```
            when tae_wpt.WPT_TIMEOUT_EVENT =>
                  -- Comment this "put_line" statement out and do nothing. (jrb)
                          null;  -- (jrb) -- STEP 22 (this line and next line)
                   -- text_io.put_line ("STUB: Event WPT_TIMEOUT_EVENT"); (jrb)

                        -- Use Wpt_SetTimeOut for this

            when tae_wpt.WPT_TIMER_EVENT =>
                        text_io.put_line ("STUB: Event WPT_TIMER_EVENT");

                        -- Use Wpt_AddTimer and Wpt_RemoveTimer and
                        -- Wpt_Extract_TimerId, Wpt_ExtractTimerRepeat,
                        -- and Wpt_Extract_TimerInterval

-- LEAST LIKELY cases follow:

            when tae_wpt.WPT_WINDOW_EVENT => null ;

                     -- WPT_WINDOW_EVENT can be caused by user acknowledgement
                        -- of a Wpt_PanelMessage or windows which you
                        -- directly create with X (not TAE panels).
                        -- You MIGHT want to use Wpt_Extract_xEvent_Type here.
                        --
                       -- DO NOT use Wpt_Extract_Parm_xEvent since this is not
                        -- a WPT_PARM_EVENT; you'll get a "storage error".

            when tae_wpt.WPT_HELP_EVENT =>          -- OR null ;
                        text_io.put("ERROR: WPT_HELP_EVENT: ");
                   text_io.put_line("should never see; reserved for TAE use");

            when tae_wpt.WPT_INTERRUPT_EVENT =>     -- OR null ;
                        text_io.put("ERROR: WPT_INTERRUPT_EVENT: ");
                   text_io.put_line("should never see; reserved for TAE use");

            when OTHERS =>
                        text_io.put
                        ("FATAL ERROR: Unknown Wpt_NextEvent Event Type: ");
                        text_io.put (wpt_eventtype'image(etype) ) ;
                        text_io.put_line (" ... Forcing exit.");
-- Comment out the "exit" statement. (jrb)
                        -- exit;  -- or raise an exception  -- STEP 21

        end case;
-- NOTE: Do not add statements between here and "end loop EVENT_LOOP"

-- Comment out the "end loop" statement. (jrb)
--    end loop EVENT_LOOP; (jrb) -- STEP 23
```

```
-- It is here that we write the parameters
-- that the "display" procedure receives to the
-- actual windows of the user-interface.          (jrb)

-- All of this code has been added, to "-- (end jrb)". (jrb) -- STEP 24 (begin)

TAE_WPT.WPT_SETREAL(main_info.panel_id, "course",TAEFLOAT(actual_course));
TAE_WPT.WPT_SETREAL(main_info.panel_id,"altitude",TAEFLOAT(actual_altitude));

case rudder_status is

      when left => TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_right");
                   TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_straight");
                    TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "rudder_left");
      when right => TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "rudder_right");
                   TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_straight");
                     TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_left");
     when straight => TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_right");
                   TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "rudder_straight");
                      TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "rudder_left");
    end case;

    case elevator_status is

      when up => TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_down");
                 TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_level");
                 TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "elevator_up");
      when down => TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "elevator_down");
                   TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_level");
                   TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_up");
      when level => TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_down");
                   TAE_WPT.WPT_SHOWITEM(MAIN_INFO.PANEL_ID, "elevator_level");
                    TAE_WPT.WPT_HIDEITEM(MAIN_INFO.PANEL_ID, "elevator_up");
    end case;

-- (end jrb) -- STEP 24 (end)

-- Transfer local variable values to actual prototype data stream parameters.

desired_course := local_desired_course; -- STEP 25
desired_altitude := local_desired_altitude; -- STEP 25


-- Comment out the next line. (jrb)
--    tae_wpt.Wpt_Finish;    -- close all display connections -- STEP 26

-- Change the name of the procedure. (jrb)
end display; -- (jrb) -- STEP 27
```

```
-- Add a "begin" statement. (jrb)

begin -- STEP 28

-- some fluff (jrb)
            text_io.put_line("CAPS autopilot Example Prototype"); -- (jrb)
            text_io.put_line("--------------------------------"); -- (jrb)

-- Here is the moved initialization code. (jrb)

-- STEP 29 (begin)

                f_force_lower (FALSE); -- permit upper/lowercase file names
                tae_wpt.Wpt_Init ("",theDisplay);

            -- PROGRAMMER NOTE:
            -- To enable scripting, uncomment the following line.  See the
            -- taerecord man page.
            -- tae_wpt.Wpt_ScriptInit ("autopilot");

                tae_wpt.Wpt_NewEvent (wptEvent);
                initializePanels ("autopilot.res");   -- single call

-- STEP 29 (end)

   -- Add a time-out limit so that the TAE+ procedure doesn't "spin its wheels"
   -- when there is no user-input to the panels. (jrb)

            tae_wpt.wpt_settimeout(1);  -- (jrb) -- STEP 30

    -- Also note that the output in the time-out
    -- portion of the "case" statement above has been commented out. (jrb)

end display_PKG; -- Package "end" statement moved to here and
            -- the name has been changed. (jrb) -- STEP 16 -- STEP 5 -- STEP 31
```

APENDIX F: CAPS USERS MANUAL

CAPS Users Manual

Table of Contents

I. Introduction

- A. Overview of CAPS
- B. Organization of Chapters
- C. Invoking CAPS
- D. CAPS User-Interfaces
- E. The CAPS Execution Window
- F. The CAPS Alert Windows

II.The Prototype Pull-Down Menu

- A. New
- B. Choose
- C. Commit Work
- D. Save to DDB
- E. Retrieve from DDB
- F. Quit

III. The Edit Pull-Down Menu

- A. PSDL
  1. The Syntax Directed Editor
     a. The CAPS-Cmds pull-down menu
  2. The Graph Viewer
  3. The Graphic Editor
- B. Ada
- C. Interface
- D. Requirements
- E. Change Request
- F. CAPS Defaults
  1. Ada Editor
  2. Text Editor
  3. Schedule Diagnostic Display
  4. Changing the Design Database

IV. The Databases Pull-Down Menu

- A. Design Database
  1. show components
  2. show steps
  3. show step details
  4. show schedule
  5. show prototypes

6. quit
B. Software Base
C. View Text File

V. The Exec Support Pull-Down Menu

A. Translate
B. Schedule
C. Compile
D. Execute

VI. The Project Control Pull-Down Menu

A. Step Operations
1. edit step
2. create step
3. approve step
4. schedule step
5. commit step
6. suspend step
7. abandon step
8. show steps
9. show step details
10. quit
B. Edit Team
C. Merge Prototypes

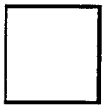
VII. The Help Selection

VIII. References

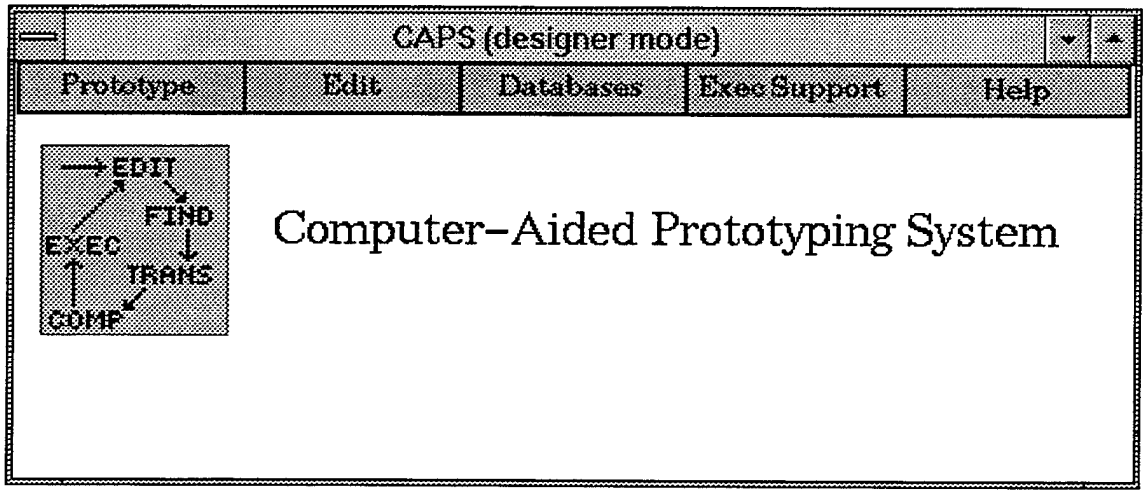
IX. Miscellaneous

I. Introduction

A. Overview of CAPS



The Computer-Aided Prototyping System (CAPS) [LK88] is a software engineering tool for developing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL) [LBY88], which provides facilities for modeling timing and control constraints within a software system. CAPS is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface.



The Computer-Aided Prototyping System (CAPS) [LK88] is a software engineering tool for developing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL) [LBY88], which provides facilities for modeling timing and control constraints within a software system. CAPS is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface.

A CAPS prototype is initially built as an augmented data flow diagram and a corresponding PSDL program. The CAPS data flow diagram and PSDL program are augmented with timing and control constraint information. This timing and control constraint information is used to model the functional and real-time aspects of the prototype. The CAPS environment provides all of the necessary tools for engineers to quickly develop, analyze and refine real-time software systems.

CAPS is a collection of tools, integrated by a user-interface. The user-interface provides access to all of the CAPS tools and facilitates communication between tools when necessary. The tools are grouped into four sections, Editors, Execution Support, Project Control and Software Base. Access is provided to each group of tools via pull-down menus in the CAPS user-interfaces.

B. Organization of Chapters

This document is organized in parallel with the CAPS user-interface pull-down menus. A chapter is provided for each main pull-down menu, with subsections provided for each sub-function of the main pull-down menu.

Throughout this document there are emphasis boxes. These boxes contain information that is very important. The information addresses such issues as current CAPS implementation restrictions and CAPS "buttonology" idiosyncracies. In any event, they appear wherever it is deemed that vital information is presented.

If the the meaning of the text in these emphasis boxes is not understood, the result can be anything from frustration to catastrophe!

C. Invoking CAPS

It does not matter from where in your directory structure you invoke CAPS. CAPS can be invoked in either a designer or manager mode.

1. The Designer Mode

Invoke CAPS in the designer mode by entering

caps

This command will bring up the main interface.

2. The Manager Mode

CAPS can be used in either the manager mode or the designer mode. The designer mode is the default. To run CAPS in the manager mode, use the -m flag.

caps -m

This command will bring up the main interface. Depending on your situation, use of the manager mode may be restricted. Consult your CAPS administrator.

3. Choosing a Design Database

The name of the design database used during a CAPS session defaults to the value of the environment variable CAPS_DDB, or to the user's login name if CAPS_DDB is undefined.

To run CAPS with a design database other than the default, use the -d flag. Note that the design database being used can be changed during a CAPS session.

caps -d <other_ddb_name>

If the -d flag is used, the name of the design database to be used MUST follow the -d flag.

The -m and -d flags can be used together, and the order is unimportant.

caps -m -d <other_ddb_name>

is the same as

caps -d <other_ddb_name> -m

When CAPS is invoked, the mode of operation and the name of the design database being used are presented to the user in the CAPS execution window. CAPS executes as a background process, allowing other use of the CAPS execution window during a CAPS session.

It is important that a single user NOT have more than one CAPS process running simultaneously. This will result in data corruptions. This applies even if one process is in the designer mode and the other is in the manager mode.

D. CAPS User-Interfaces

The CAPS user-interfaces for the designer mode and the manager mode are very similar. The primary difference is that the manager interface has an additional pull-down menu calls "Project Control". The "Project Control" pull-down menu is used for design team management, project evolution, and prototype merging. The functions provided in the "Project Control" pull-down menu are not necessary for single-user prototype development.

The main CAPS user-interfaces invoke the CAPS tools and provide intermediary support functions. These functions include presentation of selection lists, acknowledgement requests and text input requests. In most cases, when selecting from lists in CAPS interface windows, double-clicking the mouse button DOES NOT work as one would expect (or desire) for selection. For this reason, there are "OK" and "Apply" buttons in many of the CAPS-generated windows where one would expect to be able to double-click the mouse to make a selection.

E. The CAPS Alert Window

In many instances, CAPS provides messages to the user through the CAPS alert window. This window is used exclusively to provide information to the user. Such information includes alerts, warnings and errors. To acknowledge the alert and terminate the the window simply click the "OK" button.

When the CAPS alert window appears, functions in the main user-interface are still active. Users are advised, however, to acknowledge the CAPS alert and thus, terminate the window, by clicking the "OK" button before continuing with CAPS execution.

CAPS also makes occasional use of the X-Window system "alert" function. An example of such a window is:

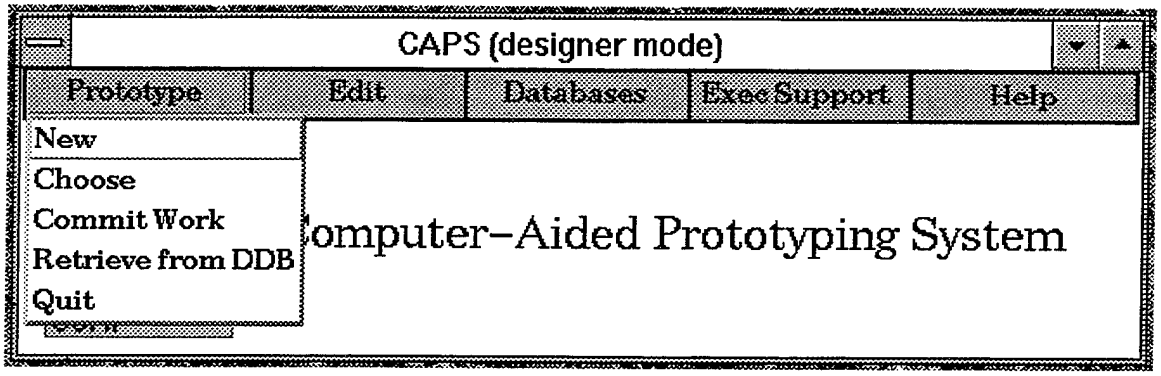
These windows can be identified by the word "alert" in the title bar. The physical appearance of these system alert windows is X-Windows controlled. When presented with such a window, the user MUST acknowledge it before continuing with normal CAPS operation, otherwise X errors (i.e. BadWindow) may occur and CAPS will terminate.

F. The CAPS Execution Window

CAPS runs in an X-Windows environment. The window in which CAPS is invoked is called the CAPS execution window. While most of the CAPS feedback and user interaction is in the more elegant form of exclusively created windows, there is some information that is presented as simple text in the CAPS execution window. Note that the CAPS execution window is NOT the same as the prototype execution window. When prototypes are executed, an exclusive prototype execution window is created for that execution.

Much of the information presented in the CAPS execution window is merely informative in nature, requiring no user response. In some cases, however, important information, on which action should be taken is presented. All such cases are described in the appropriate sections of this document.

II. The "Prototype" Pull-Down Menu



The "Prototype" pull-down menu is used primarily to select which prototype will be active during a CAPS session. CAPS allows the user to switch between active prototypes during any session. The concept of an "open" prototype is used. For many CAPS functions, there must be an open prototype. When CAPS functions are activated which expect there to be an open prototype, and there is no such open prototype the alert:

appears. Click the "OK" button to acknowledge the alert and then open a prototype using the "Open" command in order to perform the desired function.

A. "New"

This selection is used to create a new prototype.

When a new prototype is created, CAPS automatically invokes the PSDL Editor. The Syntax Directed Editor will appear with a single component (an operator), with an identical name as that of the newly created prototype. The Graph Viewer is also displayed with an empty graph when a new prototype is created. See Chapter III for details regarding the PSDL Editor.

B. "Open"

The "Open" command allows a CAPS user to open any prototype which resides in his or her private workspace. A selection list of all available prototypes is presented. After selecting the desired prototype, click the "OK" button. Unlike prototype creation, opening a prototype does not automatically invoke the PSDL editor. Use the "PSDL" selection under the "Edit" pull-down menu. to invoke the PSDL editor for existing prototypes. Only one prototype can be open at any one time. If there is currently an open prototype, and "Open" is activated, the most recent "Open" selection is active, and the previously open prototype is automatically closed.

Prototype selection is accomplished using the "OK" button. This is an instance where the selection CANNOT be double-clicked for activation.

C. "Commit Work" (Designer Mode Only)

When CAPS is used in a team development environment, designers will be assigned substeps. A substep is a unit of work which will modify some portion of a prototype. Top level steps are decomposed into

one or more substeps, and top level steps are not explicitly assigned to designers. Upon completion of a system-assigned substep, the "Commit Work" command is used. This will transfer the completed work to the project manager and activate the Evolution Control System.

The designer need only enter the number of the substep on which work is complete.

D. "Save to DDB" (Manager Mode Only)

Save an INITIAL version of a prototype in the design database.

E. "Retrieve from DDB" (Manager Mode Only)

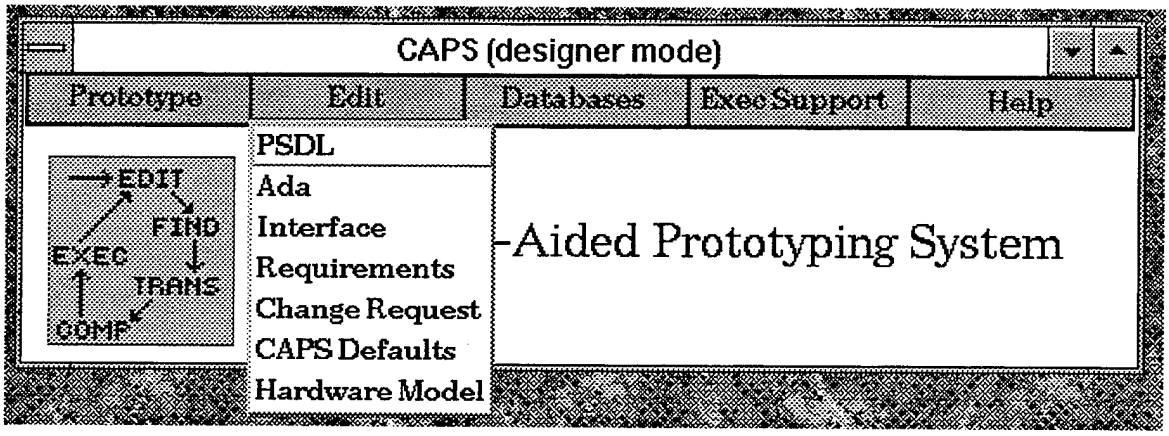
F. "Close"

Close the currently open prototype. This makes the active, open prototype null. Some CAPS functions are not available if there is not an open prototype. In such cases, the following alert appears.

G. "Quit"

Quit CAPS. Open prototypes are closed, however if any other tools are active, they may remain so. For example, if the PSDL Editor is running, and "Quit" is selected, the PSDL Editor process will not be automatically terminated. Exiting other tools after the main user-interface has been terminated with the "Quit" command should be done gracefully (e.g. use the "quit" or "exit" command provided by the still-running tool).

III. The "Edit" Pull-Down Menu



An executable CAPS prototype consists of six basic elements:

1) a PSDL program,

2) a supervisor module,

3) (a set of) implementation modules (written in Ada),

4) (a set of) user-interface graphic files (optional),

5) (a set of) requirements documents (optional),

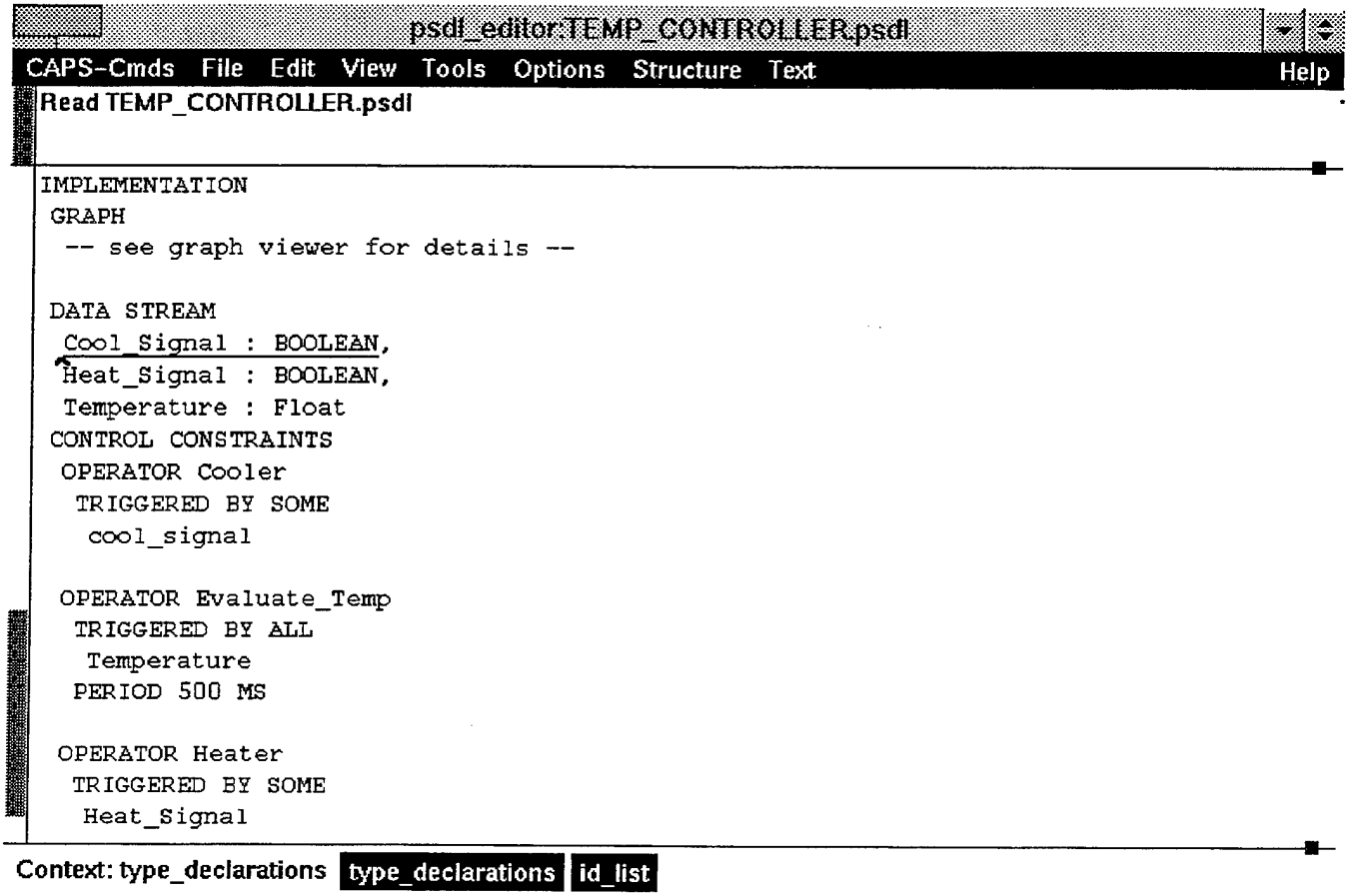
6) (a set of) change request documents (optional).

CAPS incorporates tools for editing each basic element of a prototype. This chapter describes the tools which perform these functions.

A. "PSDL"

The CAPS PSDL Editor consists of three parts. These are the Syntax Directed Editor, the Graph Viewer, and the Graphic Editor.

1. The Syntax Directed Editor

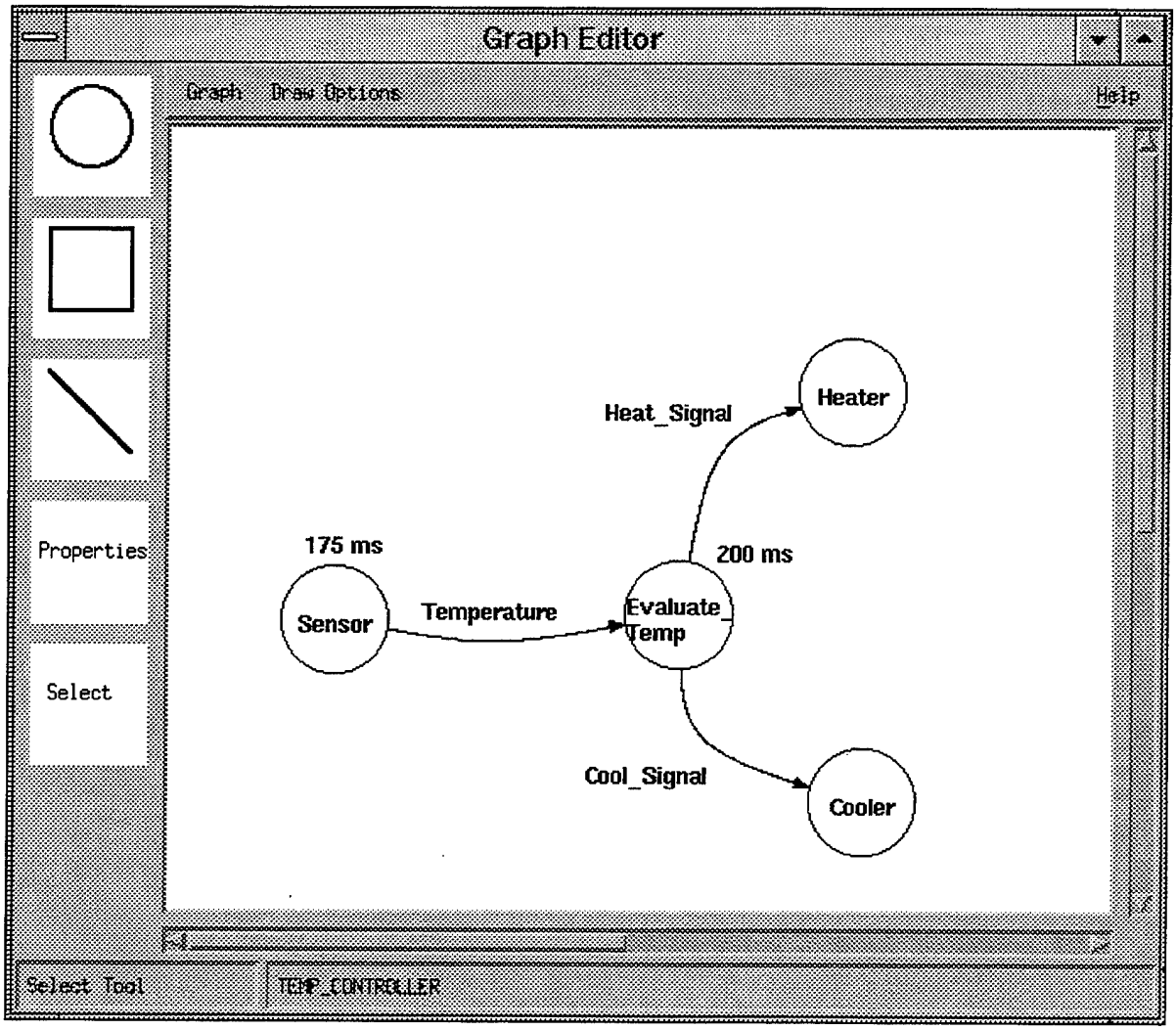


a. PSDL program modification

b. The "CAPS-Cmds" pull-down menu

2. The Graph Viewer

3. The Graphic Editor



a. The Graphic Editor Palette

CIRCLE - Draw circular operators.

SQUARE - Draw rectangular operators (terminators).

LINE - Draw data streams.

"Properties" - Assign properties to the selected operator or data stream.

"Select" - Enable selection of an object in the graph.

b. The "Graph" pull-down menu

c. The "Draw Options" pull-down menu

B. "Ada"

This selection is used to invoke the selected Ada editor. This will invoke either the Verdix Ada Syntax Directed Editor or a text editor (vi or emacs). The desired editor can be selected by the user via the "CAPS Defaults" selection in the "Edit" pull-down menu. Details regarding the operation of any of these editors is NOT provided in this document.

If the Verdix Ada Syntax Directed Editor is being used, file selection is accomplished through the VadsEdit interface.

C. "Interface"

CAPS integrates a call to the commercially available product TAE+ [TAE93] for generation of graphical prototype user-interfaces.

D. "Requirements"

E. "Change Request"

F. "CAPS Defaults"

The CAPS Defaults editing window allows the user to modify settings during a single CAPS session. Upon exiting CAPS, all settings return to their original values. Upon invocation of CAPS the defaults are as follows.

Ada Editor

Verdix Ada Syntax Directed Editor

Text Editor

vi

Scheduler Diagnostics

ON

Design Database

Set to the value of the $USER environment variable OR the value of the $CAPS_DDB environment variable (if assigned).

The "apply" button must be used to activate the selected options. If no option is selected for any of the four items, the value of the non-selected item remains unchanged upon initiating "apply". The "cancel"

button can be used to exit the CAPS Defaults window without making any changes to the default values.

1. Ada Editor

Options for Ada editing are vi, emacs and the Verdix Ada Syntax Directed Editor. The initial setting for all CAPS sessions is the Verdix Ada Syntax Directed Editor. This document does not describe the details of using the Verdix Ada Syntax Directed Editor.

When the "text editor" option is selected, the active text editor will be used for Ada editing. When "text editor" is selected for Ada editing, CAPS provides a selection menu from which the user can chose the desired Ada file. If the file is a new file and does not appear in the selection list, the user is able to enter the name of the new file in the "file to edit" portion of the file selection window.

2. Text Editor

Options for text editing are vi and emacs. The default is vi. The text editor is used for editing Ada files, requirements files and change request files.

3. Schedule Diagnostic Display

The CAPS scheduler generates diagnostic information and displays this information every time a prototype is scheduled. The display of this information can be turned on or off, as desired, in the CAPS Defaults window.

4. Changing the Design Database

To change the design database being used, enter the name of the desired database in the "Design Database Name" portion of the CAPS Defaults window. Note that if the name entered is not a registered ONTOS database, the CAPS Defaults window will NOT alert the user. Rather, a text alert will be presented in the CAPS execution window at the next attempted design database access.

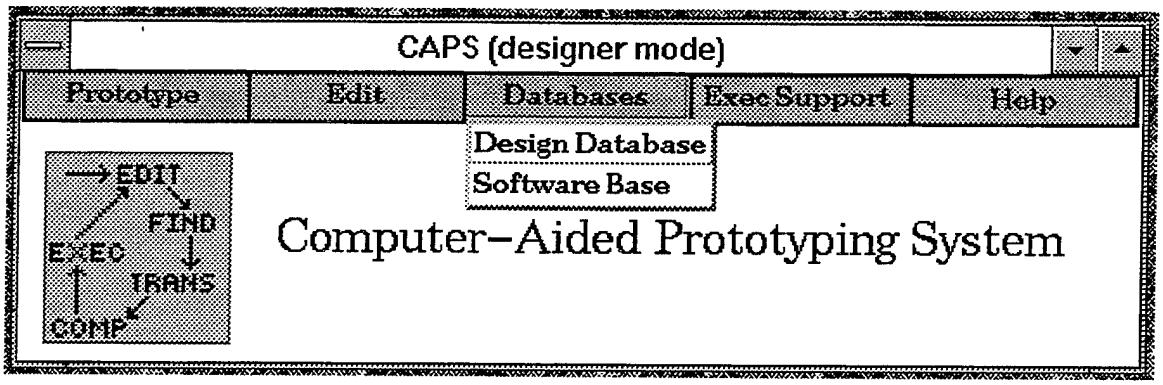
ONTOS databases CANNOT be created or registered from within CAPS. See the CAPS Administrator's Manual or the ONTOS Reference Manual for details regarding database creation and registration.

G. "Hardware Model"

In order to model target hardware that is different than the host machine, CAPS provides a CPU speed ratio. The CPU ratio is a ratio of target processor speed to host processor speed. Thus, if the CAPS CPU speed ratio is set to 1.5, then CAPS will simulate a processor with a speed 1.5 times greater than that of the host machine. This will result in fewer timing errors (if there are any) due to the simulated higher speed processor. Future versions of CAPS will have more powerful target architecture modeling capabilities.

The current CPU speed ratio is displayed in the CAPS execution window each time a prototype is executed. The CPU speed ratio can be modified between prototype executions, and prototypes do not need to be retranslated, rescheduled or recompiled to observe the effects of CPU speed ratio modifications.

IV. The "Databases" Pull-Down Menu



A. "Design Database"

1. "show components"

2. "show steps"

3. "show step details"

4. "show schedule"

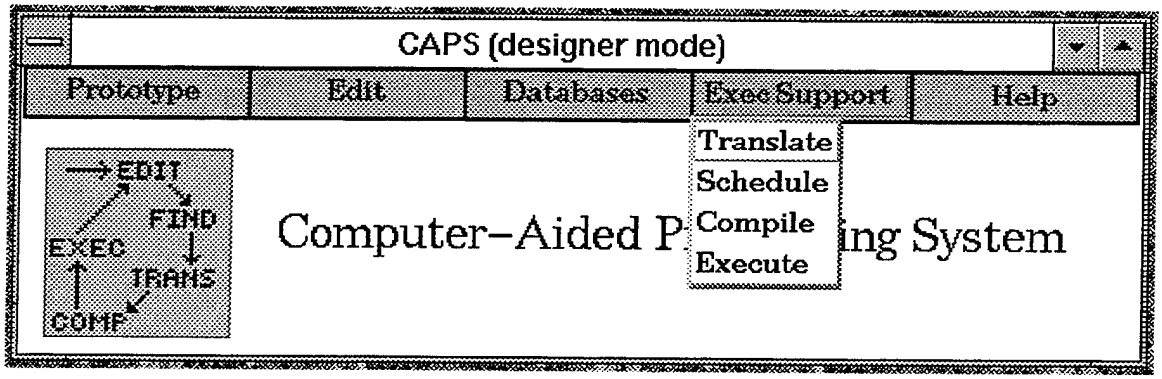
5. "show prototypes"

6. "quit"

B. "Software Base"

C. "View Text File"

V. The "Exec Support" Pull-Down Menu



A. "Translate"

This commmand invokes the CAPS translator.

B. "Schedule"

This command invokes the CAPS scheduler.

C. "Compile"

D. "Execute"

VI. The "Project Control" Pull-Down Menu

(Manager Mode Only)

A. "Step Operations"

1. "edit step"

2. "create step"

3. "approve step"

4. "schedule step"

5. "commit step"

6. "suspend step"

7. "abandon step"

8. "show steps"

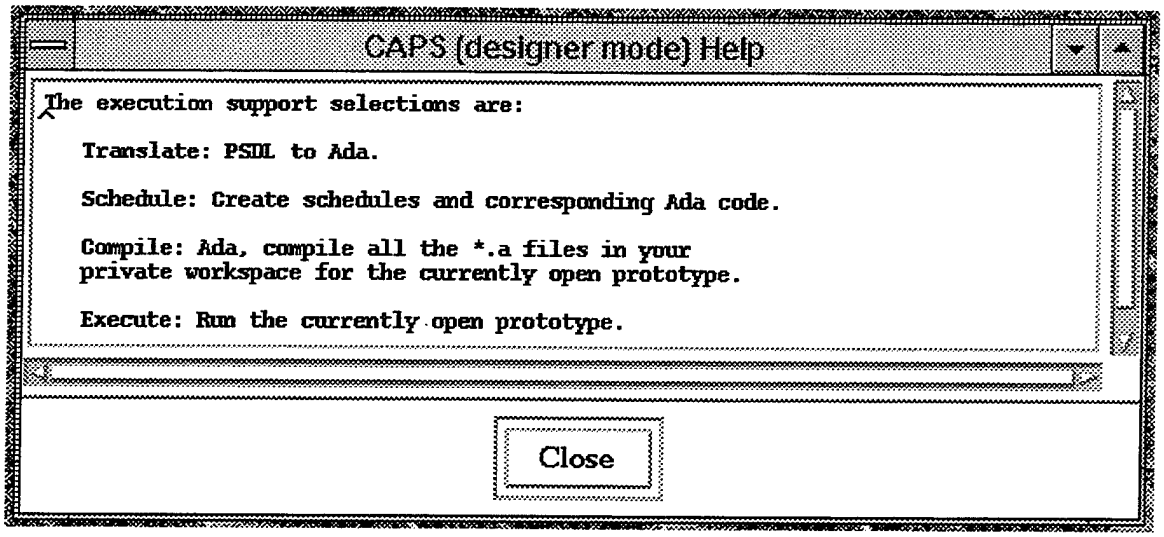
9. "show step details"

10. "quit"

B. "Edit Team"

C. "Merge Prototypes"

VII. The "Help" Selection



To Use Help, Click on the HELP Menu Button. After the question mark cursor comes up,

move the cursor and click on the menu item that you have a question about. A context sensitive help window will appear.

VIII. References

[Ba93] Badr, S., "A Model and Algorithms for a Software Evolution Control System," Ph.D. Dissertation, Naval Postgraduate School, December 1993.

[Do93] Dolgoff, S., "Automated Interface for Retrieving Reusable Software Components," Masters Thesis, Naval Postgraduate School, September 1993.

[Da94] Dampier, D., "A Formal Method for Semantics-Based Change-Merging of Software Prototypes," Ph.D. Dissertation, Naval Postgraduate School, June 1994.

[Lu89a] Luqi, "Handling Timing Constraints in Rapid Prototyping," IEEE Proceedings of the 22nd Annual Hawaii International Conference on System Science, Jan. 1989, 417-424.

[Lu89b] Luqi' "Software Evolution Through Rapid Prototyping," IEEE Computer, May 1989, pp.13-25.

[LBY88] Luqi, Berzins, V. and Yeh, R., "A Prototyping Language for Real-Time Software," IEEE Transactions on Software Engineering, 14, 10 (October, 1988), 1409-1423.

[LK88] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," IEEE Transactions on Software Engineering, October 1988.

[ONT92] ONTOS DB 2.2 Reference Manual, ONTOS Inc., Burlington, MA, Feb. 1992.

[TAE93] TAE+ Reference Manual, Century Computing, Inc., September 1993.

IX. Miscellaneous

Notification of an invalid design database will be presented to the CAPS user as text in the CAPS execution window upon the next attempted access of the design database.

In most cases, double-clicking the mouse in CAPS-generated windows will not perform the expected (or desired) selection. Use the "OK" or "Apply" buttons to activate selections.

This is an emphasis box. Be sure that you understand the comments in any such box or anything from great frustration to mild catastrophe may ensue!

When using the text editor for Ada editing, new Ada files can be created and then edited by entering the file name in the "file to edit" section of the CAPS edit selection window.

In some cases, important text information, which may require user action, is presented in the CAPS execution window. Though somewhat inelegant, this window should be frequently monitored.

When presented with an X-Window alert window, the user MUST acknowledge this window before continuing with normal CAPS operation.

In the current version of CAPS, certain prototype information MUST be entered using the Syntax Directed Editor, while certain other information MUST be entered using the Graphic Editor.

In the current version of CAPS, certain prototype information MUST be entered using the Graphic Editor, while certain other information MUST be entered using the Syntax Directed Editor.

The CAPS Graph Viewer is non-modifiable. To modify a data flow diagram, the Graphic Editor must be invoked using the "edit-graph" command in the Syntax Directed Editor's "CAPS-Cmds" pull-down menu.

A single user must not have more than one CAPS process running at a time. If multiple CAPS processes are being executed simultaneously by the same user, DATA CORRUPTION MAY RESULT!

Opening an existing prototype does not automatically invoke the PSDL Editor.

APPENDIX G: CAPS INSTALLATION GUIDE

CAPS Installation Guide

The Computer-Aided Prototyping System (CAPS) Release 1

October, 1994

CAPS

A. CAPS Automatically generates Ada code and real-time schedules

B. CAPS is designed to provide computer-aided software reuse, computer-aided project planning, automated configuration management, automated project scheduling, and automated team coordination. (These capabilites are not implemented in Release 1).

I. What You Need

To install and use CAPS, you need the following:

Platform: Sun

Workstation: SPARC-station

Operating System: SunOS 4.11 or later

Window Environment: X-Windows System version X11R4 or X11R5

You also need:

- At least 32MB of memory, to avoid paging and inaccurate timing results.
- At least 21MB of disk space
- At least 64MB of swap space is recommended, to allow large applications to be created.
- OSF/Motif 1.1.2 or later.
- Sun SPARCompiler Ada 1.1.
- TAE+ (v5.3) (Highly recommended, but not required)
- VADSedit (Highly recommended, but not required)

Note: The 21MB disk space requirement for installing CAPS does not include the space needed to install OSF/Motif, Sun Ada, TAE+, VADSedit or prototypes generated by CAPS.

II. Setting Up Your CAPS Environment

A. **Installing From Tape**

CAPS is delivered completely built and can be installed from the tape by typing the following commands:

% cd **location_of_CAPS_software** # e.g., /usr/local

% tar xvf **tape-device-name** # e.g., /dev/rst1

Write permission for **<location_of_CAPS_software>** is required.

B. Environment Setup Required

1. Other Software Systems Required

OSF/Motif and Sun SPARCompiler Ada must be installed; and the X Window System and the Motif window manager must be running before you execute CAPS. (TAE+ and VADSedit are not required, but are highly recommended). Follow the installation instructions provided with each product if they are not already installed.

2. The CAPSsetup file (C Shell Initialization Script)

The $CAPSHOME/bin/CAPSsetup file assumes the other software systems used by CAPS are at the following locations:

```
/usr/openwin
/usr/local/Ada/SunAda
/usr/local/tae.5.3
/usr/local/VADSedit
```

Edit the file to correct these path names if this is not correct for your site.

3. Modifying Users' $HOME/.cshrc Files for CAPS

To use CAPS, the following lines must be added to each CAPS user's .cshrc file:

```
if (-d location_of_CAPS_software/CAPS.RELEASE.1) then
   setenv CAPSHOME location_of_CAPS_software/CAPS.RELEASE.1
   source $CAPSHOME/bin/CAPSsetup
end if
```

C. Where To Go From Here

This completes the CAPS installation process.

The CAPS Tutorial and the CAPS User's Manual provide instructions for using CAPS.

APPENDIX H: CAPS QUICK-START GUIDE

CAPS Quick-Start Guide

The CAPS Quick-Start Guide provides the minimal information needed to develop software prototypes using CAPS. We focus on the PSDL Editor, which consists of three parts: the Syntax Directed Editor, the Graph Viewer and the Graphic Editor. The PSDL Editor allows the designer to create complete PSDL prototypes by drawing data flow diagrams and annotating them with timing and control constraints. Such prototypes can be connected to graphical interfaces using the CAPS interface editor and can be executed using the CAPS execution support system.

This guide explains how to create the prototype shown in Figure 1.

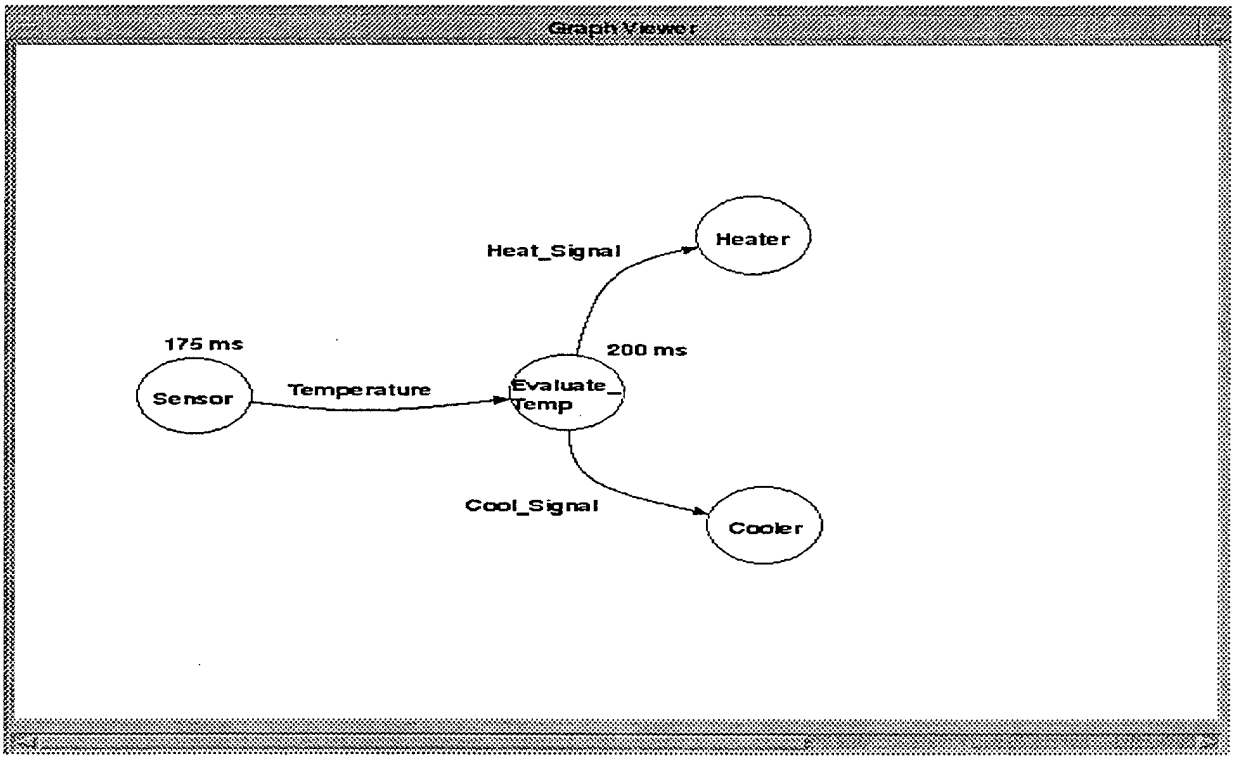


Figure 1. The CAPS Graph Viewer Showing the Temp_Controller prototype

Start the CAPS program by typing the command **caps** followed by **<return>**. The CAPS main menu will appear on your screen when the system is ready; see Figure 2.

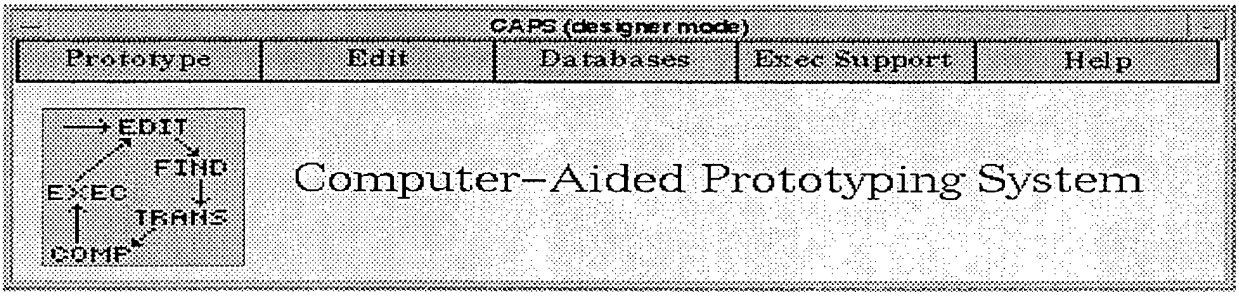


Figure 2. The CAPS User-Interface (Designer Mode[1])

1. In CAPS Release 1 the system is always in designer mode.

To start a new prototype choose the **new** option from the **Prototype** menu. (Move the mouse to put the cursor in the box labeled **Prototype** in the top left part of the CAPS main menu, push the left mouse button down, move the cursor down to the item labeled **new** in the pull down menu that appears, and release the mouse button). The window shown in Figure 3 will open up.

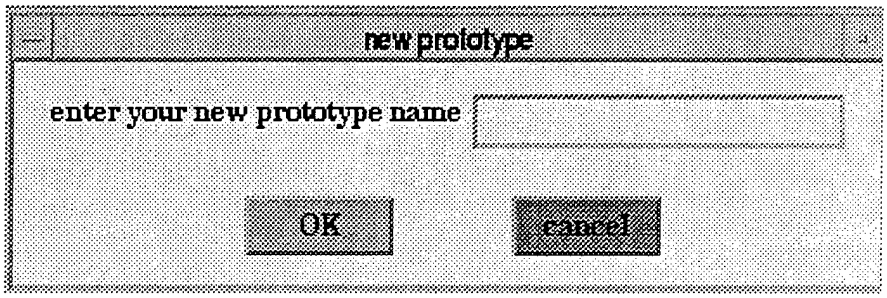


Figure 3. New prototype window.

Left click the prototype name box. It will highlight. Then type TEMP_CONTROLLER and left click your mouse on the OK button. Two windows will open up as shown in Figure 4.

In the above procedure, you must use the mouse to activate the name box and to acknowledge ok when finished. A **<return>** will not work here. In addition, CAPS is case sensitive because the underlying file structure is based on UNIX. Identifiers of two words or more should use underscores rather than spaces to separate the words.

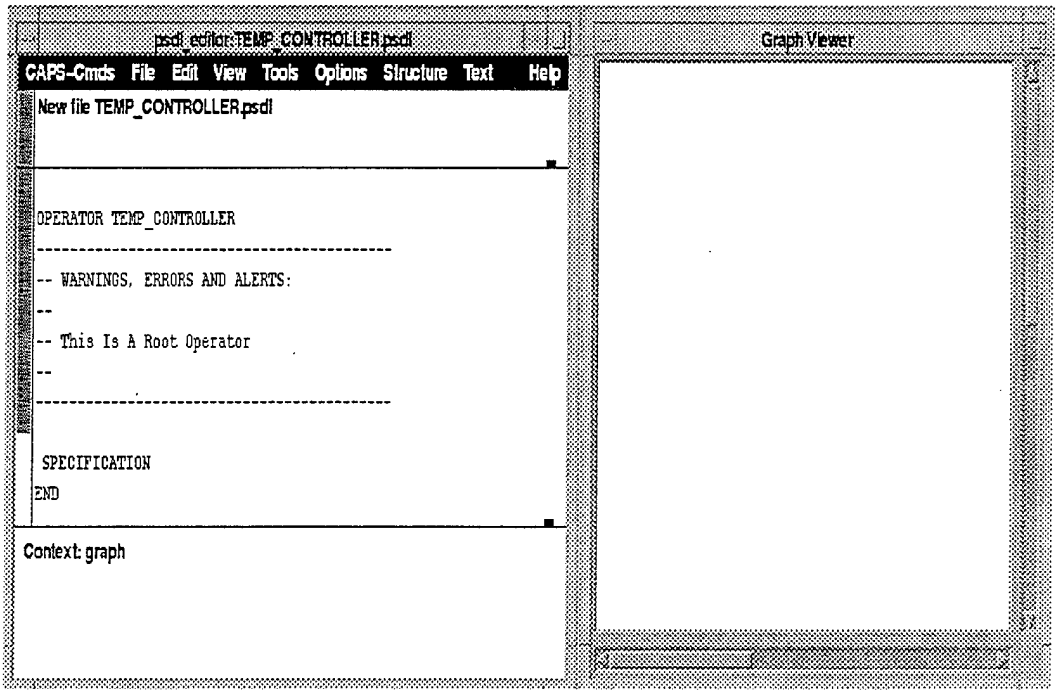


Figure 4. PSDL Syntax Directed Editor (left) and Graph Viewer (right)

Notice that the Syntax Directed Editor generates a default root operator with the same name that you provided in the new prototype selection. This operator represents the entire system to be prototyped, including simulations of the external environment in addition to the proposed software. The Graph Viewer is empty because we have not created the data flow graph for the prototype yet. That's next.

From the Syntax Directed Editor invoke the CAPS Graphic Editor by choosing the **edit graph** command from the **CAPS-Cmds** pull-down menu. The window shown in Figure 5 will appear.

Special Note: All of the commands necessary for PSDL editing and file saving are found in the **CAPS-Cmds** pull down menu, although all of the other the pull-down menus in the Syntax Directed Editor are also active. **DO NOT** use the **file** pull-down menu: the commands in this menu **will not** save a correct PSDL program.

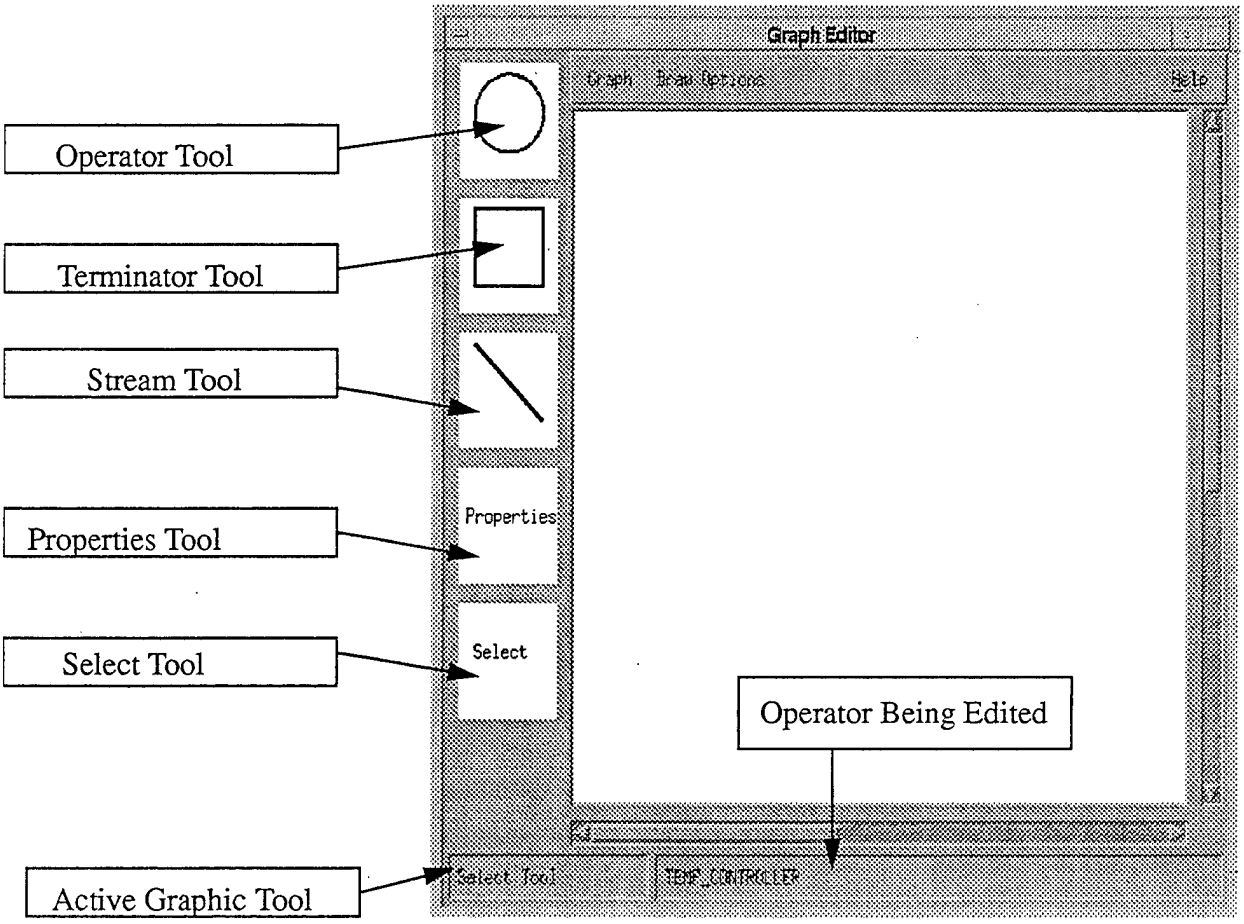


Figure 5. The CAPS Graphic Editor with notes.

Once in the Graphic Editor, the designer draws data flow diagrams, enters operator names, inserts input and output streams, and enters some of the prototype timing information. The Graphic Editor can also decompose data flow graphs into lower level diagrams. The Syntax Directed Editor is used to enter declarations of data types for streams, to complete timing information, to enter control constraints and to select implementation options. We will return to these aspects after we complete the graph of the prototype.

The CAPS Graphic Editor is used primarily to lay out the data flow design of a prototype. Operators are linked together with data streams. Names and optional timing attributes are added to operators and data streams. These attributes are **Maximum Execution Time** for operators and **Latency** for data streams. (The latency of a data stream is a lower bound on the amount of time required for transmission of data along that stream.)

The shapes and words that appear on the left hand side of the Graphic Editor (the Graphic Editor palette) are editing tools. A summary of their functions follows:

CIRCLE.....Draw circular operators to represent proposed software components.

SQUARE...Draw rectangular operators to represent simulations of external systems.

LINE.....Draw data streams.

Properties...Assign names and other properties to the selected operator or data stream.

Select.....Enable selection of an object in the graph.

The name of the active tool is displayed in the lower left portion of the Graphic Editor and the name of the operator being edited is displayed in the lower right portion.

Let's begin by putting in our software operators for the TEMP_CONTROLLER prototype. Select the Operator Tool by choosing the circle from the Graphic Editor palette on the left edge of the window, and left click your mouse. Now move your mouse cursor into the working area and left click again. A circle will appear. Now make three more. You should now have four circles in your graph work area as shown in Figure 6.

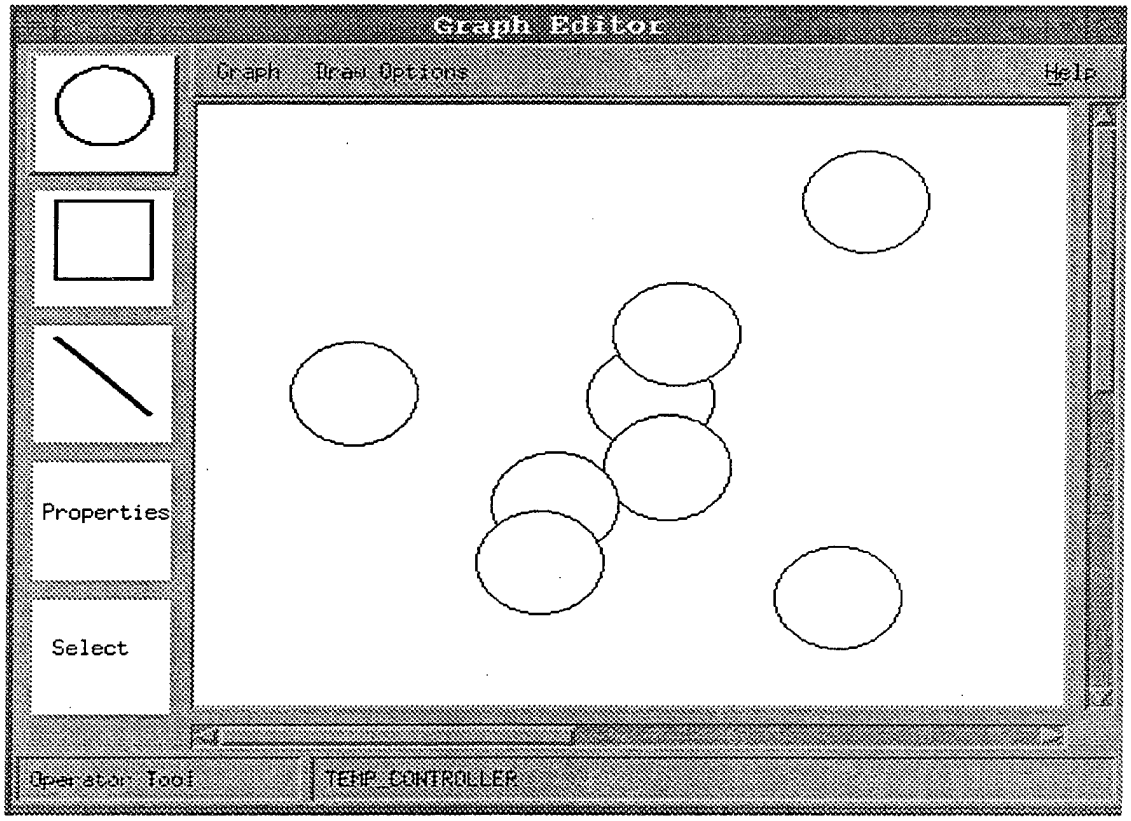


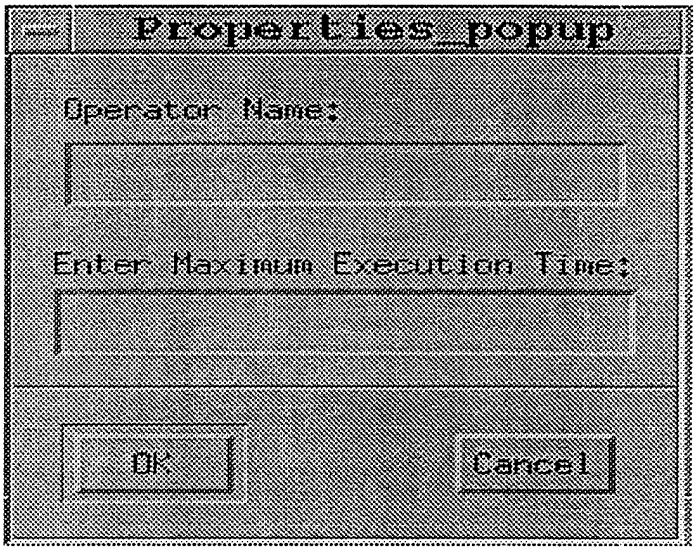
Figure 6. Too Many Software Operators

That's right. We made a few more circles than necessary. Four extra to be exact. How do we get rid of the extra circles? With the select tool! With your mouse left click on the **Select** tool. Now move the mouse unto a circle and left click again. The circle that you selected will be framed by square reference points. To delete the circle press the **delete** key or **backspace** key on your keyboard. To select another object or change the object selected simply move the mouse unto another circle and left click again.

You can also move objects around your working space with the use of the **Select** tool. Simply left click on the **Select** tool if it is not already active, and then drag the object you want to move (put the cursor on the object you want to move, push the left mouse button and hold it down

while moving the mouse to where you want to place the object, then let go of the left mouse button).

To label your software operators, click on select tool, select an object and then click on the **Properties Tool**. This will bring up the **Properties_popup** window shown in Figure 7.



A special note concerning the use of the delete and backspace in this window. The delete key deletes forward and backspace deletes backward. You can reposition the cursor with the mouse or arrow keys. You can not tab between entries and must click on OK to exit the box.

The maximum execution time is hard wired for milliseconds.

The graphic editor does not support other units of time.

Figure 7. Properties Popup Window for Operators

Enter the label for the operator in the **Operator Name** box. If the operator has a maximum execution time, enter it in the **Maximum Execution Time** box. Then click on OK. The TEMP_CONTROLLER prototype with all software operators labeled is shown in Figure 8.

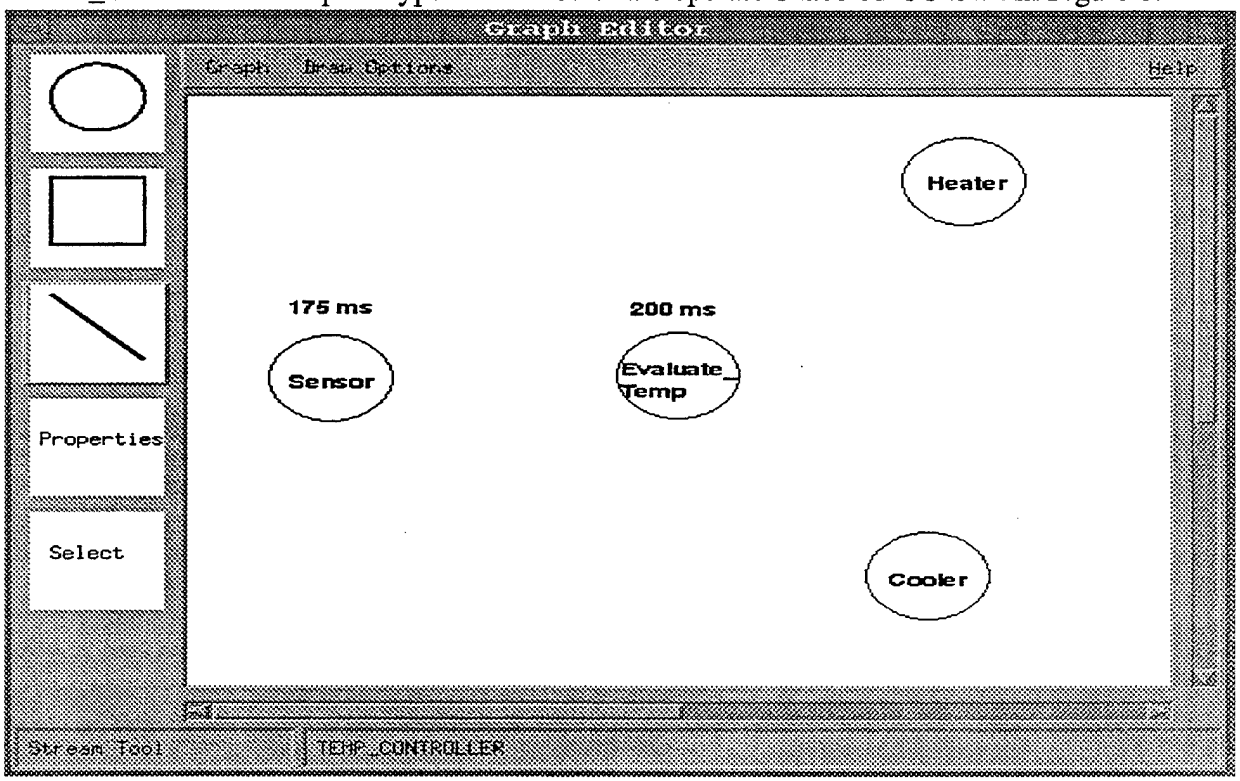


Figure 8. TEMP_CONTROLLER prototype with labels and maximum execution times.

The units of milliseconds (ms) are generated automatically. Do not enter units in the **Properties_popup** dialog box or your maximum execution times will not appear on the graph. Maximum execution time labels can be selected and moved like all other units. Be careful that

you do not misplace them however. The object associated with each label should be obvious from its position.

Next we will use the **Stream Tool** to enter data streams between operators. To use the tool move your mouse to the **Line** icon in the palette and left click. To draw a data stream place the cursor in the source operator and left click. This will start the stream. Then move the mouse to the destination operator and left click again. This will end the data stream. Data streams can originate or terminate in an object or outside of an object depending on the nature of the stream, i.e., internal or external. If curved lines are needed you can create guide points along the line by clicking the left mouse button between the beginning object and the ending object. When the line is completed it will curve to follow the guide points. Figure 9 shows the TEMP_CONTROLLER prototype with data streams entered.

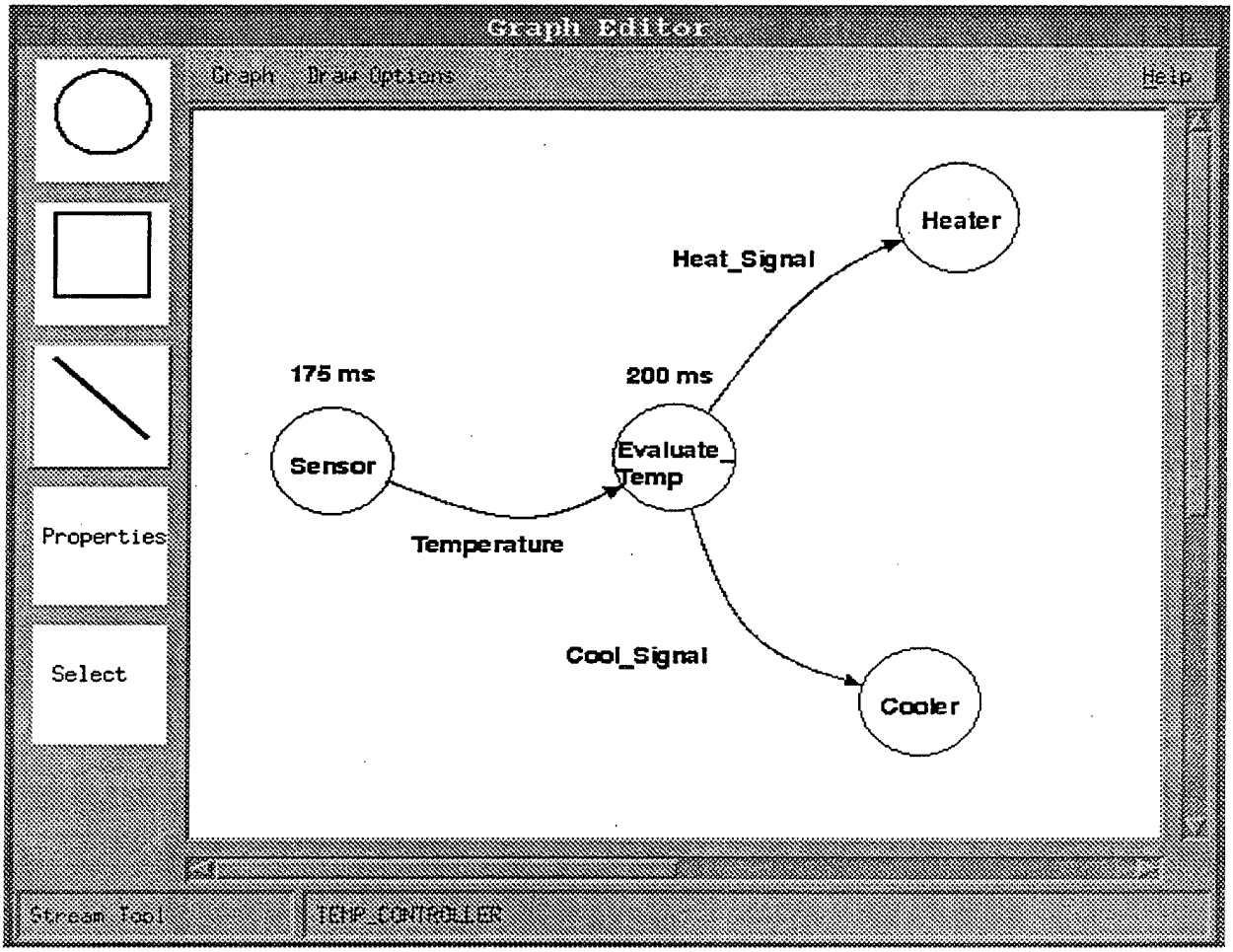


Figure 9. TEMP_CONTROLLER with data streams entered.

The data streams have properties similar to operators: they have object names and a latency attribute. The **Properties_popup** window for data streams is shown in Figure 10. Again, the latency attribute is measured in ms by default. The functional restrictions on data entry for this window are the same as in the **Properties_popup** window for operators.

Note that the editor is smart enough to display the right **Properties_popup** window.

There are two additional buttons in the data stream popup box for indicating whether this data stream is a state stream or non-state stream. In the graphics editor the default is non-state stream. In release 1 of CAPS the data types of all streams and the initial values of state streams are defined in the Syntax Directed Editor.

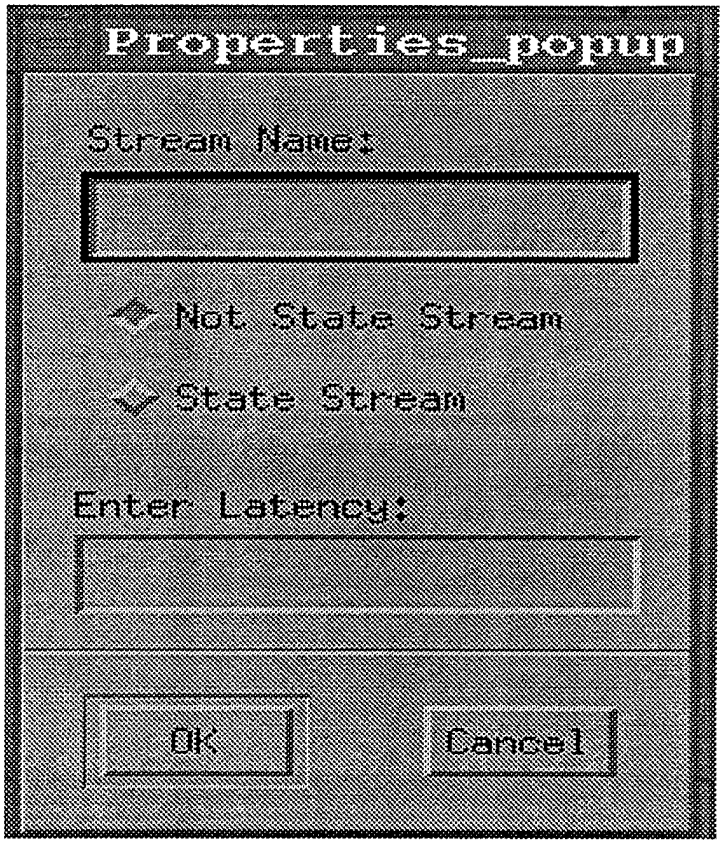


Figure 10. Properties box for data streams.

Data streams labels can be moved like other objects by using the select tool and then dragging with the mouse. A note of caution however: if a data stream name or latency label is deleted the entire data stream is deleted as well.

Our prototype data flow graph is complete and there is little left to do in the graphic model. We will now go back to the Syntax Directed Editor. Stand by for some pleasant surprises!

To return to the Syntax Directed Editor from the Graph Editor pull down the **graph** menu on the menu bar and select **return to Syntax Directed Editor**. The Graphic representation of your prototype will be saved automatically. The other choices on this menu are self-explanatory.

The first thing that you should notice when you return to the PSDL Editor (see Figure 11) is that type declaration stubs have appeared for all of the data streams that you created in the Graph Editor in alphabetical order. Also there should now be control constraint and operator specification stubs for all Operators that you created in the graphic editor. These are also in alphabetical order.

The Syntax Directed Editor knows the syntax of PSDL and will show you all of the legal options at each point in your PSDL program. These prompts appear in the grammar menu at the bottom of the PSDL editor window. In the Editor the cursor appears as a "caret". You can move it with the mouse or with the arrows on your keyboard. When you move the cursor to the end of SPECIFICATION the grammar menu in Figure 12 appears at the bottom of the Syntax Directed Editor window. These are the types of information that you can add to the operator specification at the current position of the cursor. Some of this information is input via the Graph Editor and automatically added to the PSDL by the Syntax Directed Editor, i.e., o_inputs_list and o_outputs_list. You can enter these things in the SPECIFICATION but if the information that you enter is inconsistent with that entered in the Graph Editor the Syntax Directed Editor will change it to make it

consistent. The grammar menus at the bottom will automatically adjust themselves based on where the cursor is placed.

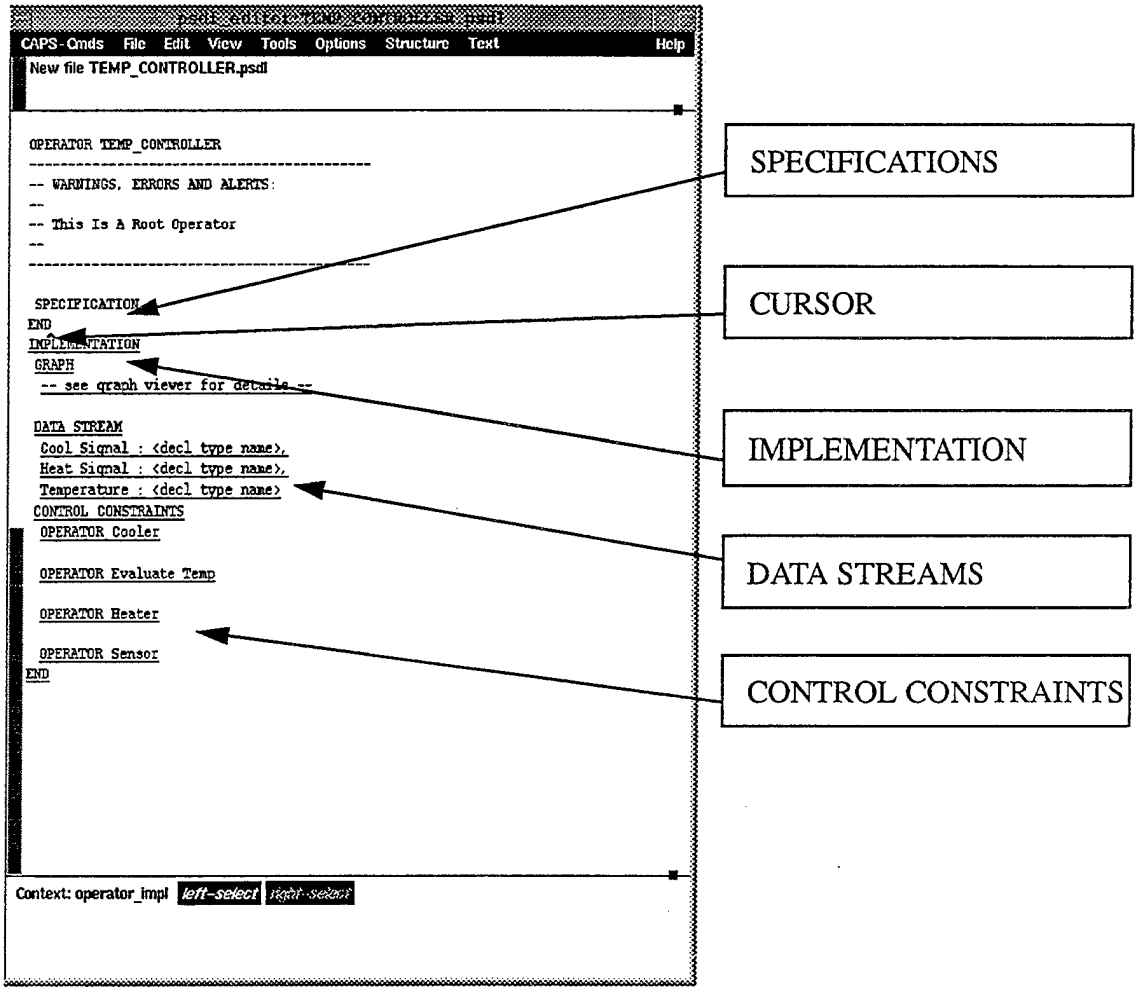


Figure 11. PSDL Editor after Graph is complete

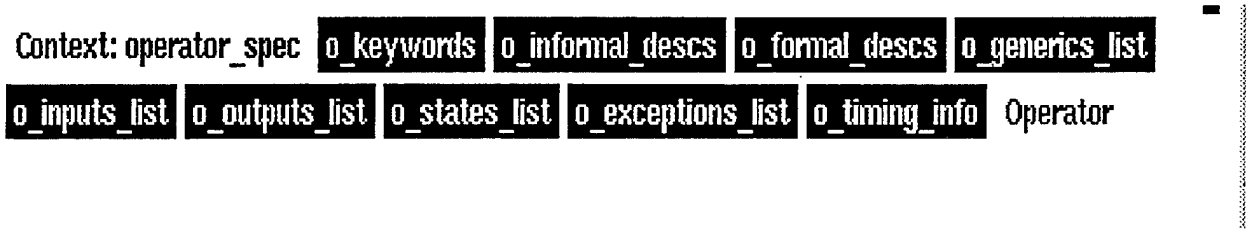


Figure 12. Specifications Grammar Menu.

Since we are positioned at the top level of the prototype the only things that we will add to the SPECIFICATION are keywords. Select o_keywords and the menu shown in Figure 13 will pop up.

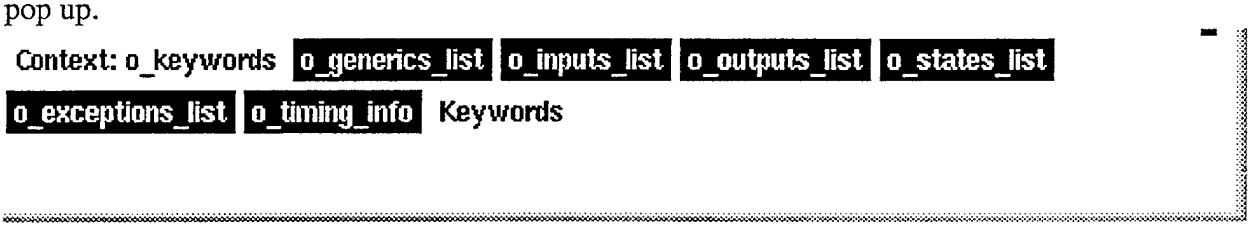


Figure 13. Keyword Grammar Menu.

When you select an item from the grammar menu, the corresponding section of your PSDL editor will change. A template for the item will appear, containing a "place holder" for each missing part of the selected grammar item, surrounded by square ([ ]) or angle brackets (< >). Any time you see square brackets, the item described inside of them is optional. Click on the Keywords option in the Keyword grammar menu and the [ ] will be replaced by KEYWORDS <identifier>. You can now type in keywords. Ensure that keywords are separated by commas and do not contain embedded spaces. Words in multi-word keywords can be separated by underscores. When you are finished you must hit **<return>**. This will save your entries. If you get an optional empty place holder where you do not want one, you can eliminate it by moving the cursor up or down with the arrow keys.

If you make a mistake when entering data you can delete individual characters with the **delete** and **backspace** keys. To delete an entire grammar item, move the cursor to highlight the part that you want to delete and hit **<ctrl> <shift> <k>** simultaneously, or choose **cut_structure** from the **Edit** pulldown menu. If you have trouble recovering from a syntax error, deteing the current grammar item will get rid of the offending part, and you can start over with a clean slate. To diagnose a syntax error, you can type **<meta> g** to go to a given line number (a window pops up asking for the line number). The **<meta>** key is similar to the **<shift>** and **<control>** keys. It is labeled with a diamond (<>) on some keyboards.

Other items that you can add to the SPECIFICATION include informal descriptions and state declarations. The informal descriptions will appear in { } and are similar to user/programmer comments in other programming languages. They are for human consumption only and are ignored by the PSDL compiler. The state declarations are necessary to initialize data streams that require initial values. In CAPS release 1, state declarations for state streams should be created in the Syntax Directed Editor before you enter the corresponding state streams in the Graphic editor.

You can look at the graph associated with an operator at any time by clicking anywhere within the PSDL declaration of the operator. The Graph Viewer will show the corresponding graph. The graph viewer window will be empty if the cursor is within an operator that does not have a graph. (One minor discrepancy related to the graph viewer: in CAPS release 1 the icon for the graph viewer will be labeled "graph-edit" when the graph viewer window is closed).

Now we will enter type declarations for the data streams. Position the cursor before, in or after the <type declaration> component of the data stream. Left click and a menu bar of standard defined types will activate at the end of the PSDL Editor. Click on the type declaration that you want and the data stream will be declared to carry that type. User defined types can also be accessed from this menu bar and typed in by the user.

The PSDL editor provides automatic propagation of type declarations throughout the PSDL program when you define the data streams. This is illustrated in Figure 14 below. Also illustrated in Figure 14 is the PSDL Editor's ability to correctly classify data streams as either input or output within the Operator Specifications found in the top of PSDL program. The Maximum Execution Times that you entered in the Graphics Editor are displayed there as well.

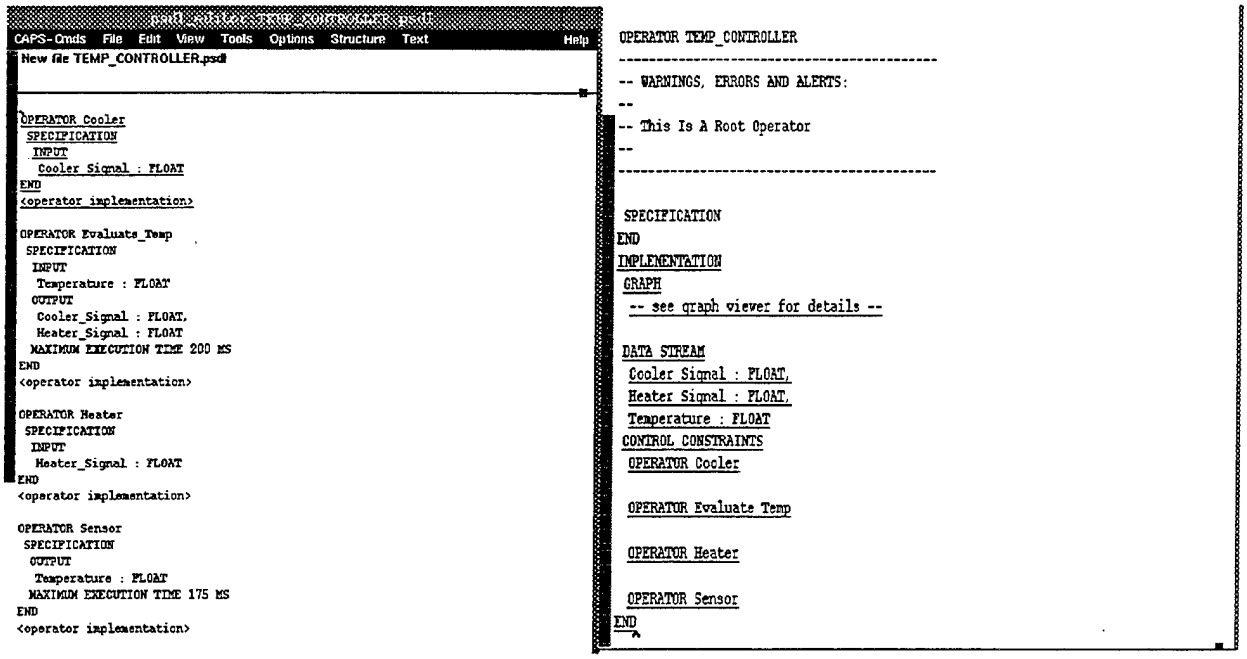


Figure 14. Complete PSDL program showing type declaration propagation.

Control Constraints can be modified directly from the PSDL Editor by using the mouse or arrow keys to activate PSDL menu bars (See Figure 15 below). Control constraints are not used in our simple example.

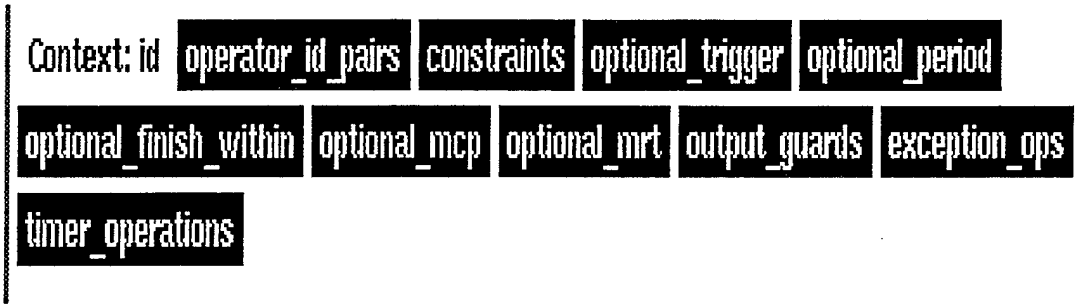


Figure 15. Control Constraints Menu Bar.

Our simple example is complete. But what if we wanted to decompose one of our operators? How would we do it? How would the Syntax Directed Editor react? Lets see.

First select "edit-graph" from the "CAPs-Cmds" in the Syntax Directed Editor menu bar. We should get a window that looks like Figure 9, which is repeated below. Select the Sensor Operator and then select Decompose from the Graph menu. This will automatically open up a Graph Editor that looks like Figure 16 below.

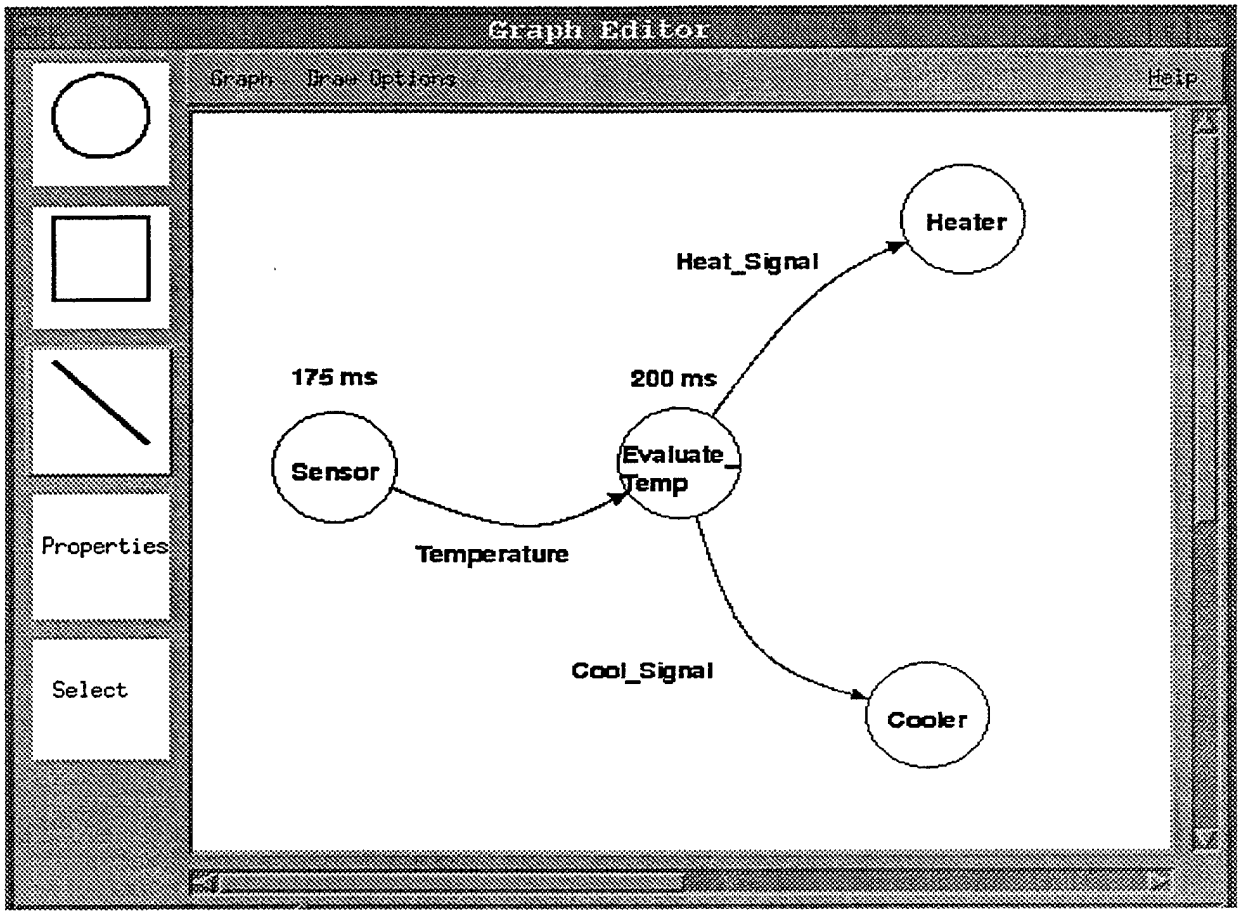


Figure 9 (repeated)

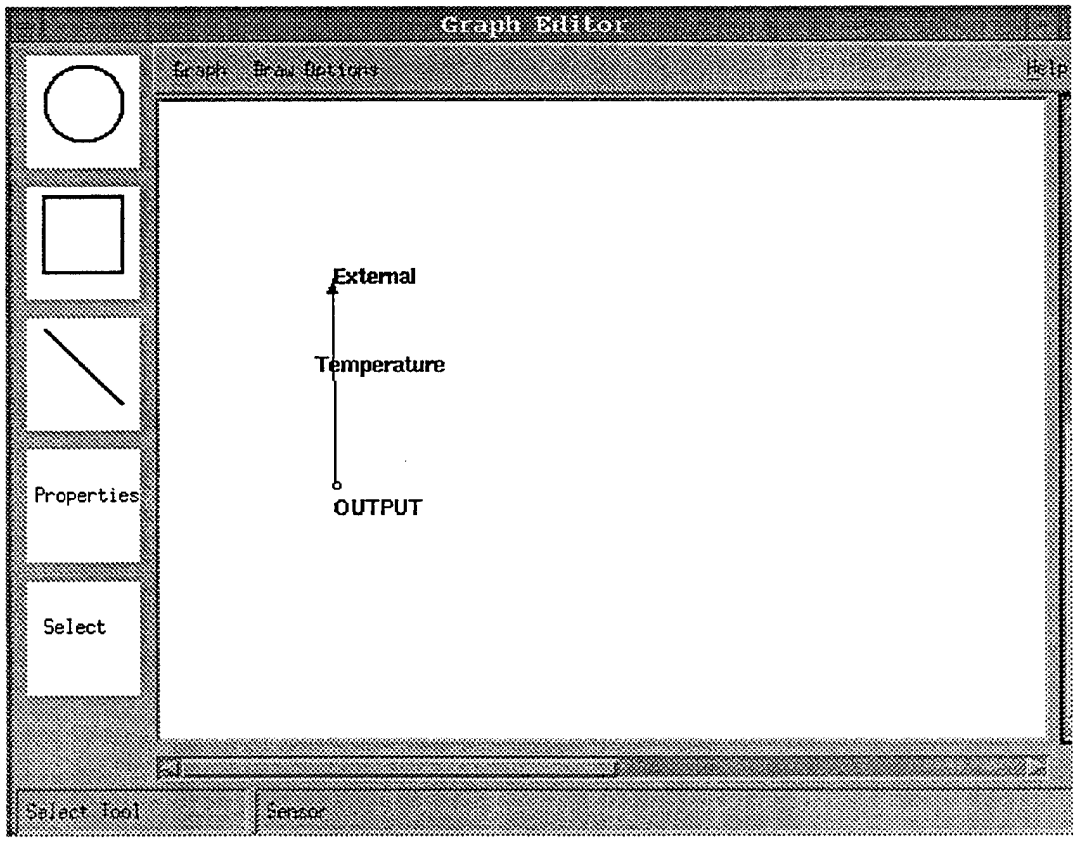


Figure 16. The Decomposition of the Sensor Operator in the Graphic Editor

The Graph Editor shows the data streams entering and leaving the sensor operator as a reminder of the required interface for the decomposition. The data stream "Temperature" is shown as external output in Figure 16 above because the decomposition of the sensor operator must produce this stream as its only output. No inputs are shown in this case because the sensor operator has no input streams.

Now decompose Sensor by graphically creating a network of lower level operators and streams that collectively realize the Sensor Operator. An example is given in Figure 17 below.

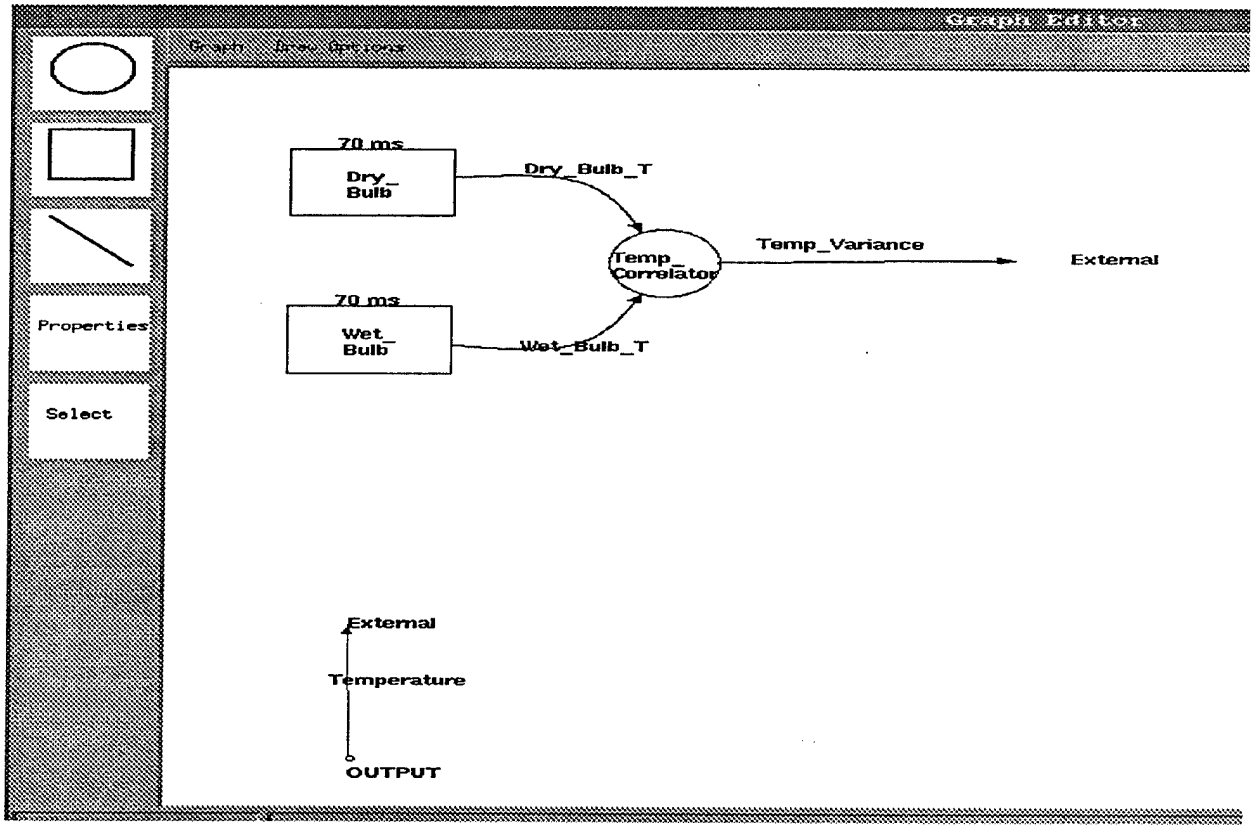


Figure 17. The Sensor Operator shown with complete decomposition.

Click on **Return to SDE** and move the cursor to the IMPLEMENTATION GRAPH. You can then view the Graph for the entire prototype, which will look like Figure 18. Note that the sensor operator is now a doubled circled to show that it has been decomposed into a lower level network of operators.

The PSDL file in the Syntax Directed Editor will reflect the changes made and will contain new operators and data streams corresponding to the newly created decomposition. The declarations and control constraints for these objects can be modified as previously described in this guide.

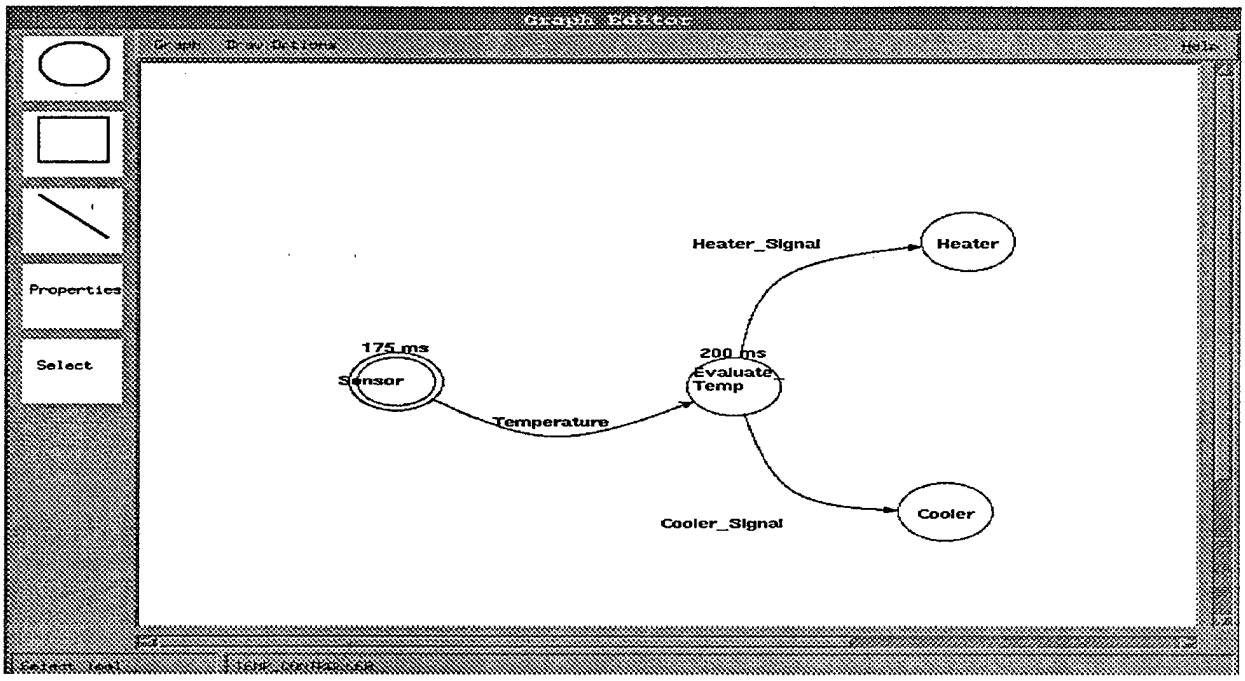


Figure18. The TEMP_CONTROLLER prototype with Sensor marked as composite.

You should now be able to build a prototype in the graphic editor, make type declarations and define constraints in the PSDL Editor or Syntax Directed Editor. The following table provides a quick reference on which type information can be entered into each editor:

TABLE 1.

Information entered via Graphical Editor	Information entered via Syntax Directed Editor
operators and streams of the graph	control constraints: triggers, execution guards, output guards timing constraints, exception guards, timer operations
operator names	timer declarations
operator maximum execution times	implementation type selections (Ada or PSDL)
operator color and shape	data type definitions
data stream names	data stream types
data stream latencies and state property	state declarations and initial values

Once the prototype is complete save your work from the Syntax Directed Editor by choosing "PSDL-Save" or "PSDL-Save-Exit" from the "CAPS-Cmds" pull down menu on the menu bar.

You can then Translate and Schedule your prototype and begin Coding and Compilation. The operators that are not decomposed into lower levels need to be implemented in a programming language; in CAPS release 1 that language must be Ada. To enter code for these atomic operators choose **Ada** from the **Edit** option in the CAPS main menu. (The option to retrieve reusable software components from the software base is not implemented in CAPS release 1). To translate, schedule, compile and execute choose the corresponding items from the **Exec Support** option in the CAPS main menu.

APPENDIX I: CAPS HOME PAGE

Netscape [Computer Aided Prototyping System Research Project]

File Edit View Go Bookmarks Options Directory Window Help

Home Edit Reload Open Print Find

Location: http://www.caps.cs.nps.navy.mil/

What's New! What's Cool! Handbook Net Search Net Directory Software

CAPS TABLE OF CONTENTS

DISCLAIMER

- Introduction
- What's CAPS?

References:

- Journal Papers
- Conference Papers
- Books
- CS Department

Sponsors:

- AJPO
- Army Research Lab
- Army Research Office
- DISA
- National Science Foundation

CAPS Applications:

- Tomahawk Missile
- SIDS Wireless Acoustic Monitor (SWAM)
- Temperature Controller
- C3I
- Fish Farm

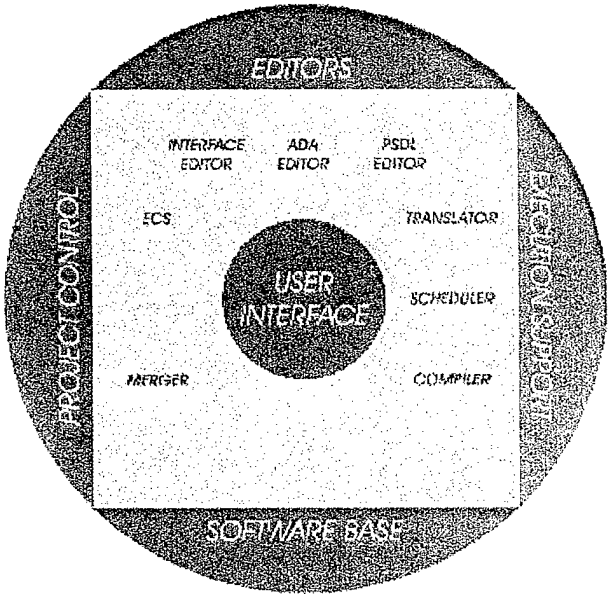
Supports:

- FAQs

Point of contact:

- caps@cs.nps.navy.mil

Computer Aided Prototyping System (CAPS)



SPONSORED BY

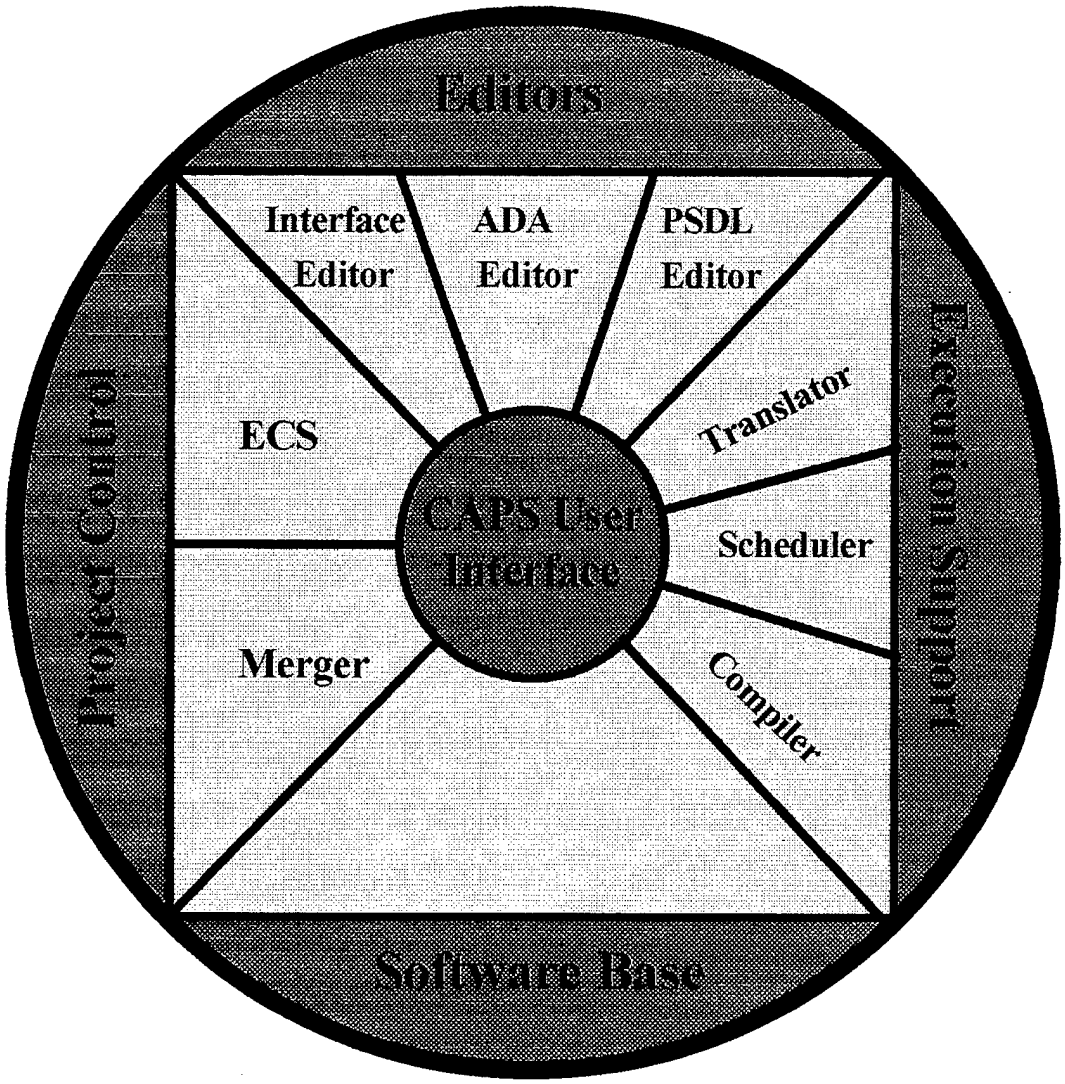
NATIONAL SCIENCE FOUNDATION

ARMY RESEARCH OFFICE | ADA JOINT PROGRAM OFFICE

Document Done

APPENDIX J: CAPS CD-ROM MULTIMEDIA DEVELOPMENT PLAN

THE DESIGN OUTLINE FOR THE CAPS MULTIMEDIA PRESENTATION



Submitted to Professor Luqi
Software Engineering Group
Naval Postgraduate School

ACKNOWLEDGMENT

This document was prepared for the Computer-Aided Prototyping System technology transfer effort. The design of this presentation is a joint effort between the Naval Postgraduate School, Software Engineering Group and Texas Tech University, College of Engineering. Principal architects of this project are:

Hua Harry Li, Ph.D.
Associate Professor
Computer Science Department
College of Engineering

LT. Robert Cooke
Information Technology Management
Naval Postgraduate School

LT. Sotero Enriquez
Maj. George Whitbeck
Ltjg. Erdinc Yetkin
Computer Science Department
Naval Post Graduate School

TABLE OF CONTENTS

I. INTRODUCTION 1

1.1 Background Information 1

1.2 The Objectives of the Project 2

II. THE OUTLINE OF THE PROJECT 2

2.1 The Overview 2

2.2 The Title and the Root Page Design 3

III. The Branch of Overview of CAPS 4

IV. The Branch of CAPS Technical Material 6

V. The Branch of Success Stories 8

VI. The Branch of Technology Transfer and Adoption 8

VII. The Development Personnel 9

VIII. The Deliverables 9

I. INTRODUCTION

The purpose of this document is to provide a design guideline for the multimedia presentation project for CAPS (Computer Aided Prototyping System). CAPS is an excellent tool for the rapid software prototyping and construction, it provides an computer automated source code generation, reusability, standardization, and verification. CAPS was the result of the research and development work sponsored by both National Science Foundation and the Department of Defense. Since its inception, CAPS has been applied to several major projects with significant benefits to the software development, cost reduction, and delivery on time. CAPS is a dual-use technology and it is now under the process of technology transfer. In order to assist the technology transfer, this project for developing a multimedia presentation on CD-ROM is proposed. Multimedia presentation with Video Clips, 3D animated computer graphics, color photos, documentation, block diagrams can be organized in an interactive fashion which allows user to browse through in real time. The CD-ROM multimedia presentation can be a highly effective tool to reach a broader audience, and it will be used as a supplementary material in addition to the technical documentation.

During the past few years, we have been witnessing the rapid growth of multimedia computing. Data presented in various formats including graphics, flow diagrams, videos and audio helps to bring an interactive, user friendly computing environment to many different applications. One of the impacts of multimedia computing is evidenced by the fast acceptance and adoption of Internet related communications
and computing. The potential of multimedia related applications of government or private industry sector related applications has been projected to reach about $4 billion by the year of 2000.

In this project, we will use the multimedia presentation on CD-ROM for assisting duel use technology transfer. In particular, the technology transfer of the CAPS system.

1.1 Background Information

This project is designed to capitalize on the technology and to design, develop, and create a multimedia presentation on CD-ROM. CD-ROM drives have become a standard feature of PC (personal computer). In addition, a recent study published by IEEE Spectrum (May issue, 1996) indicated that every 38 high school students have one multimedia computer. The need, the equipment capability, and the wide availability of the needed hardware for running multimedia presentation are already there.

The cost of making a CD-ROM presentation is mainly the cost of intensive human design, script preparation, video taping, 3D animated computer graphics, and programming. But the end product, a CD-ROM itself, is relatively inexpensive. Once the master CD is produced, each copy of the CD-ROM costs about $6 apiece.

1.2 The Objectives of the Project

The objective of this project is to create a multimedia presentation on CD-ROM to assist the CAPS (Computer Aided Prototyping System) technology transfer. We propose to design, develop, and to create a multimedia presentation with our in-house designed animated 3D computer graphics, video clips taken from the design team or from the public relation office of the Naval Post Graduate School, color photos, selected documentation of CAPS, and diagrams, are to be organized in an interactive fashion to allow users to browse through the material. It has been envisioned that this multimedia presentation can also be used as a training tool, as a supplementary material, in addition to the technical manuals.

It is the intention of this work to also provide a demonstration project for the possible use in other areas, such as training, distance learning, distance education, as well.

II. THE OUTLINE OF THE PROJECT

This multimedia presentation project, known as M-CD thereafter, consists of 3 phases: (1) the preliminary design phase, which will allow us to put the basic building blocks together; (2) the refinement phase, which is designed for further quality improvement and enhancement after the preliminary testing, then (3) the phase for finalization and mass production. This documentation is basically developed to describe the first phase of the project.

2.1 The Overview

The M-CD project consists of 4 major blocks or branches described briefly as follows,

1. The branch of "overview of CAPS," where all the information related to the research personnel, design team, funding agencies, will be introduced in this branch. Most of the material of research personnel, the design team will be given in the form of short video clips, about 10 to 13 seconds apiece.

2. The branch of "CAPS technical material," including the explanation of CAPS, why do we need CAPS for software development, and how can we benefit from CAPS etc.

3. The branch of "technology transfer and implementation," which include the description of how to obtain a copy of CAPS, its Internet access, world-wide-web home page information, and distribution information.

4. The branch of the "success stories." This part of the presentation will present some actual projects which demonstrate the success of using the CAPS tool. This branch will consists of video clips from the managers of various software development projects, who have gained first hand experience and who have benefited from the utilization of the CAPS tool.

A diagram is given in Fig.1 to illustrate the structure of the design.

2.2 The Title and the Root Page Design

One of the most important parts of the presentation is the title page, which is described in detail in this section.

1. The title page of M-CD contains text information in the layout format as follows

Automated Software Construction and Verification with CAPS Tool

Program Director: Luqi, Ph.D.
Principal and Co-Principal Investigators: Luqi, Berzins, Shing
Computer Science Department
Naval Post Graduate School
Monterey, California 93943
E-mail: Luqi@cs.nps.navy.mil

This Multimedia Presentation on CD Is Prepared by Hua Harry Li, Ph.D.
Computer Science Department, Texas Tech University
with Assistance from
LT. Robert Cooke, LT. Sotero Enriquez
Maj. George Whitbeck and Ltjg. Erdinc Yetkin
Naval Post Graduate School

The background of the title page: **a color photo** of the logo of Naval Post Graduate School, digitized in .TIF or .GIF file format, then duplicated to form a tile pattern at the background, and a piece of music playing along with the title page.

2. The animated 3D Computer Graphics page, which is placed right after the title page. It contains a piece of background music and the animated caption:

CAPS TOOL

with 3 to 4 moving point light sources to create good light effects.

3. The master page of the presentation, which contains a bank of stamp size color photos, in particular these color photos are :

3.1 **A color photo** of CAPS group, or a group of Professor Luqi's team members, students, in uniform.

3.2 **A color photo** of an instructor or student who is lecturing CAPS, (HL: George's photo in uniform.)

3.3 **A color photo** of a group of students (2 or 3) in front of computer working on something (HL: in dress-down fashion, with shorts etc., no uniform.)

3.4 **A color photo** of Professor Luqi working in front of a computer, or holding a technical report, standing and reading with her office background.)

3.5 **A color photo** of other senior research personnel at meeting around a table.

3.6 **A color photo** of a team member who is working on technology transfer (HL:Robert's photo, dressed in uniform working on a stack of documentation on the desk.)

Then this bank of color photos is placed together with the following text layout to form an array of 4 entry points leading to each major branch:

(a) Overview of CAPS.
(b) Dual-use Technology Transfer.
© Success Stories.
(d) Technical Materials About CAPS.
(e) Background Information.

III. THE BRANCH OF OVERVIEW OF CAPS

Under this branch, we will have the following material :

The submaster page with **color photo** of the CAPS group, Professor Luqi, Professor Berzins, Professor Shing having a meeting around the table and apiece of background music, and the text layout with each as an entry point.

Overview of CAPS

(a) CAPS system.
(b) The design team.
© NSF, DoD and Other Sponsors.
(d) Design Projects.

1. The page of "CAPS" with a **color photo** of the Web home page and background music and the text layout as follows,

The CAPS: Computer Aided Prototyping System.

(a) The Purpose of CAPS.
(b) The Features of CAPS.
© The CAPS Process.
(d) Typical Applications of CAPS.
(e) Tutorial Example of CAPS.

1.1 The page connected to the upper level (a) with the background music and the text layout as follows,

> ***CAPS is a carefully designed, well documented, and fully tested Computer Aided Prototyping System for software design, construction, project management, and verification.***

Then a **video clip** (Professor Luqi's or one of the students, Sotero's 10 second introduction. The speech is ended with "..... the purposes of CAPS are" then we start the following page with the layout,

> ***Purposes***
>
> ***(a) Clearly define the software system requirement.***
> ***(b) Achieve software development reduction.***
> ***© Perform system acquisition and integration.***
> ***(d) Allow early quality assessment and improvement.***
> ***(e) Assist exploratory design and innovation.***
> ***(f) Employ system and process re-engineering.***
> ***(g) Support requirement evolution.***

1.2 The features of CAPS has text only layout as follows:

> ***Features of CAPS:***
>
> ***(a) Automated computer aided code generation and Verification.***
> ***(b) Better Project Management Tool for Cost Reduction and Project Budget Overrun Prevention.***
> ***© Graphics user interface and timing control.***
> ***(d) Easy to learn, easy to use for real software development project.***

1.3 The tutorial example of CAPS with **Video Clips** (10 to 13 second, George explain how the tutorial works) then with the following text layout:

> *CAPS Example*

Then followed by a **Video Clip** of program demo taped from a computer screen using PC to TV equipment.

2. The page of "The Design Team" with a **color photo** of program director, PI, Co-PI, and a **color photo** of the team. Then each of this photo is arranged as an entry point to the **video clips** of (a) Professor Luqi (13 to 15 second), (b) PI, © Co-PI, (d) other team member to introduce themselves and briefly explain what their roles are in CAPS project.

3. The page of "The Related Work Sponsored by NSF," with the text layout as follows,

Extensive Experience and Expertise (I)

Program on Software Engineering Sponsored By NSF
(a) Project XXX, title of the project and year.
(b) NSF Workshop in 1993, title of the workshop, and date, organizer: Professor Luqi.
(b) NSF Workshop in 1994, title of the workshop, and date, organizer: Professor Luqi.
(b) NSF Workshop in 1995, title of the workshop, and date, organizer: Professor Luqi.

4. The page of "The Software Development Projects" with **color photo** of book or published papers and the following text layout,

The Extensive Experience and Expertise (II)

(a) Software development projects sponsored by DoD.
(b) XXX prototype systems and design tools developed since 198X.
© Extensive publications in IEEE Software Engineering, IEEE Software, and other professional journals.
(d) Books, book chapters.

IV. THE BRANCH OF CAPS TECHNICAL MATERIAL

Under this branch, there will be the following material :

1. A submaster page with a color photo of a World-Wide-Web home page and video clips (The explanation of the technical material) text layout as follows,

CAPS Technical Material

(a) Internet connection and www site.
(b) CD-ROM distribution of CAPS tool.
© Technical Reports and Publications.
(d) Tutorial Material.
(e) Learning how to use CAPS.

2. The page of "Internet connection," with a **color photo** of the Web page, and text layout as:

The Internet Connection of CAPS

(a) WWW address, http://wwwcaps.cs.nps.navy.mil and e-mail address: Luqi@cs.nps.navy.mil
(b) the brief description of the material at this site.

3. The page of "CD-ROM" distribution with text layout as

CD-ROM Distribution of CAPS tool

Sponsored by DoD, a free CD-ROM can be obtained from XXX.

4. The page of "Technical reports and publications" with a **color photo** of the stack of technical reports then the text layout as follows,

Technical Materials and Publications

(a) the point of contact of the technical report.
(b) the point of contact of the technical publication.
© NSF, IEEE publications, and others.

5. The page of "The Tutorial Material," with a color photo of the tutorial material, and **video clip** (The person who is responsible for the preparation of the tutorial) then the text layout of:

The Tutorial Material

(a) point of contact of the tutorial material.
(b) the scope of the tutorial, training session, etc.
© the estimated time needed for the training and adoption.

6. The page of "Learning CAPS" with text layout as follows,

Learning How to Use CAPS

(a) The CAPS requirement representation.
(b) The CAPS design representation.
© The guidelines for setting up the CAPS process.
(d) The guidelines for capturing critiques.
(e) The guidelines for modeling large systems.
(f) The guidelines for modeling real-time requirement.
(g) Exploring hardware/software trade-offs.
(h) The guidelines for tracking requirements and changes.

V. THE BRANCH OF SUCCESS STORIES

This part of the material will be provided by LT Robert Cooke who is currently working for Professor Luqi on this project. We will need at least two stories and two **video clips** for the stories to fill this part. We are especially interested in using the real design examples which demonstrate cost reduction, better project management, and better quality software product.

VI. THE BRANCH OF TECHNOLOGY TRANSFER AND ADOPTION

This part deals with the issue of dual-use technology transfer and adoption.

1. A submaster page with a color photo of CAPS user manual, installation guide, and other technical materials, and a **video clip** (Robert Cooke, 15 seconds) to explain what is dual-use technology transfer and its significance, then a text layout

Dual-Use Technology Transfer and Adoption

(a) Dual-use Technology Transfer.
(b) Intellectual Property Issue.
© On-site and Off-site Training and Workshops.
(d) Installation Issue, hardware platform, operating system requirement, and Programming Languages.

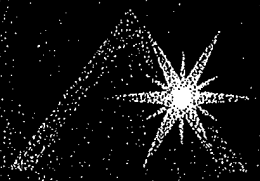
The above entries will be further explained at the next page connecting to the entry point. The detailed material will be worked out at the 2nd phase of the project.

VII. THE DEVELOPMENT PERSONNEL

This multimedia project is designed and to be developed by Professor Hua Harry Li, Computer Science Department, College of Engineering, Texas Tech University, Lubbock, TX 79409. Professor Li's group, in particular, a Ph.D. student Daniel Panturu will also work on this project. LT. Robert Cooke from Naval Post Graduate School will work on-site under the supervision of Professor Luqi at the Naval Post Graduate School to coordinate the project and to provide all necessary materials, especially the color photos, video clips, some music clips. In addition, LT. Sotero Enriquez, Maj. George Whitbeck and Ltjg. Erdinc Yetkin will assist the project development, prepare equipment and demonstration.

VIII. THE DELIVERABLES

This project is proposed as a 3-phase project. At the preliminary phase, we will finish the draft design described in details in this report and produce a CD sample by the 2nd week of September for demonstration and further discussion. The demonstration will be made on-site at Naval Post Graduate School. Then at the second phase, improvement, enhancement, and refinement will be performed based on the feedback from the demonstration. Finally, at the last phase of this project, a master CD will be prepared for mass production.



RAPID SOFTWARE CONSTRUCTION AND VERIFICATION WITH CAPS TOOL

3D Animated Graphics

Background Information

- Brief History of Software Development
- Program Director and PI's, Co-PI's Bio
- Design Team
- Funding Agencies

CAPS Technical Material

- What is CAPS
- Brief History of CAPS
- Why CAPS
- How CAPS Can Help

Adoption and Implementation

- Internet access
- WWW Home Page
- Documentation and Some Code Distribution

Success Stories

- Story 1
- Story 2

APPENDIX K: CAPS INFORMATION BROCHURES

Software Engineering Research

Software Specifications and Computer-aided Software Evolution. Formal methods and automated decision aids have the potential to substantially reduce costs and increase the quality of delivered software. We have developed tools specifically designed for large-scale applications that include parallel, distributed and real-time systems.

Fundamental theory and practical methods for:

- partial automation of software development
- correct combination of multiple changes
- computer-aided design in development and maintenance of large software systems.
- change merging for specifications and software prototypes of real-time systems

Computer-aided Prototyping of Real-time Systems. Rapid prototyping of hard real-time systems via a computer-aided prototyping system (**CAPS**) is based on a prototyping language with module specifications for modeling real-time systems and combining reusable software. These tools make it possible for prototypes to be designed quickly and to be executed for validating the requirements.

The research focuses on:

- automated methods for retrieving, adapting and combining reusable components
- establishing feasibility of real-time constraints via scheduling algorithms and simulating unavailable components
- automatically generating translators and real-time schedules for supporting execution;
- constructing a prototyping project data base using derived mathematical models;
- providing automated design completion and error checking facilities in a designer interface;
- establishing a convenient graphical user interface for design and debugging.

CAPS Web Site: The CAPS project maintains a world wide web home page at URL:

http://wwwcaps.cs.nps.navy.mil

Software Engineering Faculty

Valdis Berzins, Professor

Automated decision support for developing and assessing software requirements. Software merging for computer-aided maintenance. Automatic program generation from problem descriptions.

Luqi, Professor

Risk reduction for real-time systems via computer-aided prototyping. Elicitation and refinement of requirements based on prototypes. Legacy and system re-engineering and conversion to maintainable Ada.

Man-Tak Shing, Associate Professor

Algorithms and tools to support computer-aided rapid prototyping of real-time embedded systems. Reducing uncertainty, ambiguity and inconsistency in design and development.

Will Bralick, Assistant Professor

Software engineering, formal models, automata/formal language theory. Hardware/software co-design.

Nelson Ludlow, Assistant Professor

Natural language processing, image understanding, cognitive science, informatics, and complexity theory.

Dennis Volpano, Assistant Professor

The Advanced Type Systems Project at NPS aims to develop new forms of static program analyses within the context of type systems.

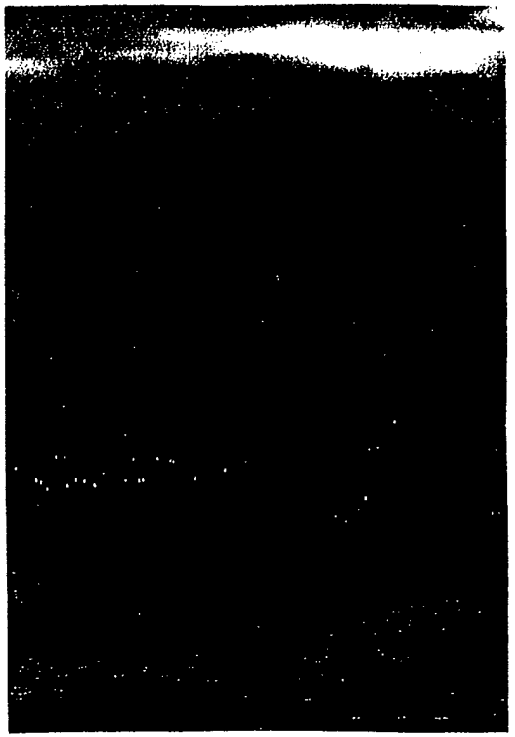
Mike Holden, Lecturer

Software methods, principles, and applications for military robots.

Naval Postgraduate School

SOFTWARE ENGINEERING

Department of Computer Science
Monterey, California 93943-5118



Prof. Luqi, Chair, Software Engineering

Prof. Lewis, Chair, Computer Science

Faculty: Valdis Berzins, Will Bralick, Mike Holden, Nelson Ludlow, Man-Tak Shing, and Dennis Volpano

se@cs.nps.navy.mil

(408) 656-2735

http://www.cs.nps.navy.mil/curricula/tracks/se

Software Engineering Curriculum

The DoD and DoN invest billions in the development of large-scale software systems. Reliability, flexibility, and delivery of these systems on time and under budget are prime concerns. Studies indicate the cost to support deployed DoD software comprises 60% - 80% of total cost.

The Software Engineering curriculum provides every student with a fundamental understanding of the theory and practice of software engineering -- *the use of sound engineering principles to economically develop software that is reliable, flexible, and works efficiently on real machines.*

Curriculum designed to provide knowledge of all aspects of software development and skills needed to efficiently and reliably plan and create large-scale software systems using the best available tools.

Core subjects include:

- specifications and requirements analysis,
- methods for
  - -- software development
  - -- software design
  - -- software evolution
- prototyping and CASE technology
- algorithms and data structures
- software engineering articulated in Ada

Descriptions of the courses offered within the Software Engineering Program are given below.

CS-2972 Object-Oriented Programming with Ada (3-2) This course is designed to teach students problem solving techniques and the object-oriented programming paradigm with Ada.

CS 3460 Software Methodology (3-1). Introduction to software engineering and the software life cycle. Methods for requirements definition, design and testing of software.

CS 4500 Software Engineering (3-1). In-depth coverage of the techniques for the specification, design, testing maintenance and management of large-scale software systems.

CS 4510 Computer-aided Prototyping Systems (3-1). Concept and application of computer-aided prototyping to the development and acquisition of DoD software systems. Prototyping software life cycle, models, methods, code generation, prototyping languages and tools.

CS 4520 Advanced Software Engineering (3-0). Methods for specifying, designing and verifying software systems are covered with emphasis on automatable techniques and their mathematical basis. Techniques are applied to construct and check Ada programs using a formal specification language.

CS 4530 Software R & D in DoD (3-0). State-of-the-art methods, techniques and standards aimed at improving the development and acquisition of DoD software systems focusing on large, real-time embedded computer systems. Automated tools for the specification, design and generation of Ada code, and DoD standards for software development and acquisition

CS 4540 Software Testing (3-1). This course covers the theory and practice of testing computer software with the intent of preventing, finding and eliminating errors in software.

CS 4560 Software Reuse (3-1). The concepts, methods, techniques and tools for supporting the evolution and maintenance of software systems. The use of formal specifications to support software evolution, design databases, configuration management, software change merging, and software re-engineering.

CS 4570 Evolution (3-1). Concepts, methods, techniques and tools for systematic reuse of software components and systems. Design and re-engineering for reuse, mechanisms for enhancing reuse, domain specific reuse and software architectures, requirements model reuse, specifications and designs, reuse tools, software library organization, and component search methods.

CS 4580 Embedded Real-time Systems (3-1). Theory and practice of embedded, real-time software systems. Establishing feasibility of real-time constraints via scheduling algorithms and techniques for simulating unavailable components via algebraic specifications.

Software Engineering Laboratory

The purpose of the laboratory is to provide a state-of-the-art educational environment for graphics-based software development automation. The laboratory is used to teach graduate students how to develop mission critical Ada software for embedded systems. Current work in the laboratory is on rapid prototyping, specification languages and computer-aided software system design, software verification and testing, software safety and computer-aided instruction. A research tool, called CAPS (Computer Aided Prototyping System) is used by students to construct software prototypes based on the requirements of the system as well as to automatically generate Ada code interconnecting reusable modules.

CAPS Rapid Prototyping Environment

* Design Entry Facility:
  - capture application requirements as augmented dataflow diagrams
* Execution Support System:
  - automatic generation of target control code from design
* Software Base:
  - automated support for software reuse
* Project Control System:
  - automated configuration management and software evolution support

Benefits

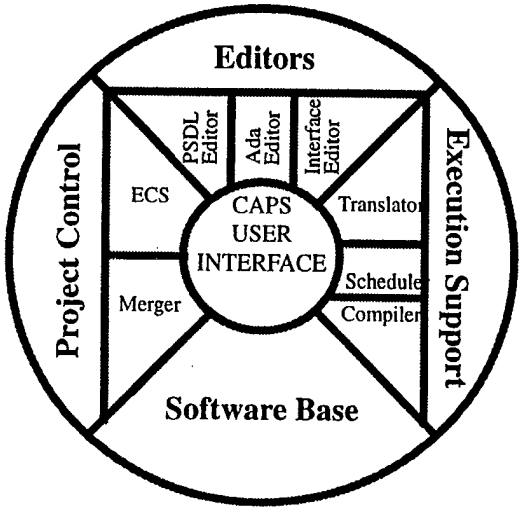
* Top level graphic requirements
* Risk assessment and reduction
* Requirements match customer needs
* Feasible real-time requirements
* Incremental delivery and integration
* No surprise project failures
* Faster software development
* Lower maintenance cost
* More flexible software

CAPS Release 1

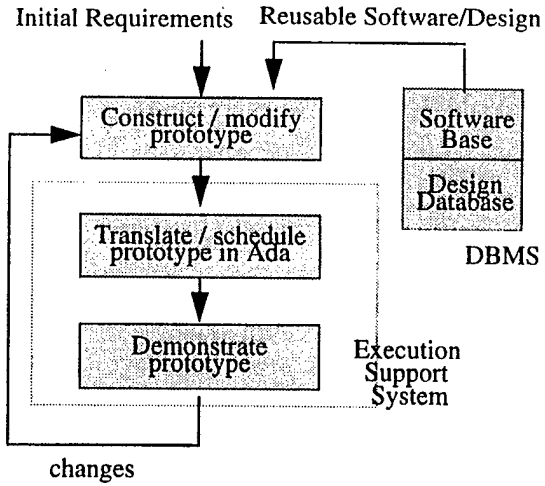
CAPS Release 1 is available through DISA's Defense Software Repository System (DSRS). Additional Information on CAPS can be obtained at:

caps@cs.nps.navy.mil,

http://wwwcaps.cs.nps.navy.mil



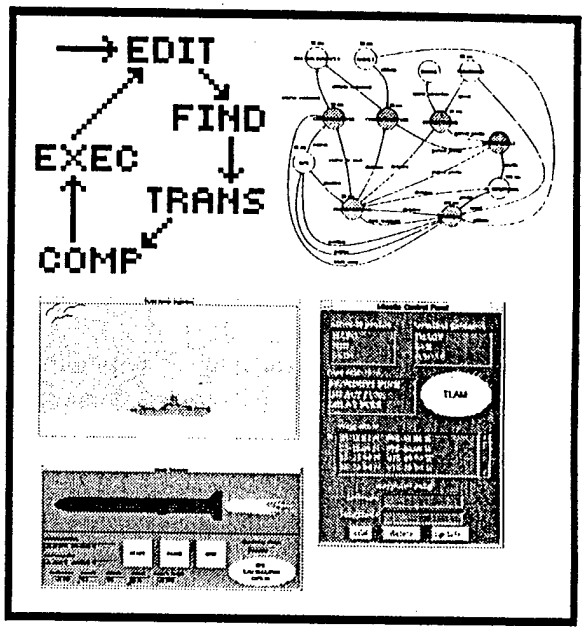
CAPS Rapid Prototyping Environment



CAPS Iterative Prototyping Process

NAVAL POSTGRADUATE SCHOOL

Software Technology for Large System Acquisition



Department of Computer Science
Monterey, California 93943-5118

POC: V. Berzins, Luqi, M. Shing
caps@cs.nps.navy.mil
(408) 656-2735

Software Technology for Large System Acquisition

Objectives

Assist DoD program managers and engineers to rapidly evaluate requirements for military real-time control software using executable prototypes and to test and integrate completed subsystems through evolutionary prototyping.

Issues

* Traditional software development methods conduct extensive testing near the end of the project in an attempt to ensure proper functioning of the system.
* The major weakness of this approach is that there is no way to recover from major faults discovered at the end of the project, when available funds have been nearly exhausted.
* Inability to accurately specify requirements for software systems results in poor productivity, schedule overruns, unmaintainable and unreliable software.
* Need functional prototypes to verify feasibility, formulate subcontracts early in the software development process and test delivered subsystems during system integration.
* Rapid prototyping supports the DoN acquisition policy, which states that "To promote effective interaction between the user and the developer, software prototyping methods shall be used in the design and construction of C2 information systems. Early delivery of software systems is emphasized through the use of prototyping methods."

Computer Aided Prototyping for Large System Acquisition

Courses

CS2972 Object-Oriented Programming with Ada
CS3300 Data Structures
CS3460 Software Methodology
CS4150 Programming Tools and Environments
CS4500 Software Engineering
CS4510 Computer-Aided Prototyping
CS4520 Advanced Software Engineering
CS4530 Software R&D in DoD
CS4540 Software Testing
CS4560 Software Evolution
CS4570 Software Reuse
CS4580 Design of Embedded Systems
CS4920 Computer-Aided Requirements Engineering
MN3301 System Acquisition and Program Management
MN3309 Acquisition of Embedded Weapon Systems Software
IS3171 Econ Evaluation of Info Systems II
IS4200 System Analysis and Design

Faculty

Valdis Berzins, Professor
Luqi, Professor
Ted Lewis, Professor
Man-Tak Shing, Assoc. Professor
William Bralick, Assis. Professor, MAJ, USAF
Nelson Ludlow, Assis. Professor, MAJ, USAF
Dennis Volpano, Assis. Professor
Michael Holden, Lecturer, CDR, USN
Ramesh Balasubramaniam, Assis. Professor
Barbara Pawlowski, Lecturer, Lt Col USAF

Computer Aided Prototyping Laboratory

The Computer Aided Prototyping Laboratory (CAPS Lab), a part of the Naval Postgraduate School's Computer Science Department, is the premier site for the study of computer aided prototyping technology for large system development and acquisition. It is one of the best software engineering labs in the country for distributed real-time software. The lab has been developed through support from many sponsors who are committed to its success, including the NSF, ONR, ARO, ARL, DISA, NRad, NSWC, NAVDEC, NISMIC, and SPAWAR. The facilities include a network of workstations and multi-processor servers, connected via a high performance fiber-optic network.

The laboratory contains a large collection of public domain reusable component libraries, including thousands lines of Ada code on missile navigation and C3I systems developed locally. The CAPS tools help users build requirements models for proposed systems rapidly, perform feasibility and risk assessment via simulation, reduce human error, reduce development cost, perform incremental integration, check software quality incrementally, and prevent surprise project failure. It has been used to support both teaching and research on computer-aided software engineering, in a program which has been ranked the best among all academic institutions and third overall in the nation by the Journal of Systems and Software in 94.

LIST OF REFERENCES

1. William J. Perry, Secretary of Defense, Department of Defense Memorandum, 2 June 1995.

2. Jerome S. Gabig, Jr., "The New DOD Clauses On Rights In Technical Data And Computer Software," http://venable.com/govern/nwsoftdv.htm, downloaded March 1996.

3. Barry Frew, classnotes from the Information Technology Management capstone course at the Naval Postgraduate School, Monterey, Winter, 1996.

4. Lee Gramelian, presentation to the Information Technology Management capstone course at the Naval Postgraduate School, Monterey, Winter, 1996.

5. Lloyd K. Mosemann, II, quoted in the Policies, Laws, Regulations Affecting Software Reuse in DoD, Student Manual, p. 1-26.

6. Naval Post Graduate School, Systems Engineering Group, Brief to Assistant Secretary of the Navy, March 1996.

7. Sprague, Ralph H., Jr., and Barbara McNurlin, Information Management Systems in Practice, Chapter 9, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

8. Robert L. Glass, "An Assessment of Systems and Software Engineering Scholars and Institutions," Journal Of Systems Software, No. 27, 1994

9. U.S. Department of Defense (DoD) "Ada Mandate",URL: http://ugrad-www.cs.colorado.edu/~csci3155/Ada/LoveLace/doduse.html, 24 August 1996.

10. Luqi and Valdis Berzins, "Rapidly Prototyping Real-Tme Systems," IEEE Software, September 1988.

11. Policies, Laws, Regulations Affecting Software Reuse in DoD, Student Manual, ver 1.0, prepared by DLA Office of Counsel Columbus Region for the Defense Information Systems Agency, October, 1994.

12. Reuse Executive Primer, working draft prepared by the Software Reuse Initiative Program Management Office, DISA, February 8, 1995.

13. Jeffrey L. Whitten, et. al., Systems Analysis and Design Methods, Chapter 5 and Chapter 12, Irwin, Boston, 1994.

14. Mathews, William, "DoD Initiative Targets Better Management of Software," Defense News, 16-22 October 1995.

15. Luqi, "Computer-Aided Prototyping for a Command and Control System Using CAPS," IEEE Software, January, 1992.

16. G.Whitbeck, "System Requirements Design for the ATACMS using CAPS", Thesis, Naval Postgraduate School, September 1996.

17. Risdon, Penny, "Understanding the Technology Transfer Process", New Zealand Science Review 49, 15-20, 20 Dec 1992.

18. Stevenson-Wydler Technology Transfer Act of 1980, Public Law 96-480, United States Congress, Washington D.C., 21 October 1980.

19. Scott Morgen, "Cooperative Research and Development Agreement", Thesis, Naval Postgraduate School, September 1993.

20. M. K. Knudson, "Incorporating Technological Change in Diffusion models", American Journal of Agricultural Economics 73, 724-733, October 1991.

21. E. H. Weiss, "How to Write Useable Documentation, Second Edition, The Oryx Press, 1991.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, CA 93943-5101	2
3.	Professor Luqi Computer Science Department Naval Postgraduate School Monterey, CA 93943-5101	9
4.	Professor Barry Frew Systems Management Department Naval Postgraduate School Monterey, CA 93943-5101	1
5.	LT Robert Cooke, Jr. 1224 Wivenhoe Court Virginia Beach, Virginia 23454	1