

AD-A226 984

2

IDA MEMORANDUM REPORT M-241

PROCEEDINGS OF THE THIRD IDA WORKSHOP ON
FORMAL SPECIFICATION AND VERIFICATION OF Ada*
14-16 May 1986

Editors:

W.T. Mayfield
John Chludzinski
John McHugh
S.R. Welke

August 1986

Prepared for
Ada Joint Program Office

DTIC
SELECTE
SEP 12 1990
S E D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited



INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

90 09 12 0 1

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Memorandum Reports is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Memorandum Report is published in order to make available the material it contains for the use and convenience of interested parties. The material has not necessarily been completely evaluated and analyzed, nor subjected to formal IDA review.

© 1990 Institute for Defense Analyses

The Government of the United States is granted an unlimited license to reproduce this document.

Approved for public release; unlimited distribution.

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 instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE August 1986		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Proceedings of the Third IDA Workshop on Formal Specification and Verification of Ada, 14-16 May 1986			5. FUNDING NUMBERS MDA 903 84 C 0031 T-D5-304	
6. AUTHOR(S) W.T. Mayfield, John Chludzinski, John McHugh, S.R. Welke				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard Street Alexandria, VA 22311-1772			8. PERFORMING ORGANIZATION REPORT NUMBER IDA Memorandum Report M-241	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office (AJPO) The Pentagon, Room 3D159 Washington, D.C. 20301			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution.			12b. DISTRIBUTION CODE 2A	
13. ABSTRACT (Maximum 200 words) The Third IDA Workshop on Formal Specification and Verification of Ada was conducted at the Research Triangle Institute, Research Triangle Park, North Carolina on May 14-15, 1986. The theme of the workshop was "Researching Verifiable Ada Systems by 1990" and addressed the following issues: (1) advances in verification technology, (2) adaptation of current technology in Ada verification systems and methods, (3) broadening the base of support for work in Ada verification, and (4) encouraging the participation by larger segments of both the Ada and the verification communities. A detailed exposition of the Ada formal definition being developed by the European Economic Community. This exposition took the form of a series of tutorial presentations, enclosed in this document, on various aspects of the dynamic and static semantics of the definition and its underlying formalisms. Dr. Harlan Mills from IBM's Federal Systems Division was the keynote speaker.				
14. SUBJECT TERMS Ada Programming Language; Verification; Specification; Secure Systems; Semantics; Concurrency; Computer Security; Software; Support Library.			15. NUMBER OF PAGES 620	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	

IDA MEMORANDUM REPORT M-241

PROCEEDINGS OF THE THIRD IDA WORKSHOP ON
FORMAL SPECIFICATION AND VERIFICATION OF Ada
14-16 May 1986

Editors:
W.T. Mayfield
John Chludzinski
John McHugh
S.R. Welke

August 1986

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-D5-304



ACKNOWLEDGMENTS

The Institute for Defense Analyses would like to thank all those who attended the Third IDA Workshop on Formal Specification and Verification of Ada, and in particular, those who gave time to prepare presentations of their work. We would like to express special thanks to Dr. Wooten for the use of Research Triangle Institute facilities and Ms. Sandy Waters for her invaluable assistance as workshop administrator.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	i
WORKSHOP SUMMARY	1
Overview	1
The Keynote Address	1
Contributed Papers	2
The Draft of the European Formal Definition of Ada	6
Concluding Remarks	7
Human Verification in Ada (Paper)	9
Human and Machine Ada Verification (Presentation)	15
Harlan D. Mills, IBM Corporation	
Applying Semantic Description Techniques to the CAIS (Paper)	21
Applying Semantic Description Techniques to the CAIS (Presentation)	51
Timothy E. Lindquist, Arizona State University	
Roy S. Freedman, Polytechnic University	
Bernard Abrams, Grumman Aircraft Systems	
Larry Yelowitz, Ford Aerospace	
MAVEN: The Modular Ada Validation Environment (Paper)	71
MAVEN: The Modular Ada Verification Environment (Presentation)	89
Norman H. Cohen, Softech, Inc.	
Software Hazard Analysis and Safety Verification Using Fault Trees (Paper)	127
Verification of Ada Programs for Safety (Presentation)	145
Nancy G. Leveson, University of California, Irvine	
A Proof Rule for Ada (Paper)	173
A Proof Rule for Ada (Presentation)	187
Ryan Stansifer, Purdue University	
Revisiting Axiomatic Exception Propagation (Paper)	209
Revisiting Axiomatic Exception Propagation (Presentation)	223
Timothy E. Lindquist, Arizona State University	
Program Development by Specification and Transformation (PROSPECTRA) (Paper)	239
European Strategic Programme for Research in Information Technology	
The PROSPECTRA Project (With an Emphasis on Verification) (Paper)	279
Andrew D. McGettrick, University of Strathclyde	

Program Development by Specification and Transformation (Presentation)	293
B. Krieg-Brueckner, Bremen	
H. Ganzinger, Dortmund	
M. Broy, Passau	
R. Wilhelm, Saarlandes	
A. McGettrick, University of Strathclyde	
I. Campbell, SYSEA LOGICIEL	
G. Winterstein, SYSTEAM	
L. Treff, SYSTEAM	
 On the Use of Semantic Specification for the Verification and Validation of Real Time Software (Paper)	313
On the Use of Semantic Specification for the Verification and Validation of Real Time Software (Presentation)	343
Patrick de Condeli, CR2A et Aerospatiale/Space Division	
 An Empirical Study of Testing Concurrent Ada Programs (Paper)	371
An Empirical Study of Testing Concurrent Ada Programs (Presentation)	401
Huo-Chung Tai, North Carolina State University	
Richard H. Carver, North Carolina State University	
Evelyn E. Obaid, San Jose State University	
 Logical Foundations and Formal Verification (Paper)	417
R.B. Jones, ICL Defense Systems (UK)	
 Trusting Compilers -- A Pragmatic View (Paper)	441
Trusting Compilers -- A Pragmatic View (Presentation)	447
Scott Hansohn, Honeywell Secure Computing Technology Center	
 An Introduction to the Draft Formal Definition of Ada (Paper)	459
Egidio Astesiano, Universita' di Genova	
Jan Storbak Pedersen, Dansk Datamatik Center	
The Draft Formal Definition of Ada: An Introduction (Presentation)	471
Jan Storbak Pedersen, Dansk Datamatik Center	
The Draft Formal Definition of Ada: Trial Definition of the Static Semantics (Presentation)	481
Jan Storbak Pedersen, Dansk Datamatik Center	
The SMoLCS Methodology and its Application to the Ada FD -- Dynamic Semantics (Presentation)	511
Egidio Astesiano, Universita' di Genova	
SMoLCS Methodology of Specification (Presentation)	539
Egidio Astesiano, Universita' di Genova	
Gianna Reggio, Universita' di Genova	
Martin Wirsing, University of Passau, W. Germany	
C. Bendix Nielsen, DDC, Denmark	
J. Storbak Pedersen, DDC, Denmark	
A. Giovini, DDC, Denmark	
F. Mazzanti, DDC, Denmark	
E. Zucca, DDC, Denmark	
A. Fantechi, DDC, Denmark	
P. Inverardi, DDC, Denmark	

The Draft Formal Definition of Ada: Other Dynamic Semantics Aspects (Presentation)	565
Jan Storbak Pedersen, Dansk Datamatik Center	
Formal Definition, Specification, and Verification (Presentation)	593
Egidio Astesdiano, Universita' di Genova	
APPENDIX A LIST OF ATTENDEES.....	A-1

Workshop Summary

Overview

The Third Institute for Defense Analyses (IDA) Workshop on Ada Verification was conducted May 14-16, 1986, at the Research Triangle Institute, in the Research Triangle Park, North Carolina. Unlike the preceding workshops, the third was structured more as a symposium, with a keynote speech and contributed papers occupying the first day and a half of the meeting. The remaining two half days were given over to a detailed exposition of the Ada formal definition being developed by the European Economic Community. This exposition took the form of a series of tutorial presentations on various aspects of the dynamic and static semantics of the definition and its underlying formalisms.

The theme of the workshop was "Reaching Verifiable Ada Systems by 1990". The objectives were:

- 1) to address advances in verification technology,
- 2) to continue the adaptation of current technology in Ada verification systems and methods,
- 3) to identify the road map for future basic/applied research in Ada verification technology,
- 4) to broaden the base of support for work in Ada verification, and
- 5) to encourage participation by larger segments of both the Ada and the verification communities.

The remainder of this introduction will summarize and comment on the papers presented in each session.

The Keynote Address

The workshop was extremely fortunate to have as its keynote speaker, Dr. Harlan Mills from IBM's Federal Systems Division (FSD). Dr. Mills has long been advocating the introduction of mathematical rigor into the software engineering process and has been leading a quiet revolution to introduce, within FSD, a programming methodology based on formal verification at the unit level and statistical testing at the system level. Known as the "Cleanroom" methodology, the technique is being adopted throughout FSD with training being done in "top down" fashion (i.e., management first). In fact, much of the impetus and support for the development of the "Cleanroom" methodology is derived from management's conviction that "verification methodology is an especially potent and force for effective intellectual and management control of software development."

IBM was fortunate, in that the top managers at FSD had a good background in mathematics and were both receptive to the new methodology and able to understand and apply it. By introducing the cleanroom techniques downward through the management structure, FSD has ensured that support for the techniques exists before they are applied at the project level. The common, but unfortunate reaction of a first line manager to a technique proposed by an employee just back from a training course -- "That's interesting but we don't do it that way here." is thus avoided.

Dr. Mills' talk provided some background and anecdotal experience on the method and its application. Several interesting points arose in the talk and discussion which followed:

- 1) The volume of specification exceeds the volume of code by a large factor, often as much as 5 or 6 lines of specification per line of code.
- 2) A similar ratio is observed for the verification arguments.
- 3) The functional approach used avoids problems with pre- and post-conditions, such as: $P \{200 \text{ pages of code}\} Q$.
- 4) It also avoids problems with excess logical notation for unchanged state components.

Following the presentation, Dr. Mills was asked about the problem of formal specifications being as complicated as source code. From experience, Dr. Mills believed the biggest gain is that formal specifications remove iterations and sequences, both of which are more difficult than people think. He suggested reading Structured Programming: Theory and Practice by Linger, Mills, and Witt (Addison Wesley, 1979) to obtain a more complete presentation of this issue.

Dr. Mills concluded his talk with regrets that he was unable to stay for the remainder of the workshop, and he offered to use his influence wherever possible to assist the community represented by the workshop in obtaining research funding.

Contributed Papers

Dr. Timothy Lindquist

Arizona State University

"Applying Semantic Description Techniques to the CAIS"

Dr. Lindquist introduced the CAIS and presented a paper discussing the need for and the problems with a formal semantic description of the CAIS. Formalism is necessary because the general, if not universal, rule is that natural language descriptions tend to be verbose, ambiguous, and incomplete. The paper covers the three approaches of the authors: operational semantics (abstract machine), axiomatic semantics (algebraic), and denotational semantics. Not surprisingly, the paper concluded that no single formal method will uniformly satisfy all the needs for a formal definition of the CAIS, and thus, a combination of methods is required.

Dr. Norman Cohen
SofTech, Inc.
"MAVEN"

Dr. Cohen presented a paper on the Modular Ada Validation Environment (MAVEN), an environment to provide module-by-module validation for Ada libraries of reusable code. The paper outlined some of the features supported by the environment: a) validation together with verification (formal correctness) for mission critical software, b) integration of multiple validation approaches, c) assistance for writing specifications, and d) libraries of validated reusable Ada software.

Dr. Cohen believed MAVEN offers an excellent framework for the specification, design, and implementation of Ada software. MAVEN provides a practical role for verification and makes a continuum of validation methods available. Dr. Cohen hopes that exposure to formal methods will encourage industry and academia to learn about these techniques. However, he warned against a premature implementation of this nascent idea.

After Dr. Cohen's presentation, there were challenges to his module-by-module approach to validation. One comment was that verifying modules before putting them into a library is impractical because of the need to modify modules before reuse and to reverify every modified module. Another comment was that modules cannot be clearly and completely specified. It was mentioned that Kowalski once said he never got the specifications right the first time; Paines later said he never got the specifications "right".

Dr. Nancy Leveson
University of California at Irvine
"Verification of Ada Programs for Safety"

In a well received talk, Dr. Leveson discussed the possibilities of using formal verification techniques to examine safety properties of systems. She pointed to the need for safety concerns in a continually growing number of computer applications (e.g., monitor and/or control of physical processes or mechanical devices). Dr. Leveson discussed several real examples of supposedly "correct" software. She explained how all too often, programs, which are verified as being correct with respect to some set of specifications, exhibit unacceptable behavior with respect to the actual requirements. Consequently, several alternatives were considered:

- 1) don't build the systems, which may actually be a good idea at times, but most often is unrealistic,
- 2) take more of a "systems" view, or
- 3) approach system reliability with a less "absolute" view.

Further, it was pointed out that making software "safe" sometimes makes it unreliable, and that every system has some measure of inherent risk associated with its use.

Dr. Leveson also noted that many of the issues of "software hazard analysis" are not well understood and need further research. Most of the work to date has centered around fault tree analysis, started in the 1960's at Bell Laboratories. Dr. Leveson pointed

to one of the weaknesses of fault tree analysis: the events symbolized by the nodes are assumed to be mutually independent.

At the end of the presentation, while summarizing her thoughts, Dr. Leveson drew a diagram to visualize her concerns. The diagram indicated that the state space of a process may be partitioned into *correct states* and *incorrect states*, with respect to a set of specifications. Within both the correct and incorrect state spaces, there may exist *unsafe states*. What is needed are more precise ways of defining these unsafe states and better mechanisms for avoiding them during execution of the program.

Dr. Ryan Stansifer
Purdue University
"A Proof Rule for Ada"

Dr. Stansifer presented a paper giving a new proof rule for loops with explicit **exit** statements. The **exit** statements may cause control of the program to exit the current loop or any of the outer nested loops. This work gives attention to non-sequential control flows and examines a single construct closely akin to the Ada loop mechanism.

In addition to the work done on the proof rule, Dr. Stansifer presented his work in the area of generating verification conditions with attribute grammars. He expressed the opinion that, by 1990, it will be feasible to have proof rules for all of Ada.

After his presentation, Dr. Stansifer was asked whether attribute grammars are better than current methods (e.g., GYPY) for generating verification conditions, or whether they are merely a different means. He responded that attribute grammars are a different means which are not necessarily better.

Dr. Timothy Linquist
Arizona State University
"Reviewing Axiomatic Exception Propagation"

Dr. Lindquist presented a paper discussing an axiomatic approach to exception specifications that avoids the need for augmented specifications. The main motive for this work was to provide a better mechanism for representing the semantics of exception propagation. This work gave further attention to non-sequential control flows, as did Dr. Stansifer's work on proof rules for an Ada-like loop construct, and examined the Ada exception mechanism.

There was a question as to whether or not this method was an attempt to remove the abnormal case from the specifications in order to preserve the appearance of normalcy. Dr. Lindquist responded that it was not, and that it depended on the context of the dynamic calling structure. It was also pointed out that this method is presently restricted to a taskless subset of Ada.

Dr. A. D. McGettrick
University of Strathclyde, Scotland, U.K.
"PROSPECTRA"

Dr. McGettrick summarized the PROSPECTRA (PROgram development by SPECification and TRAnsformation) project now in development under the auspices of the European ESPRIT program. The PROSPECTRA system will provide a rigorous method and environment for the development of "correct" software. In particular, the PROSPECTRA project plans to use the Ada/ANNA program transformation technique for the development of validated software. When implemented, the environment will rely on a knowledge base of proven transformation scripts to be used interactively by the programmer.

Dr. Peter de Bondeli

Conception et Realisation d'Applications Automatisees (CR2A) and Aerospatiele
"Use of Semantic Specifications for the Validation and Verification of Real-Time Software"

Dr. de Bondeli presented a paper outlining a Predicate-Transition Petri Net (Pr-T Net) approach to the analysis of concurrency, in particular, the Ada model of concurrency. The purpose of this work is to analyze the behavior of software designed for real-time applications and to show that verification of real multi-tasking programs written in Ada is feasible, though difficult. This system is applicable to the analysis of concurrency in general, but most applications are more likely to be real-time than simply concurrent.

Some members of the workshop asked if there was an advantage to reasoning with Petri Nets over reasoning directly with Ada primitives. Dr. de Bondeli responded that by dealing with Ada primitives, the analysis is bound to a particular implementation; it does not have the generality of the Petri Net analysis. Additionally, the Ada structures required to implement a relatively simple notion are quite complex.

Another advantage of using Petri Nets is the redundancy resulting from first designing the communication requirements with Petri Nets followed by an implementation in Ada. It is then possible to verify the Petri Net design with Ada, and subsequently verify Ada design with Petri Nets. For people who are familiar with Ada, Ada may seem more natural; but, the goal is an exact formalism for everyone, not just the small community of Ada users.

Editor's note: Often in our quest for improvements in formal methods, we forget that most programmers and project managers are not acquainted with the more traditional denotational or axiomatic approaches. The power of this Pr-T Net approach is really something that should be highlighted. To me this was best exemplified by a statement from a young attendee, new to the area of formal verification: "now this is something I am able to understand!" This statement should remind us all that understandability must be the watchword of the verification community!

Dr. K. C. Tai

North Carolina State University
"An Empirical Study of Testing Concurrent Ada Programs"

Dr. Tai presented a paper concerned with a testing method for tasking in Ada. He pointed out that testing concurrent programs written in Ada is particularly difficult due to the non-reproducible/non-deterministic behavior of such programs. In his testing method, Dr. Tai uses the *deterministic execution approach* to reproduce rendezvous sequences and to determine whether the rendezvous sequences are feasible. Presently, the work considers only single-processor systems.

Mr. Scott Hansohn
Honeywell Secure Computing Technology Center
"Trusting Compilers -- A Pragmatic View"

Mr. Hansohn presented a paper discussing the problem of compiler trustworthiness. The main thrust behind Mr. Hansohn presentation was that by relying upon tools for the development of software, the trustworthiness of the software is subject to many risks. These risks range from malicious or incompetent tool writers to inherently incomplete descriptions or specifications. Mr. Hansohn also pointed out that it is impossible to guarantee the "correctness" of every level of the system, especially the compiler level. Some trade-offs must be made between programmer productivity and software trustworthiness.

Dr. Roger Jones
ICL Defence Systems
"Logical Foundations and Formal Verification"

Dr. Jones presented a paper discussing an approach to formal semantics based on pure combinatorial logic. In the discussion which followed the presentation, it appeared that the work is far from finished and its effectiveness is still open to question.

The Draft of the European Formal Definition of Ada

Dr. Egidio Astesiano
University of Genoa

Mr. Jan Pedersen
Dansk Datamatik Corporation
"An Introduction to the Draft Formal Definition of Ada"

Mr. Pedersen described the current European approach to a formal definition of Ada, though this method is not restricted to Ada. The definition is a hybrid of denotational and algebraic semantics and is supplemented by an English language narrative as a further explanation. The definition is keyed and cross-referenced to the Ada Language Reference Manual (LRM), ANSI/MIL-STD 1815A, to permit direct comparison with the official, but informal, definition. Finally, the definition is based on the SMoLCS (Structured Monitored Linear Concurrent Systems) methodology, an approach to the specification of concurrent systems.

Mr. Pedersen focused on the static semantics portion of the formal definition. Dr. Astesiano focused on the dynamic semantic portion of the formal definition. Additionally, Dr. Astesiano discussed the relationship between the European formal definition of Ada and specifications used in the formal verification of program text. There was a tremendous amount of material to be presented regarding the formal definition, and the best that could

be expected was a quick, detailed run through. In retrospect, a tutorial by one of the external reviewers of the work would have been a useful preliminary.

At the conclusion of the presentation, Dr. Platek thought it might be helpful to give some historical remarks concerning the European effort to construct a formal semantics for Ada. About four years ago, people familiar with the real problems in developing a formal semantics began to study the NYU and CEC definitions instead of the Ada LRM; hence, the current European effort to formulate a formal definition. Dr. Platek and Dr. Luckham are on a review committee accessing and evaluating the European formal definition of Ada. The committee is currently concerned with such questions as:

- 1) Does the formal definition create a language different from the one defined by the Ada LRM?
- 2) What are the potential uses of the formal definition?
- 3) How can the formal definition be verified?

The full definition is due January 1, 1987, and it will be reviewed until June 1987, at which time it may be adopted as an ISO standard.

Concluding Remarks

A common theme, not established at the outset of the workshop, did emerge. This theme was "Multi-Paradigms", indicating that there is room for, and indeed a need for, a multitude of paradigms for practical verification of Ada programs. A substantial number of the papers presented at the workshop addressed various verification techniques targeting different mechanisms within the language. Papers were presented on the following: specifying concurrent programs, based on Predicate-Transition networks; verifying loop control structures, based on Hoare triples; and verifying exception handlers, based on axiomatics. An outline of a method for generating verification conditions from attribute grammars was also presented. Although these methods are diverse, their applied purpose was singular: the verification of Ada program text.

The problem with all of these varied approaches is that in each case, the researcher had to "go back to the well" of the LRM to justify the semantics used. To achieve consistency across approaches, it would be a reasonable goal for the verification community to urge the adoption of a formal definition as final arbiter for questions of language meaning. This, of course, implies the establishment and support of a maintenance capability for the formal definition. This capability should be integrated with, or at least commensurate with, the maintenance capability of the Ada language itself. Even if such a definition is not suitable for direct use in automated verification systems, it would provide a better basis for resolving issues concerning the soundness of interpretations of language semantics and verification implementations.

What appears to be the next needed step, therefore, is to leverage the work done on the EEC Formal Definition of Ada to the verification tool and environment work that has been done previously here in the U.S.A. Specific projects should be proposed, along with project milestones and funding profiles, which can be coordinated as an overall program. While there is yet no money readily available, it is important that the definition of what might be included in such a program should be developed so that appropriate funding can be sought.

To date, the security community has been the major sponsor of research and development of the verification technology. However, it should not be expected that the security community will continue to be the primary or sole supporter of all future research in this area. Sponsorship within the Ada community is diverse and many sponsors, other than the security community, have both an interest in and a need for Ada verification technology. As it increases in complexity, mission-critical software is a particular area where such technology could provide significant benefits. To assist in obtaining this funding, papers need to be written targeting managers and touting the advantages and potentials of the formal definition work and the technologies that should develop around it to support the needs of these sponsor's programs. Success stories should be provided to amplify the advantages and potentials of developing verified software. Unless other members of the Ada community are convinced to support these activities, there is a real danger that the lack of funding for new or sustained research efforts will continue.

THIRD IDA WORKSHOP ON ADA[®]
SPECIFICATION AND VERIFICATION
Research Triangle Park, NC - May 14-16, 1986

Human Verification in Ada

Harlan D. Mills

IBM[®] Corporation

Objectives

It is an honor to be with you today. I regard the verification community as a critical national resource in software engineering. For too many people, even in the universities, software engineering is more a buzzword than a bonafide engineering discipline. The rigor of verification is a key discriminator between buzzword and engineering discipline in software. So it is an opportunity, indeed, for me to speak with you.

I have two objectives today. First, I would like to encourage you to transfer your verification knowledge and technology to people as well as to machines. People are more fallible than machines in the verification process, but they are more flexible, too. I would like to convince you that people are worth the effort. Second, I would like to see human verification supported in Ada environments, in particular by the inclusion of verification language and processing facilities to complement and complete such facilities for programs and specifications.

In support of these objectives, I would like to discuss our experience in IBM with the so-called Cleanroom method of software development, in which human verification is used to replace program debugging before release to independent testing. We are finding that human verification is surprisingly synergistic with statistical testing -- that mathematical fallibility is very different than debugging fallibility, and that errors of mathematical fallibility are much easier to discover in system testing than are errors of debugging fallibility.

I would also like to describe the method of human verification used in IBM, called functional verification, that is somewhat different than axiomatic verification or predicate transformers usually taught in universities. It is based on a denotational semantics and the reduction of verification to ordinary mathematical reasoning about sets and functions as directly as possible. While harder to teach than axiomatic verification, we find that functional verification scales up for reasoning about million line systems in top level design as well as about hundred line programs at the bottom.

[®]Ada is a registered trademark of the U.S. Government
[®]IBM is a registered trademark of the International Business Machines Corporation

. Finally, I would like to encourage you to expand your efforts in automatic verification. We are in need of breakthroughs there, possibly in the very fundamentals of how we frame the problems and processes.

Who Drives Verification

In the Ada paper, "The Status Of Verification Technology For The Ada Language" by Nyberg, Hook, and Kramer, it is stated that "...verification technology has been driven more by the security community than any other community" [10, p.3]. In contrast, in IBM, the human verification methodology has been driven by the management community [8]. This management concern has led to the IBM Software Engineering Institute (SEI), an internal teaching institution with a faculty of some 50 worldwide, whose curriculum is centered on the systematic design and functional verification of both programs and modules [6]. The driving force in setting up the IBM SEI was a management conviction that verification methodology is an especially potent force for effective intellectual and management control of software development. The SEI courses are pass/fail, to be required eventually of all software personnel in IBM.

Verification education is a potent management tool because it literally induces behavior modification in industrial programmers in design and communication activities, whether or not formally practiced program by program. For example, mathematical verification requires formal specifications to improve communications, repeatable reasoning about why programs meet their specifications, and places a premium on clear, simple designs rather than clever, baroque ones. In fact, programs that are difficult to verify, even informally, are automatic candidates for redesign.

Cleanroom Software Development

While it may sound revolutionary at first glance, the Cleanroom method is an evolutionary step in IBM software development. Very briefly, program testing and debugging is prohibited before software is released for independent system testing. Then the system is tested with user representative, statistically generated, inputs. It is evolutionary in eliminating debugging because over the past twenty years more and more program design has been developed in design languages that is verified rather than executed, so that the relative effort in debugging in advanced IBM groups, compared to verifying, is now quite small, even in non-Cleanroom development. It is evolutionary in statistical testing because user representative testing is correspondingly a greater and greater fraction of the total testing effort. And, as already noted, we have found a surprising synergism between human verification and statistical testing. People are fallible with human verification, but the errors they leave behind for system testing are much easier to find and fix than those left behind from debugging.

The feasibility of combining human verification with statistical testing makes it possible to define a new software development process with bonafide statistical quality control [3]. For that purpose, we define a development life cycle of several incremental releases to a so-called structured specification of function and (statistical) usage. A structured specification is a formal specification (a relation, i.e., set of ordered pairs) with a decomposition into a nested set of subsets, to provide a subspecification for each release that includes that of all previous releases. That is, a structured specification defines not only the final software, but also a release plan for its incremental implementation and statistical testing. As each release becomes available, statistical testing provides statistical estimates of its reliability (e.g., in MTTF), and software process analysis and feedback can be used to meet prescribed reliability objectives (e.g., by increased verification inspections, more specification formality, etc.) for subsequent releases. As errors are found and fixed during system testing, the growth in reliability of the maturing system can be estimated as well, so that at final release, a certified reliability estimate of the system tested software can be provided.

Our Cleanroom experience to date includes three projects, an IBM language product (35 KLOC), an Air Force contract helicopter flight program (35 KLOC), and a NASA contract space transport planning system (45 KLOC). The major finding in these projects is that verification and inspections can replace debugging, even though fallible. That is, even informal human verification can produce software sufficiently robust to go to system test without debugging. Typical increments are 5-15K lines of software; with experience and confidence such increments can be expected to increase in size significantly. All three projects showed productivity equal to or better than expected for ordinary software development. That is, human verification takes no more time than debugging (although it takes place earlier in the cycle).

The verification process in these projects is very informal, with almost no automatic support beyond word processing, used mostly for decomposing relatively formal specifications during stepwise refinement. Yet, significant systems have been brought up, tested, and delivered on the strength of human verification alone. So just imagine how automatic verification support could amplify these human abilities!

Functional Verification of Software

Human verification in IBM is based on a denotational semantics for structured programs based on set theory (rather than lattice theory). This denotational semantics defines an algebra of functions (meanings) of program parts, with composition and union operators (for sequence and selection).

Iteration is defined by recursion, so composition and (disjoint) union are the only operators needed in the algebra of functions; of course, termination questions arise in iteration and are treated explicitly.

The paradigm for functional verification is somewhat different than for axiomatic verification or predicate transforms. It is to compute the meaning of a program or part (i.e., evaluate an expression in the algebra of functions), using the meanings of its parts, then to compare that meaning with the meaning of the specification. In the case of iteration, this paradigm can be converted into a termination question and the verification that a certain recursive function equation is satisfied by the specification. In particular, no loop invariant is needed for this verification. (See [6, p280] for the relation between iteration specifications and loop invariants).

The time to verify software is during stepwise refinement of a design. Every sequence, selection, and iteration verification requires specifications for their constituent parts, so stepwise refinement involves a lot of specification decompositions into subspecifications, altogether much more specification writing than program writing.

The motivation for functional verification, and earliest possible reduction of verification reasoning to sets, functions, and relations, is the problem of scaling up. A set, function, or relation can be described in three lines of ordinary mathematics notation or three hundred lines of English text. There is more human fallibility in three hundred lines of English than three lines of mathematical notation, but the verification paradigm is the same. By introducing verification in terms of sets, functions and relations of abstract states, with neither programming nor mathematical variables, we establish a basis for reasoning that scales up. Large programs have many variables, but only one meaning function.

Eventually these abstract states are represented in terms of programming variables, but those are convenient details of design rather than necessities of verification. So verification reasoning can (and does) take place at high levels of design, in connecting specifications with subspecifications before programming variables are invented.

For human verification we find pre- and post-conditions hard to scale up, for two reasons. First, two parts of a single specification become separated by considerable text as the design expands. It seems more convenient to give the specification in one place. Second, the post-conditions require state (or state part) replication to describe correspondences between initial and final states. These are not theoretical difficulties, or even pedagogical difficulties for small programs, of course.

Ada Suggestions

I have a direct suggestion for Ada environments -- include verification language and processing facilities to implement and complete such facilities for programs and specification. As de Bruijn [1], Martin-Lof [7] and Constable [2] have shown, we can represent natural deduction in a modern programming/specification language by adding propositions as types whose members are proofs of their propositions. Today's interactive verification facilities provide valuable experience in the human factors and use of such facilities. My suggestion is to bring verification writing into Ada environments as a full partner with program and specification writing.

Although possibly surprising to rank and file programmers, I believe the formal source in an Ada environment will see program text as a distinct minority, with at least three to five times as much specification text as program text, and with at least three to five times as much verification text as specification text. The stepwise refinement process generates much more intermediate specification text than initial specification text. But there is another reason why specification text should be more voluminous than program text. By its very use, program text can take advantage of iterations and sequential process, and thereby can be miniaturized in ways not available for specification -- people do not have to understand program text in the sense that they understand specification text, namely to provide an independent opinion of its relevance in its problem domain. They only need to understand that program text be correct with respect to the specification. So, except for toy problems, or extremely well understood problem domains, where abbreviations are easily recognized (e.g., mathematics), specification text will be longer than program text.

While program and specification text have identical meaning domains, verification text finds its meaning in whether its proposition types are empty or not. The main proposition types of interest are those that claim that a program is correct for a specification. But to exhibit a member of such a type constructively requires the equivalent of a constructive mathematical proof of the proposition. We know from experience that proofs of correctness are larger than their programs and specifications, even quite informal proofs. I would expect verification text to contain human proof designs for human inspection and judgement, rather than full proofs. Even so, human developed full proofs could be eventually proof checked, e.g. as in NUPRL [2], or Automath [1], in ADA environments.

I also believe new breakthroughs in automatic verification theory are possible. Experimental systems such as Gypsy [5] and AFFIRM [4] have provided valuable experience in the automation of verification based on the first order predicate calculus with axiomatic verification. It would be interesting to discover a complexity theory for verification that could explain this

experience, and shed light on performance issues across a broader set of alternative bases for verification. For example, de Bruijn in Automath begins with a typed lambda calculus, in which natural numbers and predicate calculus must be explicitly axiomatized, if required. While this seems a primitive base with extremely little built-in logical machinery, Automath has been used to proof check an entire mathematics text (Landau's Grundlagen der Analysis) in a relatively small, slow computer. There seems to be something to understand in that achievement. Constable, in NUPRL [2], begins on a base intermediate between the typed lambda calculus and the predicate calculus, with built-in natural numbers and lists of natural numbers, so performance in NUPRL should give an additional data point for a complexity theory of verification.

References

1. de Bruijn, N. G. A survey of the project Automath, in Essays in Combinatoric Logic, Lambda Calculus, and Formalism, J. P. Seldin and J. R. Hindley, eds. Academic Press (1980) pp 589-606.
2. Constable, R. L. et al. Implementing Mathematics with the NUPRL Proof Development System, Prentice-Hall (1986).
3. Currit, A., M. Dyer, and H. D. Mills. Certifying the Reliability of Software, IEEE Transactions on Software Engineering, SE-12, 1, (January 1986) pp 3-11.
4. Gerhart, S. L. Fundamental Concepts of Program Verification. AFFIRM Memo-15-SLG, USC ISI (1980).
5. Good, D. I., et al. Report on the Language Gypsy, Version 2.0 TR ICSCA-CMP-10, Inst. for Computing Science, U. of Texas, Austin (1978).
6. Linger, R. C., H. D. Mills and B. I. Witt. Structured Programming: Theory and Practice, Addison-Wesley (1979).
7. Martin-Lof, P. Constructive Mathematics and Computer Programming. In Sixth International Congress for Logic, Methodology, and Philosophy of Science, North Holland (1982) pp 153-175.
8. Mills, H. D. et al. The Management of Software Engineering, IBM Systems Journal, 15, 4 (1980).
9. Mills, H. D. and R. C. Linger. Data Structured Programming, IEEE Transactions on Software Engineering, SE-12, 2 (February 1986).
10. Nyberg, K. A., A. A. Hook and J. F. Kramer. The Status of Verification Technology For The Ada Language, P-1859 IDA (1985).

3RD IDA WORKSHOP ON ADA
SPECIFICATION AND VERIFICATION
RESEARCH TRIANGLE PARK, NC
MAY 14-16, 1986

HUMAN AND MACHINE ADA VERIFICATION

HARLAN D. MILLS
IBM CORPORATION
BETHESDA, MD

AGENDA

OBJECTIVES

TRANSFER VERIFICATION TECHNOLOGY TO PEOPLE
REALIZE VERIFICATION BENEFITS IN ADA ENVIRONMENTS

CLEANROOM EXPERIENCE

PROGRAMMING WITHOUT DEBUGGING SURPRISINGLY EASY
MATHEMATICAL FALLIBILITIES AND DEBUGGING FALLIBILITIES

FUNCTIONAL VERIFICATION

SIMPLIFIED DENOTATIONAL SEMANTICS
REDUCTION TO ORDINARY MATHEMATICAL PROCESS

ADA SUGGESTIONS

ADA VERIFICATION LANGUAGE
BRING IN STATISTICAL TESTING
EXPAND AUTOMATIC VERIFICATION EFFORTS

NYBERG, HOOK, KRAMER

IDA P-1879

"...VERIFICATION TECHNOLOGY HAS BEEN DRIVEN MORE
BY THE SECURITY COMMUNITY THAN ANY OTHER COMMUNITY."

IN IBM

VERIFICATION METHODOLOGY DRIVEN BY THE MANAGEMENT
COMMUNITY.

IBM SOFTWARE ENGINEERING INSTITUTE CURRICULUM IS BASED ON
MATHEMATICAL (FUNCTIONAL) VERIFICATION OF BOTH PROGRAMS AND
MODULES (DATA ABSTRACTIONS).

CLEANROOM SOFTWARE DEVELOPMENT

SOFTWARE DEVELOPMENT UNDER STATISTICAL QUALITY CONTROL

STRUCTURED SPECIFICATIONS OF FUNCTION AND USAGE
CERTIFIED RELIABILITY STATISTICS AT DELIVERY
PROCESS FEEDBACK TO MEET RELIABILITY OBJECTIVES
STATISTICAL TESTING OF INCREMENTAL RELEASES
PROGRAM VERIFICATION, NOT DEBUGGING, BEFORE RELEASE
VERIFICATION INSPECTION, NOT PROGRAM INSPECTION

PROJECT EXPERIENCE

IBM LANGUAGE PRODUCT	35K
HELICOPTER FLIGHT PROGRAM	35K
SPACE TRANSPORT PLANNING	45K

PRELIMINARY FINDINGS

MATHEMATICAL FALLIBILITY PRESENT BUT TRACTABLE
MATHEMATICAL ERRORS ARE PROGRAMMING BLUNDERS
SYNERGISM OF MATHEMATICAL VERIFICATION AND
STATISTICAL TESTING
HIGH PROGRAMMER MORALE AND ACCEPTANCE

FUNCTIONAL VERIFICATION OF SOFTWARE

SIMPLIFIED DENOTATIONAL SEMANTICS

OPERATIONAL SEMANTICS IS WORM'S EYE VIEW

DENOTATIONAL SEMANTICS IS BIRD'S EYE VIEW

CAN BASE ON SET THEORY RATHER THAN LATTICE THEORY

VERIFICATION PARADIGM

COMPUTE MEANING OF PROGRAM

USE MEANINGS OF PARTS IN COMPUTATION

COMPARE WITH MEANING OF SPECIFICATION

VERIFICATION WITH STEPWISE REFINEMENT

VERIFY EVERY SEQUENCE, SELECTION, ITERATION TOP DOWN

CONVERT ITERATION INTO RECURSION AND TERMINATION

WITH FUNCTION SPECIFICATION NO INVARIANT NEEDED

NEEDS A LOT OF SPECIFICATION REWRITING!

VERIFICATION BY PEOPLE

VARIABLE FREE THEORY NECESSARY FOR SCALE UP

PRE AND POST CONDITIONS CONSIDERED HARMFUL

DESIGN/PROOF DISCIPLINE CRITICAL

ADA SUGGESTIONS

ADA VERIFICATION LANGUAGE

ADD PROPOSITIONS AS TYPES

DEFINE ADA VERIFICATION SEMANTICS

ADA ENVIRONMENT FORMAL SOURCE SIZES

PROGRAM	5%
SPECIFICATION	25%
VERIFICATION	70%

BRING IN STATISTICAL TESTING

ADD USAGE STATISTICS TO SPECIFICATIONS

ADD STATISTICAL INFERENCE TO TEST ANALYSIS

EXPAND AUTOMATIC VERIFICATION EFFORTS

TOP DOWN NATURAL DEDUCTION

VERIFICATION COMPLEXITY THEORY

PEOPLE - MACHINE SYNERGISMS

**APPLYING SEMANTIC DESCRIPTION TECHNIQUES
TO THE CAIS**

by

Timothy E. Lindquist
Arizona State University
Tempe, Arizona 85287
Lindquis%asu@csnet-relay
(602) 965-2783

Roy S. Freedman
Polytechnic University

Bernard Abrams
Grumman Aircraft Systems

Larry Yelowitz
Ford Aerospace

April 15, 1986

ABSTRACT

Throughout the development of the CAIS, which is an operating system interface to be hosted on several systems, semantics have been an issue. Aside from the benefit to designers, having a formal description of the interface is important to various aspects of its use. The current effort to create a government standard of CAIS would certainly benefit from a formal definition. Increasing the transportability of Ada* software built on CAIS is one of its design goals. To assure a reasonable level of transportability will require a validation suite that may be applied to implementations. Although the need for validation has typically not extended beyond a single vendor, we now see its potential for savings in software development. Constructing proofs of tools or applications that use kernel facilities additionally motivates a formal description, and finally as we've seen in programming languages, formal descriptions can direct implementations. In this paper, various methods of description are analyzed regarding their applicability to kernel interfaces. The methods treated include English narrative, abstract machines, axiomatics, and denotational descriptions. For each method, we show an example from CAIS and analyze the methods applicability to various features.

Keywords. Kernel interfaces, operating systems, verification, axiomatic and denotational semantics.

*Ada is a registered trademark of the U.S. Government Ada Joint Program Office.

1. INTRODUCTION

This paper describes and evaluates alternative methods of specifying the semantics of kernel level facilities. Both formal semantic methods and informal methods are examined. The authors have been involved with an effort to develop a common set of services to support APSE (Ada Programming Support Environment) tools. The methods we describe are exemplified using this common set, called CAIS (Common APSE Interface Set, pronounced as case). CAIS is an operating system interface that supports software development tools.

If CAIS is implemented consistently on a variety of host systems then the effort needed to transport tools will be reduced. In the same manner as for the Ada Language, a validation capability is being developed for the CAIS [E&V 85]. Validation must address the consistency and completeness of CAIS implementations with respect to the specification. In doing preliminary work on developing validation tests, we found the need for a precise specification of the system. Various specification methods have been examined for their applicability to CAIS features. In this paper, we present and compare the applicability of each method.

Although the problem of describing kernel facilities has not received adequate treatment, the benefit of formal description is clear. Any effort to standardize on a low level interface, such as graphics or process management, needs a precise specification to be complete enough and unambiguous. As standards arise, we are seeing the development of validation mechanisms to assure consistency among implementations, as mentioned above with CAIS. It has also become clear that formal specifications can be used to direct implementation efforts. Technology is advancing to the point where directed implementations are as efficient as ad hoc implementations.

2. CAIS: A COMMON OPERATING SYSTEM INTERFACE

The Department of Defense has developed and standardized on the programming language Ada. When development environment tools are considered, however, a single language is only part of what is needed for transportability. Tools access environment data and control processes through operating system services. The combination of a standard language and a standard operating system would increase tool transportability.

CAIS defines a common interface to the operating system. The interface is a set of Ada packages containing procedures and data definitions that are used by Ada programs to request system services. If the format of the call for services (syntax) is standard and the response to the call (semantics) is the same, then the effect of a standard operating system has been achieved.

CAIS has been designed by a working group of the KIT/KITIA (Kernel APSE Interface

Team/Industry and Academic) under sponsorship of the Ada Joint Program Office through Naval Ocean Systems Center [KIT-82]. A Government Standard CAIS specification [CAIS-85] is currently being reviewed, and an effort is underway to address incorporating capabilities deferred from the initial design, such as distributed environments. Several prototypes and implementations are currently being developed.

3. SPECIFYING KERNEL FACILITIES

3.1 Syntax and Semantics

A typical CAIS facility is the OPEN procedure, whose format is shown in Figure 1. The procedure specification gives the procedure name, the parameters and their types. This format is the syntax and is expressed through Ada package specifications. The CAIS document augments the syntax with English narrative describing the call. The actions performed by OPEN are its semantics.

```
procedure OPEN(NODE: in out NODETYPE;  
              NAME: in NAMESTRING;  
              INTENT: in INTENTION := (1 => READ);  
              TIMELIMIT: in DURATION := NODELAY);
```

Figure 1. The OPEN facility's syntax.

Ada provides a well understood notation that completely and unambiguously defines syntax. Ada semantics are conveyed in English text and the Language Reference Manual states that meanings are as defined in Webster's Dictionary. Text benefits from the power and suffers from the ambiguities of a natural language specification. The English description is adequate for most purposes but is often incomplete and ambiguous.

One example is the OPEN statement of Figure 1. Its function is to create an association between an Ada program variable and a CAIS environment node. The internal variable, called a node handle, is used by the program to reference the node in operations. One parameter to OPEN is an array, called INTENT, that conveys intended access to the node being opened. Typical values of Intent are Read, Write, and ExclusiveRead. As an example of incompleteness, the explanation of OPEN does not indicate behavior if the Intent array contains overlapping or contradicting intents. What if both Read and ExclusiveRead are requested?

Natural language specifications contain implied assumptions about their context. An implied assumption of the open statement may be that a user doesn't care which one of a contradicting intents is chosen. Such a specification may be precise enough for a user but not for validating,

implementing, or arguing formally about programs using CAIS. Semantics of CAIS are mostly well defined, however, one can anticipate uses requiring a more thorough and formal description.

3.3 Methods of Supplementing a Semantic Definition

The semantics of CAIS is specified in MIL-STD-CAIS by English narrative, with some additional semantics implied by the Ada package specifications. The methods we present for supplementing CAIS semantics can be grouped into formal and informal methods. The formal methods are mathematical in nature and include axiomatic, denotational, and abstract machine notations. The informal methods are additional narrative and examples.

4. INFORMAL SEMANTIC SPECIFICATION

The informal methods of supplementing semantics, English narrative and examples, have strengths and weaknesses. English or other natural language narratives can be verbose, ambiguous, and context dependent. The interpretation of an English sentence depends on the background of the reader. Further, English words have many meanings. My dictionary lists 12 for "be", 33 for "beat", and 15 for "bend". There is, however, no match for the understandability and generality of English. Even texts in theoretical mathematics use more English than mathematical notations to communicate.

Description by examples is done through small programs or parts of programs using CAIS. Test cases from a CAIS test suite would make good examples since these are small programs exercising one CAIS feature. Examples are not general and not concise but are very understandable. When there is a choice of methods of specifying semantics the most precise, concise, and abstract method should be used. Formal mathematical methods, when they are applicable, usually meet these criteria. But there are still many cases where informal methods are needed. The informal methods are supplements and not replacements for formal methods. Examples of the use of informal methods to supplement CAIS semantics follow.

4.1 OPEN Facility

OPEN, as discussed above, establishes a connection between an external file and an internal node handle. Objects in CAIS are managed using a node model. A node can be a file node, a structure (directory) node, or a process node. Figure 2 is an example of the use of OPEN. It is a test case showing how an internal program variable called a node handle is connected to an external node by OPEN. The handle is then used to access the node. The example shows semantics in the sense that it shows how the OPEN procedure is used.

Applying Semantic Description Techniques to Kernel Facilities

Examples may not show what happens in any of the exceptional cases. Supplementary English narrative can be added showing what happens if, for example, incompatible Intents are presented to the procedure. The Intent array argument to OPEN could be:

(READ, WRITE, CONTROL)

There is nothing to stop a user from specifying an incompatible set of intents. If both WRITE and EXCLUSIVEWRITE are specified there is a conflict. The first Intent lets many users write simultaneously and the second permits only one at a time. This uncertainty can be resolved by additional narrative in the specification such as:

1. In the event of conflict use the most restrictive interpretation; thus, EXCLUSIVEWRITE has precedence over WRITE, or
2. In the event of conflict reject the call with an exception or
3. Any conflict resolution scheme is acceptable.

Any one of the above clarifications is sufficient from the viewpoint of creating validation tests. Which one is chosen is a design issue, however, without specifying one option, the validator is forced to make the design decision.

```

-----
-- CAIS TEST OF OPEN
-- Open by Name -- good data - take defaults
--
-- Open the node and verify with an inquiry
-- Precondition -- Initial State 1
-----
with Node_Management, use Node_Management;
with Node_Definitions; use Node_Definitions;
with TextIO; use TextIO;
procedure Open1 is
  Node1: Node_Type;
  Name: NameString;
begin
  ----- OPEN THE NODE -----
  Name := "DOT(F1)";
  Open(Node1, Name)
  PutLine ("Open has been called");
  ----- Verify with an inquiry -----
  if Is_Open(Node1) then
    PutLine ("Open Verified");
  else
    PutLine ("Open Failed");
  end if;
  ----- END OF TEST -----
end Open1;

```

Figure 2 Example of Open Procedure

4.2 The CAIS Node Model

Nodes, node handles, and path names are part of the node model. CAIS manages files, directories, devices, and processes by representing them as nodes in a network. Nodes are related to each other by relationships. Relationships are uniquely specified by a relation name and a relationship key, and a relationship may be either primary or secondary. Primary relationships are constrained to maintain a hierarchical structure of nodes. A typical network is shown in Figure 3.

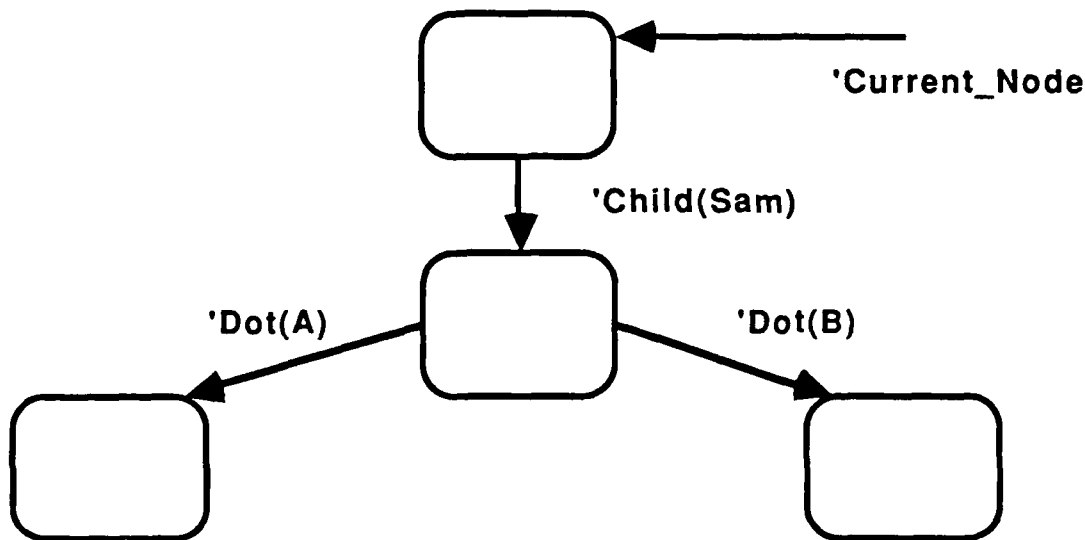


Figure 3. The Node Model.

An object, such as a file, is found by following the path of relationships from a known node to an object node. A path is specified by a path name made by concatenating all the element names along the path.

Another CAIS function is PRIMARY_NAME. The input to the function is a node handle, and the function returns the name of the primary path to the node. The Primary Name function returns the path name. For example the path from node Joe to node Sam is: 'Current_Node'Child(Sam). Since Current_Node is the default relation, the path can also be expressed as: 'Child(Sam). Child is the name of a relation. Sam is a particular instance of the relation. Since Sam is the only instance, the relation key "Sam" can be omitted and the path expressed as: 'Child . The relationship 'DOT(A) uses the default relation name (DOT) that can be expressed in two ways, Dot or (.). Two of many ways of expressing the path between the current node and "A" are:

'Child(Sam)'Dot(A) or 'Child(Sam).A

It is clear that the semantics of PRIMARY_NAME are ambiguous. There are many different strings that could be returned by the function. If the intended meaning of the designers was that any valid name string is acceptable, this could be stated in one sentence. However, allowing any string to be returned may promote implementations that hinder transportability. One way to supplement the semantics is with the following paragraph:

The full path name up to the Current_Node shall be returned. Relationship keys shall be spelled out even when they are unique. The dot relation shall be in the long form.

An example of a correct name using the network of Figure 3 is:

'Child(Sam)'Dot(A)

4.3 ANALYSIS OF INFORMAL METHODS

The primary strengths of Narrative and examples deal with the ease with which they may be created and comprehended. These advantages clearly outweigh any disadvantages during the initial design of the interface. A large portion of CAIS can be specified in a short description. The short description provides a quick introduction, to which details may be added. Ambiguity in the CAIS specification can be corrected by a combination of narrative and/or examples without requiring a formal description. The primary disadvantage is the difficulty in obtaining completeness in narrative specifications. Narratives cannot be used for formal arguments of correctness or arguments of interface characteristics. Descriptions using these techniques can best be viewed as a step in the process of developing more complete and formal descriptions. The most useful form of narrative is one that is developed in conjunction with or based on a formal description. Doing so reduces the tendency toward incompleteness or ambiguity.

5. ABSTRACT MACHINE DESCRIPTION

A report from a preliminary study of validation in an APSE [KAF-82] indicates that specifying the semantics of an interface such as CAIS requires more than a description of the syntax and functionality of its routines. Interactions that exist at the interface must be specified. Interactions may include routines that operate on a common data structure, routines that rely on data produced by a tool or routines depending on the Ada runtime environment. Furthermore, any pragmatic limits which apply to implementations must also be specified. These might include the length of identifier strings, field sizes, maximum number of processes, or the maximum number of times that an interface routine may be called.

Lindquist [LIN-84], describes an Ada-based Abstract Machine approach to describing CAIS. Using this approach functionality is operationally described in the form of Abstract Machine Programs. The programs are written in Ada. One is written for each CAIS routine to describe what that routine does. If there existed an executor for the programs (the Abstract Machine) then an operational definition of CAIS would exist. In later papers, [LIN-85a, SRI-85], the technique is demonstrated using the CAIS process model and applied to the problem of generating a validation mechanism for CAIS implementations. As depicted in Figure 4, an Abstract Machine consists of three components:

1. A processor,
2. A storage facility, and
3. An instruction set.

The processor is able to recognize and execute instructions from a predefined set. Each instruction has an action that the processor carries out in some data context. One component of the processor, called the environment pointer, indicates the data context in which an instruction is to be executed. Another component, called the instruction pointer, sequences processor execution through the instructions of the program.

The storage of the processor is memory for both data and programs. Data storage constitutes the environment used by the processor to execute instructions. The final component is the instruction set.

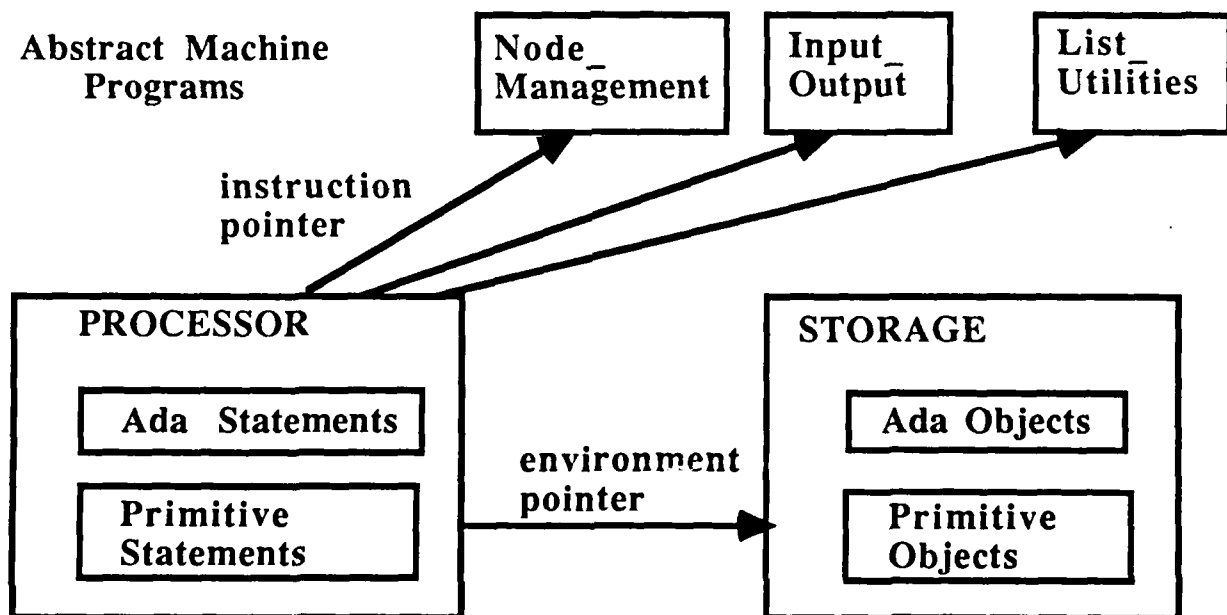


Figure 4. CAIS Abstract Machine.

Instructions are taken from the Ada language and are augmented needed primitive operations. The meanings of these primitive operations are left to the description of the Abstract Machine. Additional operations can be viewed as extending the instruction set of the Abstract Machine to include operations beyond the scope of Ada.

5.1 Ada Abstract Machines to Describe CAIS

Although other Abstract Machines could be used, this section presents one that is Ada-based.

Applying Semantic Description Techniques to Kernel Facilities

Several aspects of Ada make it a desirable choice for this description. One is the richness of the Ada control constructs and typing facilities. The most compelling reason for using Ada is compatibility with the uses of the CAIS specification. CAIS implementors and users are familiar with Ada, thus making an Ada-based Abstract Machine more comprehensible and useful. Although we ultimately desire a formal semantic description, the Ada-based Abstract Machine provides an excellent intermediate between Narrative and a formal description. The Ada-based Abstract Machine is easier to construct than a formal description and is more complete than Narrative. Although Ada has not been totally specified using a formal technique, the language's controlled definition provides an adequate basis for the Abstract Machine.

5.1.1 Node Management and List Utilities.

CAIS defines a set of list manipulation facilities that may be used in conjunction with the CAIS. Lists may be either named or unnamed. Named lists are those in which each element in the list has a unique name. The package includes routines for constructing generalized lists containing string, interger, list, and floating point elements. Routines to add, remove, and examine elements of a list are provided. An Ada-based Abstract Machine description of List Utilities follows the same approach as an Ada implementation. Figure 5 demonstrates the linking structure our definition uses for the example named list:

(APPLE => "GREEN", GRAPE => (RED => "SEEDLESS"))

List manipulation routines are constructed in Ada using this representation. A criticism of Abstract Machine descriptions is that the code itself specifies an implementation technique. Independent of the machinery selected, instructions to carry-out an operation must indicate an implementation technique. The meanings of the routines are not, however, derived from the code, but instead by the effect of executing the code on the Abstract Machine.

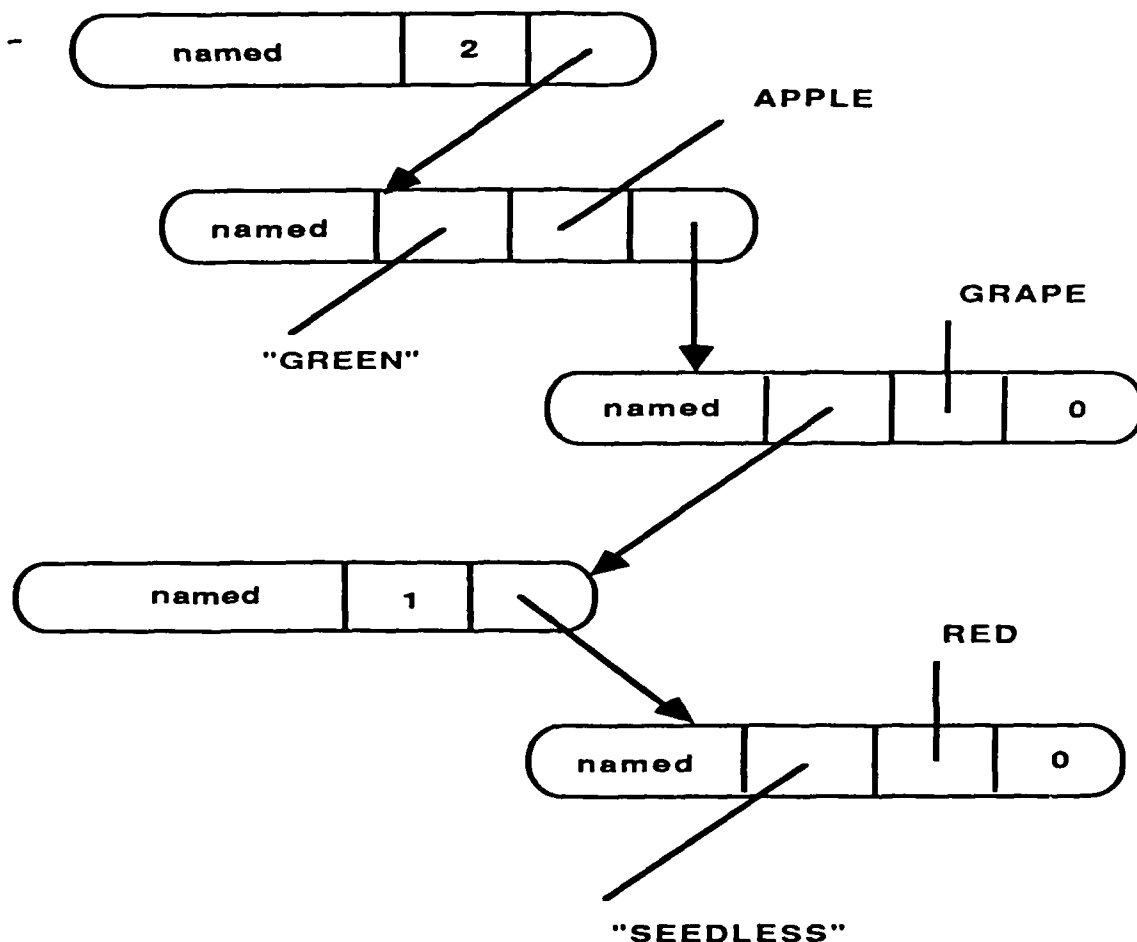


Figure 5. A sample list implementation.

The CAIS Node Management package includes facilities for manipulating nodes, which represent entities of the CAIS environment. Nodes may exist for processes, files, devices, queues, and node structures. Nodes may be related to one another using either restricted or unrestricted relationships. The restricted form of relationships, called primary relationships, require that each node have exactly one parent (except a single root node). The unrestricted form allows more general (cyclic) relationships to exist among nodes. CAIS Node Management provides routines for manipulating nodes, relationships among nodes, and attributes (either node or relationship attributes). Node Management also includes access mechanisms, which control the operations that a process may perform on a node.

The Abstract Machine description of Node Management relies heavily on data mechanisms of Ada. Included in the description are substantial uses of dynamic structures to represent nodes, to

store relationships between nodes, and to store lists and attributes. Access types, constrained record types and exception handling mechanisms are all used in the description.

Exception handling facilities are used throughout the CAIS to return status information to the tool calling CAIS. Using Ada exception mechanisms in the Abstract Machine provides an excellent definition of status returns for CAIS. With any other formal semantic description (axiomatic or denotational) a reasonable overhead equal to describing Ada language exceptions is incurred.

One use of Ada exceptions within Abstract Machines illustrates the problem of over specifying semantics. For instance, suppose the CAIS specification indicates an incomplete order for generating status exceptions to allow for flexible implementations. Thus, when a CAIS routine is called with arguments that would produce multiple status exceptions, the specification does not impose a complete order for checking. An Abstract Machine description does, however, fully specify the order of status checking.

5.1.2 Process Control

The Process Control section of CAIS provides routines to create and manage the execution of Ada programs. Facilities are included for different forms of invoking processes, awaiting completion, and manipulating built-in process attributes.

The Abstract Machine description relies on Ada's tasking facilities to describe asynchronous processes in the CAIS environment. For example, in the Abstract Machine description [SRI-85], a process node is represented as a dynamically created (allocated) record object. Components of the object contain instances of task types which provide the parallelism and synchronization needed for spawning and awaiting processes. A user's process structure is built dynamically and is a tree of tasks. Each process includes tasks for synchronization and for representing the Ada program. An example of two CAIS processes is shown in Figure 6. **Process_node_1** has spawned **Process_node_2**, and the spawned_process task is used to synchronize among processes. Again, the use of Ada's tasking facilities in the Abstract Machine description alleviate the need to formally redefine asynchronous facilities in some other descriptive technique. Both axiomatic and denotational approaches have a cumbersome time accommodating concurrency. Tasking is well understood by the users of a CAIS specification, which eases comprehension. However, we note that a formal specification of Ada tasking does not yet exist.

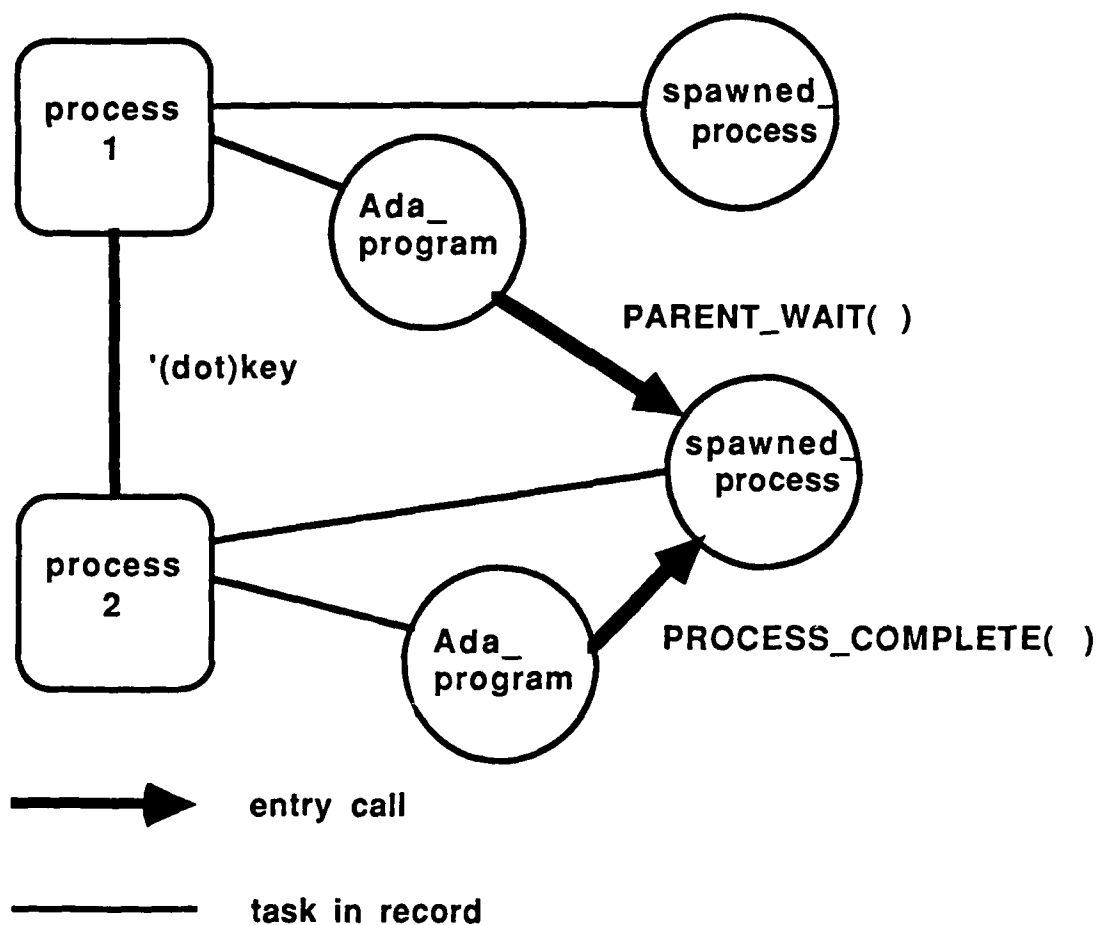


Figure 6. Sample CAIS process structure.

5.1.3 Input and Output.

Routines for manipulating file nodes of various types are included in the Input/Output section of CAIS. Further support is provided for common types of terminals and magnetic tapes. To construct an Abstract Machine description of this section of CAIS, the Abstract Program must create software devices that appear to the CAIS just as actual devices would appear. While it is possible to define a majority of the input/output facilities using an Ada-based Abstract Machine, some routines do not lend to formal specifications using any technique. Facilities to require the operator to physically mount or dismount tapes from a drive exemplify those difficult to define formally. Although one could formally define routines requesting these services, the need to formally define such facilities can be argued.

5.3 ANALYSIS OF ABSTRACT MACHINES

An Ada-based Abstract Machine description of CAIS provides some distinct advantages in the progression to a more formal specification of CAIS. Some of these advantages would be lost if the Abstract Machine description was not Ada-based. For example, an Ada-based description can be constructed quickly. If another language were used, then the problems of translating the meanings of asynchronous activity and exception handling into the notation of that language would need to be overcome. Additionally, an Abstract Machine description in another language would not be as comprehensible to the Ada community as is a specification in Ada. To be a complete formal specification of CAIS, an Ada-based Abstract Machine description must be accompanied by an appropriate formal specification of the machine instruction set. Inventing and defining appropriate instructions to augment Ada could be done to deal with drawbacks such as over specification.

6. AXIOMATIC DESCRIPTION

One of the ubiquitous comments received from the public review of CAIS 1.1 is the need for a semantics. There are a variety of methods for presenting a formal semantics, and this section treats the axiomatic approach. There is no escaping the fact that some degree of mathematical maturity is required to comprehend any formal semantics. It is our feeling, however, that axiomatic semantics is comprehensible to a large set of CAIS readers.

Axiomatics was first presented by Hoare [HOA-69], and has been applied to various languages; the most notable of which is PASCAL [HOA-73]. London [LON-78] has applied the method to EUCLID, which is especially interesting since the language was designed with the goal of simplifying program proofs. A large portion of this presentation is based on Yelowitz [YEL-84].

In a mathematical sense, a theory is defined by applying the Axiomatic method to a programming language. The theory consists of a language for expressing theorems, a set of axioms and rules of inference. A theorem of the theory is a program together with its input and output specifications. Minimally, it is required that all theorems of the system be programs which match their specifications; that is, the system must be sound. Axioms and rules of inference are defined to determine whether or not a program and its specifications form a theorem. If a program and its specifications are derivable from the axioms and rules, then they constitute a theorem.

By derivable, we mean that there exists a proof of the theorem in the system. A proof is a sequence of statements in the theory that begins with an axiom and ends with the theorem. Each statement in the sequence is either an axiom or it is a statement that can be written by applying a rule

of inference to statements preceeding it in the proof.

Syntactically, the theorems of the system take the form:

$$\vdash P \{ S \} Q.$$

Where S is a statement or set of statements of the programming language and P and Q are predicates (assertions) over the identifiers used in S . Our statements are Ada language statements augmented by calls to CAIS interfaces. The turnstile, $\vdash$, indicates that $P\{S\}Q$ is a theorem of the system. Intuitively, $P\{Q\}S$ can be interpreted to mean, if P is true before execution of S , then Q will be true after execution provided S halts.

An axiomatic semantic description of CAIS can be formulated in conjunction with that of the Ada language. Assuming that such a definition of the language already exists, we outline here how it may be augmented to accommodate CAIS. CAIS interfaces may be treated in the same manner as other procedures or functions invoked by an Ada program. Input and output predicates may be constructed to define what the procedure does. The free variables of the predicates are the parameters and nonlocals referenced by the procedure. A rule for the CALL statement defines how the input and output assertions are used to prove procedure calls. Although input and output assertions could be defined in this manner, we choose to represent the meaning of CAIS facilities with Axioms (schemes) to more accurately reflect the relationship between CAIS facilities and the language.

We now present the background needed for the Axiom scheme for the Node Model routine COPY_NODE. To do so requires formalization of notions such as types of nodes, contents of nodes, attributes of nodes, and relationships among nodes.

6.1 The CAIS Node Environment

The node environment can be described as a directed graph in which arcs are labeled and may possess attributes. We define NODES to be the set of nodes in an APSE.

The set ARCS includes all directed edges in the graph. Thus:

$$ARCS \subseteq (NODES \times NODES)$$

If the pair $(n_1, n_2) \in ARCS$ then there is a directed edge from n_1 to n_2 . We refer to an element of ARCS with the shorthand a_i . Labels formalize the relationships that ARCS represent. LABELS is a set of

(relation_name, relationship_key)

pairs associated with each arc in CAIS. The function LABEL names each arc with the appropriate pair as:

LABEL : ARCS --> LABELS

A pathname is a sequence of labels. Thus, using the Kleene star, all valid pathnames are in: **LABELS (LABELS*)**.

OUTARCS is a function providing for each node, a set of all arcs emanating from the node.

OUTARCS : NODES --> 2^{ARCS}

That is, an edge is in the set of out arcs of a node, n, when it emanates from n.

$a \in \text{OUTARCS}(n)$ iff $\exists n_1 \in \text{NODES}$ and $a = (n, n_1) \in \text{ARCS}$

Similarly we define **INARCS** to be the set of all edges emanating to a node.

INARCS: NODE --> 2^{ARCS}

The predicate **ISPRIMARY** partitions the set of arcs into primary and secondary relationships. CAIS requires that all primary relationships maintain the hierarchical structure of nodes. We describe this requirement using the following:

ISPRIMARY : ARCS --> {true, false}

Any node (except the system root) must have exactly one primary relationship emanating to it. This CAIS requirement can be expressed as:

$\forall n \in \text{NODES} | n \neq \text{SYSTEMROOT}$ and $\forall a, b \in \text{INARC}(n)$

$\text{SINK}(a) = \text{SINK}(b) \implies \text{not ISPRIMARY}(a) \text{ or not ISPRIMARY}(b)$

Where **SINK** is the node an arc emanates to: **SINK: ARC --> NODE**

CAIS specifies that distinct arcs emanating from a node must have distinct labels. To describe this property, we have the following predicate:

$\forall n \in \text{NODES}, \forall a_1, a_2 \in \text{ARCS}$

$a_1, a_2 \in \text{OUTARCS}(n)$ and $a_1 \neq a_2 \implies \text{LABEL}(a_1) \neq \text{LABEL}(a_2)$

For notational convenience, we define the following:

$\forall (x, y) \in \text{ARCS}, R \in \text{LABELS}$

- $P(x,R,y)$ denotes $ISPRIMARY((x,y))$ and $LABEL((x,y)) = R$
 $P(x,R,y)$ means there is a primary relationship from x to y labeled R , and

$S(x,R,y)$ denotes $\text{not } ISPRIMARY((x,y))$ and $LABEL((x,y)) = R$
 $S(x,R,y)$ means there is a secondary relationship from x to y labeled R .

There is a partitioning of the set of nodes into four disjoint subsets:

PROCESS_NODES,
 STRUCTURAL_NODES,
 FILE_NODES,
 DEVICE_NODES, and
 QUEUE_NODES.

These subsets, which represent the different types of CAIS nodes, allow the axiomatic description to distinguish characteristics particular to different types.

6.2 SEMANTICS OF COPY NODE

This interface is used to make a copy of a file or structural node having no primary relationships emanating from it. Secondary relationships emanating from the node are copied, as appropriate. The syntax of one overload of the routine is:

```
procedure COPY_NODE (FROM,TO BASE: in NODE_TYPE;
                    TO KEY:   in RELATIONSHIP_KEY;
                    TO_RELATION in RELATION_NAME :=
                    DEFAULT_RELATION);
```

Our goal is a predicate transformer for each interface of the CAIS. Since the transformers can be quite extensive, we present one by its parts. A shorthand notation is also used to avoid complexity. The predicate transformer for a NAME_ERROR is:

$$(\text{NOT } (RLN,KEY) \in \text{LABELS}) \text{ or}$$

$$(\exists n \in \text{NODES} \mid (BASE,n) \in \text{ARCS and } LABEL((BASE,n)) = (RLN,KEY))$$

$$\{ \text{COPY\_NODE (FROM, BASE, KEY, RLN) } \}$$

$$\text{NAME\_ERROR}$$

The meaning of this transformer is: if prior to executing the call to COPY_NODE the relation name, relationship key pair are either illegal or the node to be created already exists then, if

Applying Semantic Description Techniques to Kernel Facilities

execution of COPY_NODE completes, the predicate NAME_ERROR will be satisfied. To simplify and continue the example, we present an abbreviated form of the transformers for USE_ERROR, STATUS_ERROR and the functionality of COPY_NODE.

USE_ERROR is generated according to the following predicates. First, USE_ERROR is raised when there is a primary arc emanating from the source of copying.

$$\exists n \in \text{NODES} \mid (\text{FROM}, n) \in \text{ARCS} \text{ and } \text{PRIMARY}((\text{FROM}, n))$$

Next, when the node to be copied (FROM) isn't either a FILE_NODE, or a STRUCTURAL_NODE, USE_ERROR is generated:

$$\text{not } \text{FROM} \in \text{FILE\_NODES} \quad \text{and} \quad \text{not } \text{FROM} \in \text{STRUCTURAL\_NODES}$$

The status of a node is defined by the function NODE_STATUS as:

$$\text{NODE\_STATUS} : \text{NODES} \rightarrow \{\text{OPENED, CLOSED, UNOBTAINABLE, NONEXISTENT}\}$$

With this we can define the predicate transformer producing a STATUS_ERROR as:

$$\text{NODE\_STATUS}(\text{FROM}) \neq \text{OPENED} \text{ or } \text{NODE\_STATUS}(\text{BASE}) \neq \text{OPENED}$$

Normal Action. With these definitions for exceptional conditions, we can define the predicate transformer for a call to COPY_NODE in which the copying actually takes place. The exceptional status conditions given above can all be placed into a single predicate transformer. To do so, the precondition for each precludes the others, as does the corresponding postcondition. Each unit of the predicate transformer corresponds to a separate action. Below is the transformer describing normal operation of COPY_NODE. The precondition is abbreviated as not STATUS_EXCEPTION to indicate that no status returns occur. In that instance copying takes place.

$$\begin{aligned} &\text{not } \text{STATUS\_EXCEPTION} \{ \text{COPY\_NODE}(\text{FROM}, \text{BASE}, \text{RLN}, \text{KEY}) \} \\ &\quad \exists n \in \text{NODES} \mid (\text{BASE}, n) \in \text{ARCS} \text{ and } P(\text{BASE}, (\text{RLN}, \text{KEY}), n) \\ &\quad \text{and } \text{LABEL}((\text{BASE}, n)) = (\text{RLN}, \text{KEY}) \quad (0) \\ &\quad \text{and } \text{CONTENTS}(n) = \text{CONTENTS}(\text{FROM}) \quad (1) \\ &\quad \text{and } (\text{ATTRIBUTES}(n) = \text{ATTRIBUTES}(\text{FROM})) \quad (2) \\ &\quad \text{and } \text{KIND}(n) = \text{KIND}(\text{FROM}) \quad (3) \end{aligned}$$

and $\exists a \in \text{ARCS} \mid a=(n, \text{BASE})$ and $\text{LABEL}(a) = (\text{PARENT})$
and $S(n, (\text{PARENT}), \text{BASE})$ (4)

and $\forall a \in \text{ARCS} \mid a=(\text{FROM}, \text{FROM}) \exists b \in \text{ARCS} \mid$
 $b=(n, n)$ and $\text{LABEL}(a)=\text{LABEL}(b)$ (5)

and $\forall a \in \text{ARCS} \mid a \neq (\text{FROM}, \text{FROM})$ and $\text{LABEL}(a) \neq (\text{PARENT})$
 $\exists b \in \text{ARCS} \mid b=(n, \text{SINK}(a))$ and $\text{LABEL}(b)=\text{LABEL}(a)$ (6)

The postcondition for normal operation is lengthy, so its components are explained by line number. Line (0) indicates that a new node, n , has been created with a primary relationship emanating from BASE to the node. The relation and key are as specified through arguments. Note, however, that the CAIS indicates that the key may not be the argument. If a '#' appears as the key or appended to the key, then CAIS returns a unique key. This could be expressed axiomatically by adding additional conjuncts to both the pre and post assertions.

Lines (1), (2), and (3) indicate that the contents, attributes, and kind of the copied node match the original. Lines (4), (5), and (6) describe the newly created relationships emanating from the copy. Line (4) indicates that the secondary relationship, parent, for the copied node is set to BASE . CAIS indicates that any secondary relationships that emanate from the node to be copied must exist in the copied node as relations emanating back to the copied node. Line (5) defines this situation. Note that it is not necessary to specify only secondary relationships in the predicate since there are no primary relationships emanating from a node to itself. Line (6) indicates that there exists a secondary relationship in the copied node for all others of the from node. Thus, for all arcs from FROM , which don't point to FROM and which aren't parent relationships, there is a corresponding arc from the copied node with the same destination and label.

6.3 Analysis of Axiomatic Semantics

Axiomatic descriptions that rely on first order predicate calculus, which we have assumed here, can be characterized as removing all temporal information from the description. Having no order specified alleviates the problem of over specification that was found with Abstract Machines. Since time is not specified, one is tempted to state that some forms of status returns from kernel interfaces can't be specified. For instance, suppose the kernel indicates that "when conditions for status A and status B are both satisfied, that A is to be signaled". This can, however, be described axiomatically with predicates indicating that B is raised only when the conditions causing it exist and those causing A don't; that is, it is not a temporal condition.

Applying Semantic Description Techniques to Kernel Facilities

There are, however, two problems arising from the lack of temporal information. First, aliases may exist. In CAIS, two names within a CAIS implementation may refer to the same object. For example, suppose a single object is used as the argument to two or more **in/out** parameters for an interface. To answer the question: which value produced for the parameters will be given to the argument, requires temporal information about the implementation (indicating which receives a value last). Second, asynchronous and parallel computations require greater descriptive capability. The inability to specify time dependencies also implicates the inability to specify time independencies. The CAIS process model provides interfaces for concurrently executing processes, as well as for communication and synchronization among processes.

Exception handling causes no problem to axiomatic descriptions providing that the routine signaling the exception also handles it. In the CAIS this is rarely the situation. Exceptions are used to return status information. Although an axiomatic description can be generated to indicate that a status exception has been raised, the action performed to handle the exception is cumbersome to describe. Thus although we can describe the CAIS, we can't describe the meaning of a program that uses CAIS facilities. Binding a raised exception to a handler in Ada depends on the execution flow through the program. The procedure call history is needed when nested procedure calls are made. Ada's rule for binding exceptions requires that the exception be propagated outward to all calling procedures until one containing a handler is found. The program execution path needed for this binding is not available from static analysis.

7. DENOTATIONAL DESCRIPTION

7.1 Denotational Semantics: Pragmatics

The denotational approach to formal semantics involves specifying abstract mathematical meanings to objects, in such a way that the meanings of the objects are modelled by the mathematical abstractions giving the meanings of the elements making up the object. The mathematical entities that are used for this purpose (the denotations) are well-understood classes of sets and functions. The denotational approach is suitable for modelling machine-independent meanings because of its emphasis on mathematical constructs. The denotational approach has frequently been used for the formal implementation-independent specification of programming languages, and for deriving rules for proofs of program properties (an axiomatic semantics).

The essential idea in a denotational semantics is to map the syntactical structures (some sets and functions) of a language onto some semantic structures (other sets and functions). This is done so that every legal program in a language can be mapped into its meaning. The approach taken is to describe the semantics of a construct in terms of its sub-constructs. The use of the denotational

approach is applicable to certain types of sets, called domains, in order to insure convergence in the recursive application of functions. The formal mathematics of this approach was presented by [SCO-71].

There are several notations (or "meta-languages") for specifying a denotational semantics. The most common one, used by [TEN-81], [GOR-79] and [STO-79] is a variant of Lambda Calculus. This notation, while mathematically precise, is hard to read by many programmers and language implementers. Other notations that have also been proposed include the "Ada-like" notation in the Ada Formal Semantic Definition [INR-80], and the notations developed in the Vienna Definition Method [BJO-82].

Many of these notations have automated facilities that help evaluate and sequence a large number of recursive function calls that establish the meaning of a construct. For example, [KIN-83] has developed tools for testing the denotational semantic definitions of programming languages, as long as these languages are defined in AFDL+ (an extension of the INRIA notation). Mosses [MOS-76] has also developed the Semantics Implementation System based on the notation in [GOR-79]. These systems run programs that "execute" the meta-language equations defining the semantics of a construct. In one sense, development of these tools results in an operational semantics of a construct.

Denotational semantics have been used to formally specify programming languages, compilers [CLE-85], interpreters [STO-79], and databases [BJO-82]. There is also a formal specification of concurrency presented using denotational semantics [CLI-81]. Some of the issues involved with specifying kernel facilities based on the denotational approach were first addressed in [FRE-82] and [FRE-85]. In the following sections, we show what is entailed to develop a denotational semantics for kernel interfaces.

7.2 Denotational Semantic Domains

The denotational semantics of a kernel interface language consists of the semantics of procedure and function calls, as well as the semantics of expression evaluation. In order to create this denotational semantics, we need to specify the following components:

- Syntactic Domains
- Syntactic Clauses
- Semantic Domains
- Semantic Functions
- Semantic Clauses

The syntactic domains of a language consists of different syntactic categories that may be assigned meaning. These categories may (recursively) define other categories; to assure convergence, domains are specified. Some examples of syntactic domains are a domain of identifiers, a domain of commands, and a domain of expressions. For CAIS interfaces, these domains consist of identifiers, expressions, commands, and declarations.

The syntactic clauses show how a syntactic category may be described in terms of sub-categories. For example, one clause may specify that all kernel interface commands have the form:

$$C ::= \text{open}(E) \mid \text{close}(E)$$

where E is in the domain of expressions. The notation for syntactic clauses usually follows the notation for specifying the concrete syntax (phrase structure) of a language. However, since only the meanings of constructs and sub-constructs are emphasised, and not how a construct is formed, this type of syntax is termed the abstract syntax.

The semantic domains consist of well-understood domains that are either given (like the domain $\text{Bool} = \{\text{TRUE}, \text{FALSE}\}$) or are constructed from other domains. These domains are the actual "denotations" for our semantics. The most important of these domains are the Environment, the Store, and the Continuation domains. For example, an **Environment** domain may be described by the domain of functions from the domain of identifiers Ide to the domain of denotable values Dv , or

$$\text{Env} = \text{Ide} \rightarrow \text{Dv}$$

The domain of denotable values must be defined in terms of other domains: the denotable values usually contain the domain of locations. The Environment is changed by the elaboration of definitions. **Stores** may be described by the domain of functions from the domain of Locations Loc to the domain of Storable Values Sv , or

$$\text{Stores} = \text{Loc} \rightarrow \text{Sv}$$

Stores are changed by the execution of commands. The continuation domains may be described by functions from "intermediate results" to "final results." Final program results are usually expressed in terms of the Store domain. For example, since the effect of executing a command is to change the Store, the domain of command continuations is defined by

$$\text{ComCont} = \text{Store} \rightarrow \text{Store}$$

As another example, since the effect of evaluating expressions is a value and a store (from possible "side-effects"), the domain of expression continuations is

$$\text{ExpCont} = [\text{Dv} \times \text{Store}] \rightarrow \text{Store}$$

The above expression may also be written as

$$\text{ExpCont} = [\text{Dv} \rightarrow \text{Store}] \rightarrow \text{Store}$$

and also as

$$\text{ExpCont} = \text{Dv} \rightarrow \text{Store} \rightarrow \text{Store}$$

This particular form of function notation (the "curried" form) is what makes traditional denotational semantics difficult to read.

The semantic functions are functions that specify the denotation of the syntactic domain constructs in terms of the semantic domain constructs. For example, the semantic function for expressions may be

$$E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Dv}$$

This expresses the fact that the semantics of "evaluating an expression" is a value that depends on an environment and a store. Semantic functions are defined for all syntactic domains.

The actual semantics for the constructs that range over all syntactic domains are defined by semantic clauses. A semantic clause is a semantic function definition for a particular syntactic construct. In one sense, the semantic functions form specifications, while the semantic clauses actually "implement" the semantics. For example, the evaluation of the expression "1=1" denotes TRUE, given an arbitrary store *s*, and an arbitrary environment *u*:

$$E [1=1] u s = \text{TRUE}$$

Semantic functions traditionally utilize square brackets around syntactic constructs to increase readability. Other notation for semantic clauses may correspond to more familiar programming language syntax. For example, in the AFDL [INR-80] "Ada-like" notation, the semantic function *E* for expressions may be represented as

```
function EVAL_EXPRESSION ( T: Syntax_Tree; En: Environment; S: Store)
  return Denotable_Values;
```

The semantic clauses for all expressions would correspond to the function bodies of EVAL_EXPRESSION, for all possible elements of Syntax_Tree. The disadvantage of this notation is its ineconomy: other functions (and the non-Ada like "function type") must be defined to achieve all meanings of the functional notation form for *E*. For example,

$$E [\text{open} (E1, I2, I3, E4)] u$$

is a function, not a value.

7.2.1 An example of Denotational Semantics for the Specification of Kernel Interfaces

We provide an example of the denotational approach to describe the kernel interfaces of CAIS package `Node_Management`. This example shows the beginning specification that must be specified for a denotational semantics: the domains `Node` and `Asv`, as well as most semantic clauses are left incomplete.

Kernel Facility: package `Node_Management`

Syntactic Domains

Id	The domain of identifiers with elements <code>I1, I2, ...</code>
Exp	The domain of expressions with elements <code>E1, E2, ...</code>
Com	The domain of commands with elements <code>C1, C2, ...</code>
Dec	The domain of declarations with elements <code>D1, D2, ...</code>

Syntactic Clauses

```
C ::= open (E1,I2,I3,E4);  
      | close (I1,I2,I3,I4);  
      | change_intent (I1,I2,E3);  
      | copy_node(I1,I2,I3,I4);  
      | copy_tree (I1,I2,I3,I4);  
      | rename (I1,I2,I3,E4);  
      | link (E1,E2);  
      | iterate (I1,I2,I3,E4,E5,E6);  
      | get_next (I1,I2);  
      | set_current_node (E1,E2);  
      | get_current_node (I1);
```

```

E ::=  is_open (I1)
      | kind (I1)
      | primary_name (I1)
      | primary_key (I1)
      | primary_relation (I1)
      | path_key (I1)
      | path_relation (I1)
      | obtainable (I1, I2, E3)
      | more (E1)
      | is_same (E1,E2)

D ::=  I1: node_iterator ;
      | I1: relationship_key_pattern := E1;
      | I1: relation_name_pattern := E1;

```

Semantic Domains

```

Env   The domain of environments with elements u:
        Env = Ide --> {Dv + {unbound}}

Dv    The domain of denotable values with elements d:
        Dv = Loc + Asv + Cc (Exceptions are denotable.)

Loc   The domain of locations with elements l.

Asv   The domain of assignable values with elements a.

Store The domain of stores with elements s:
        Store = Loc --> {Sv + {unused}}

Sv    The domain of storable values with elements v:
        Sv = Node + Asv

Node  The domain of nodes with elements n.

Cc    The domain of command continuations with elements c:
        Cc = Store --> Store

Ec    The domain of expression continuations with elements k:
        Ec = Dv --> Cc

Dc    The domain of declaration continuations with elements d:
        Dc = Env --> Cc

```

Semantic functions

Semantics of expressions:

E: Exp --> Env --> Ec --> Cc

Semantics of commands:

C: Com --> Env --> Cc --> Cc

Semantics of declarations:

D: Dec --> Env --> Dc --> Cc

Semantic Clauses (some examples)

Commands

C [open (E1,I2,I3,E4)] u c = {meaning}

Expressions

E [is_open (I1)] u k = {meaning}

Declarations

D [I1: node_iterator] u d = {meaning}

7.3. Analysis of Denotational Approach

The denotational approach to formal semantics can adequately specify kernel interfaces, provided one interprets these interfaces as defining a language. The complete specification of CAIS semantics for storage management and input/output can also be expressed, although it would be a laborious undertaking, even if aided by automated tools. The major tasks in these areas involve selecting a formal mathematical model for the CAIS data structures and devices. These formal models would then be represented in the notation chosen for the domains. Semantics for process management can also be described in the denotational style, assuming that a formal model of concurrency (like Actor Semantics) is also similarly selected.

The denotational approach is not an alternative method to specifying semantics, rather, it emphasizes a different perspective toward specification. The denotational approach corresponds to a "top-down" solution to the problem of defining a language: the emphasis is on developing mathematical domains and functions to model machine meanings resulting from program execution. The operational approach corresponds to a "bottom-up" solution, whereby the emphasis is on constructing machine operations that will execute programs. An algebraic semantics is also a denotational semantics; in this approach, other specific mathematical constructs are used (more specific than domains) for representing the denotations. As observed above, a denotational specification becomes an operational specification if tools are provided that can "execute" the

Applying Semantic Description Techniques to Kernel Facilities

denotational semantic notation. Both approaches are used to construct rules of program properties to enable an axiomatic semantics.

8. CONCLUSIONS AND RECOMMENDATIONS

We have described how several semantic description techniques would be applied to a set of kernel facilities, using CAIS as an example. Considering informal methods, such as English narrative and example use, we have shown that these techniques are most useful during the developmental stage. They are quickly prepared and easily comprehended, which are important criteria for design reviews. The techniques lack in that it is easy to prepare descriptions that don't adequately treat details and are ambiguous. One recommendation is to explore a narrative description that is developed in close conjunction with a formal description. By doing so, the resulting description would be precise and nearly complete, as provided by the use of a formal definition as a basis. Further, the result would be more comprehensible than the formal specification.

Abstract Machine, Axiomatic and Denotational descriptions of kernel facilities have also been studied. These techniques have all been found to contain strengths and weaknesses with respect to the task at hand. The Abstract Machine description we presented, while comprehensible to the Ada community, lacks in applicability to other sets of interfaces. Further, the reader of Abstract Machine programs is tempted to infer a single implementation technique. It is all too easy to adopt the techniques used in the Abstract Machine. The primary advantages of the Abstract Machine descriptions presented are:

1. All sections of the CAIS are equally well described.
This is an attribute that is not shared with other methods.
2. The technique lends to an early and complete operational definition.
3. Although the description is not formal, it defines the CAIS in terms of the Ada language; thus centralizing related products.

An axiomatic description of the node management facility COPY_NODE is presented in the paper as an example. It demonstrates an application amenable to axiomatic description. With few exceptions, an axiomatic description of the Node Management section of CAIS provides a

straightforward semantics. As noted, it is difficult to describe exception status returns and constructions allowing aliases Axiomatically. To describe the process control facilities of CAIS, additional formalism is needed. Additionally, an axiomatic description of input/output facilities would be bulky. The Axiomatic method does, however, lend itself to proving properties of programs using CAIS facilities.

Adaptation of denotational semantics to CAIS is also straightforward for the Node Management facilities. Existing denotational mechanisms can be applied directly from denotational descriptions of programming languages. Again with this approach, input/output and process control present the greatest challenge to a concise denotational description.

It is not clear which approach is best (or whether one is indeed best from all perspectives). An operational approach would probably be easier to understand (but harder to modify or check for consistency or completeness) than a denotational specification; conversely, a denotational specification is more amenable to a machine independent meaning. This last characteristic is important for achieving interoperability and transportability. On the other hand, the use of denotational semantics for the specification of concurrent computation in Ada has not been as adequately addressed as in some other languages; this implies that for process management, at least, many researchers are more comfortable with an operational approach.

9. REFERENCES

- [HOA-69] Hoare, C.A.R. "An axiomatic basis for computer programming", *Communications of the ACM*, Vol. 12, No. 10, (Oct. 1969) pp.576-83.
- [HOA-73] Hoare, C.A.R. and Wirth, N. "An axiomatic definition of the programming language Pascal", *Acta Informatica*, Vol. 2, pp. 335-55.
- [KAF-82] Kafura, D., Lee, J.A.N.; Lindquist, T.E. and Probert, T. "Validation in Ada programming support environments", Technical Report Department of Computer Science, CSIE-82-12, Virginia Tech Blacksburg Virginia.
- [LIN-84] Lindquist, T.E. and Facemire, J.L., "A specification technique for the common APSE interface set", *Journal of Pascal, Ada and Modula-2*, Sept/Oct.
- [LIN-85] Lindquist, T.E. and Facemire, J.L. "Using an Ada-based abstract Machine description of CAIS to generate validation tests", *proceedings of the Washington Ada Symposium*, ACM, March 1985.
- [LON-78] London, R.L.; et.al "Proof rules for the programming language Euclid", *Acta Informatica* Vol. 10, pp. 1-26.

- [SRI-85] Srivastava, C.S. and Lindquist, T.E., "An abstract machine specification of the process node section of CAIS", *proceedings of the Annual National Conference on Ada Technology*, Houston Texas, March 1985.
- [YEL-84] Yelowitz, L. "Toward a formal semantics for the CAIS", *Public Report of the KIT/KITIA*, Vol. III, 1984.
- [BJO-82] Bjorner, D., Jones, C., *Formal Specification and Software Development*, Prentice-Hall International Series in Computer Science, 1982.
- [CLI-81] Clinger, W., *Foundations of Actor Semantics*, AI-TR-633, MIT Artificial Intelligence Laboratory, May, 1981.
- [CLE-85] Clemmensen, G., Oest, O., "Formal Specification of an Ada Compiler- A VDM Case Study," Dansk Datamatik Center, 1983-12-31, 1985.
- [FRE-85] Freedman, R.S., *Programming with APSE Software Tools*, "Chapter 5: Addendum: Formal Semantics," Petrocelli Books, Inc., Princeton, 1985.
- [FRE-82] Freedman, R.S., "Specifying KAPSE Interface Semantics," in *Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume II* (P. Oberndorf, ed.), NOSC TD 552, October, 1982.
- [GOR-79] Gordon, M., *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York 1979.
- [INR-80] INRIA, *Formal Definition of the Ada Programming Language (Preliminary Version for Public Review)*, Ada Joint Program Office, November 1980.
- [KIN-83] Kini, V., Martin, D., Stoughton, A., *Tools for Testing the Denotational Semantic Definitions of Programming Languages*, ISI/RR-83-112, USC, May, 1983.
- [MOS-76] Mosses, P., "Compiler Generation Using Denotational Semantics, in *Lecture Notes in Computer Science*, Vol. 45, Springer-Verlag, New York, 1976.
- [SCO-71] Scott, D., Strachey, C., "Towards a Mathematical Semantics for Computer Languages," *Proceedings of the Symposium on Computers and Automata* (ed. J. Fox), Polytechnic Institute of Brooklyn, New York, 1971.
- [STO-79] Stoy, J., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1979.
- [TEN-81] Tennent, R., *Principles of Programming Languages*, Prentice-Hall International, 1981.

Applying Semantic Description
Techniques to CAIS

May 14, 1986

presented by:

Timothy E. Lindquist

Arizona State University
(csnet: Lindquis@asu)

other authors:

Roy S. Freedman

Bernard Abrams

Larry Yelowitz

Polytechnic University

Grumman Aircraft Systems

Ford Aerospace

OUTLINE

- Motivation and Background
- English Narrative and Examples
- Ada-Based Abstract Machine
- Axiomatic
- Denotational

Why a Semantic Description of CAIS?

- Directed Implementations
- User's Reference (tool writer)
- Constructing Proofs of Tools
- CAIS Design Feedback
- Validation of Implementations
- Definition of the Interface

Who are the Users?

- CAIS Designers
- Tool Writers
- Implementors
- CAIS Validation Contractor

CAIS NODE MODEL

NODE KINDS :

STRUCTURAL

PROCESS

FILE (Secondary storage, queue,
terminal, tape drive)

RELATIONSHIPS

PRIMARY or SECONDARY

RELATION NAME (Predefined, User-defined)

RELATIONSHIP KEY (Latest_key)

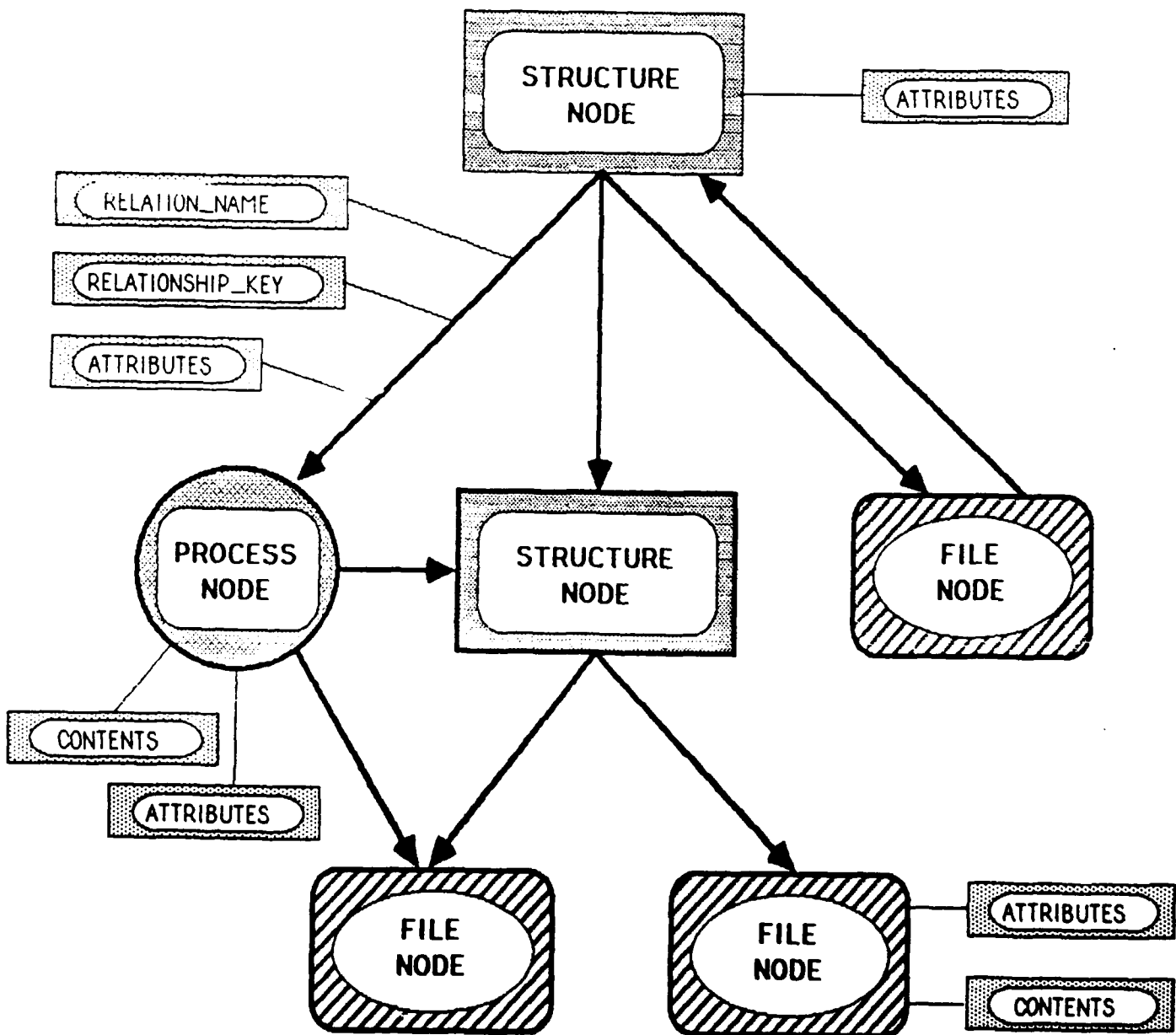
ATTRIBUTES

CONTENTS

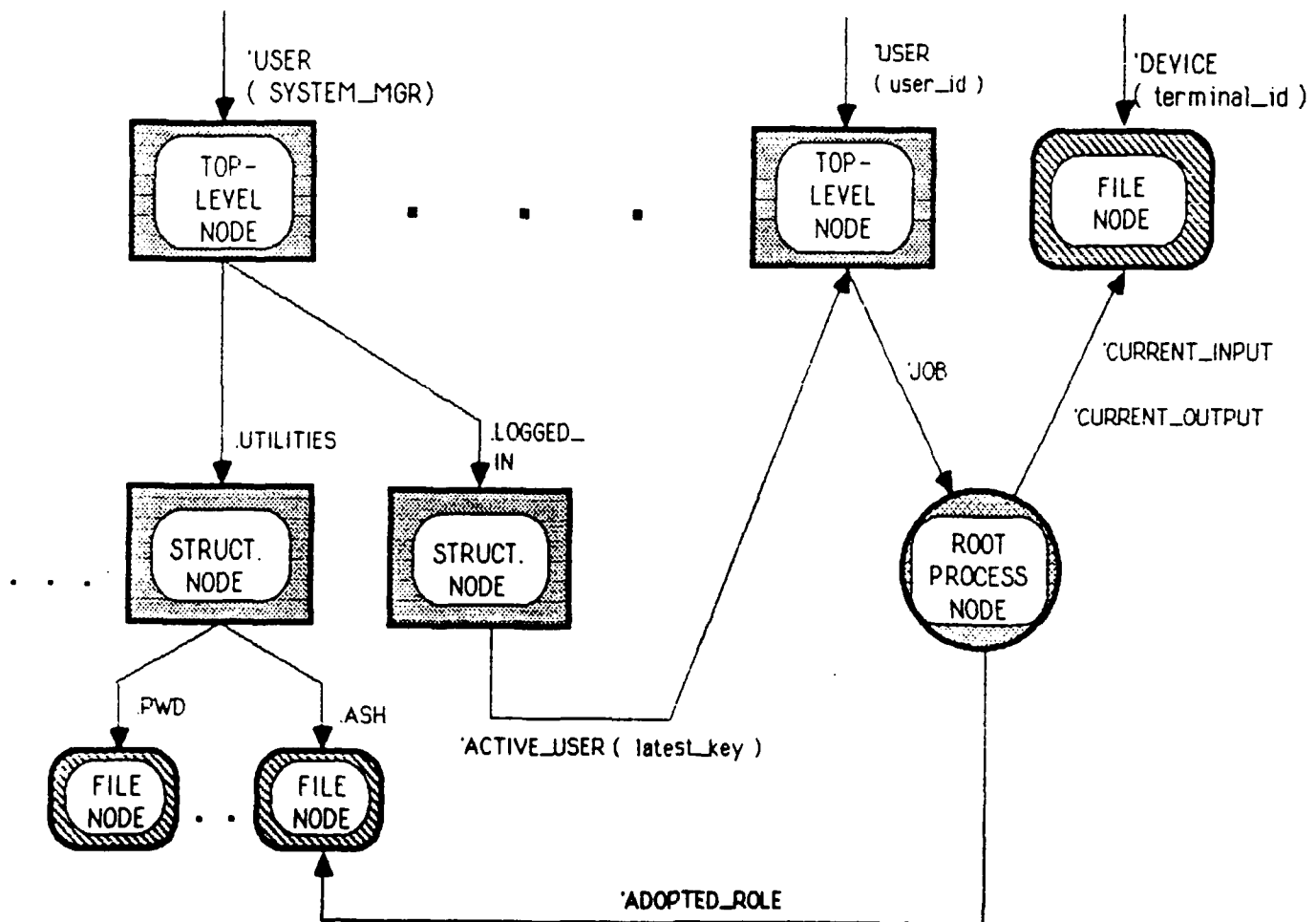
PATHNAME

'USER(JONES)
'USER(JONES)'DOT(A_DIR)
'USER(JONES).A_DIR

CAIS NODE MODEL



LOGIN PROCESS



LIST UTILITIES

LIST KIND :

EMPTY	()
UNNAMED	("A", "B")
NAMED	(A => "HI", B => "LOW")

ITEM KIND :

STRING	("APPLE") (PASSWORD => "DAVID")
INTEGER	(2, 7) (AGE => 30)
FLOAT	(33.3, 45.0) (WEIGHT => 162.5)
IDENTIFIER	(OPEN) (MODE => PIPELINE)
LIST	(("DAVID", 2), (OPEN)) (SIZE => (2.2, 4))

PROCESS CONTROL

SPAWN - immediate return to the calling process

INVOKE - return when the process completes

CREATE JOB - creates root process node

DETERMINE STATUS - ready, suspended,
terminated, aborted

APPEND / GET RESULTS

GET PARAMETERS

ABORT / SUSPEND / RESUME

Informal Semantic Specification

Natural Language:

- verbose
- ambiguous
- context dependent
- incomplete

By Example – small programs or parts using

- Tool fragments
- Test programs from validation suite

Recommendation: Construction based on

Formal description to:

- reduce tendency to incompleteness
- remain comprehensible and concise

Specifying the CAIS using an Ada-Based Abstract Machine

Specification Components:

- Syntax -- Ada Package Spec's
- Functionality -- Abstract Machine
- Interactions
- Pragmatic Limits

Abstract Machine:

1. Storage -- Ada and primitives
2. Instructions -- Ada and primitives
3. Processor

Pragmatic Limits:

- Use Limits OPEN_NODE_COUNT
- Value Limits NAME_STRING of at least 255 characters

Interactions:

- Within CAIS
 - Functional Dependency
 - Use Sequences
- With Tools
 - Login creates needed attributes
 - Linker produces load image
- With Ada Run-Time System
 - Exception Handling
 - Parameter Mechanisms

Axiomatics Applied to CAIS

Approaches:

- User defined extension
 - Use an approach to verifying the Abstract Data Types. (alphard)
- Language defined extension
 - Invent rules for each routine and supporting axioms describing CAIS types and objects (node model)

Example:

```
procedure COPY_NODE (  
    FROM, TO_BASE: in  NODE_TYPE;  
    TO_KEY: in  RELATIONSHIP_KEY;  
    TO_RELATION: in  RELATION_NAME  
        := DEFAULT_RELATION);
```

Make a copy of a file or structural node having no primary relationships emanating from it. Secondary relationships emanating from the node are copied as appropriate.

Definitions:

$NODES$ = the set of CAIS nodes

$ARCS$ subset of $NODES \times NODES$
such that $(n1, n2)$ in $ARCS$ if there
is a directed edge from $n1$ to $n2$.

$LABELS = \{ (RLN, KEY) \mid$
 $RLN \text{ and } KEY \text{ are Ada identifiers} \}$

$LABEL = ARCS \rightarrow LABELS$
a name function for arcs.

Axiomatic Summary:

- Alias – CAIS objects may have many names.
- Asynchronous Facilities (concurrency)
- Exception Propagation
- Input and Output Facilities

Denotational Approach

Define meanings functionally in terms of the syntactic components of the language elements (interface procedures)

Example, node model:

Syntactic domains

Id	identifiers $I_1, I_2, \dots$
Exp	expressions $E_1, E_2, \dots$
Com	commands $C_1, C_2, \dots$
Dec	declarations $D_1, D_2, \dots$

Syntactic Clauses

$C ::=$	$\text{open} (E_1, I_2, I_3, E_4);$
	$\text{close} (I_1, I_2, I_3, I_4);$
	$\dots$

$E ::=$	$\text{is\_open} (I_1);$

Semantic Domains

Env	environments u $Env = Ide \rightarrow [Dv + \{unbounded\}]$
Dv	denotable values d $Dv = Loc + Asv + Cc$
Loc	locations l
Asv	assignable values a
Store	stores s $Store = Loc \rightarrow [Sv + \{unused\}]$
Sv	storable values v $Sv = Node + Asv$
Node	nodes n
Cc	Command continuations c $Cc = Store \rightarrow Store$
Dc	declaration continuations d $Dc = Env \rightarrow Cc$
Ec	expression continuations k $Ec = Dv \rightarrow Cc$

Semantic functions

Expressions:

$$E: \text{Exp} \rightarrow \text{Env} \rightarrow \text{Ec} \rightarrow \text{Cc}$$

Commands:

$$C: \text{Com} \rightarrow \text{Env} \rightarrow \text{Cc} \rightarrow \text{Cc}$$

Declarations:

$$D: \text{Dec} \rightarrow \text{Env} \rightarrow \text{Dc} \rightarrow \text{Cc}$$

Conclusion

- There is no single formal method that will uniformly satisfy needs
- A combination of methods should be investigated
- A formal definition of CAIS should be constructed

MAVEN: The Modular Ada Validation Environment

Norman H. Cohen

SofTech, Inc.
One Sentry Parkway, Suite 6000
Blue Bell, Pennsylvania 19422-2310

NCohen@Ada20

Abstract. The Modular Ada Validation Environment, or MAVEN, is proposed as an integrated set of tools to support the validation of Ada programs, by formal verification and other means. These tools could reside in an Ada Programming Support Environment. The principles underlying MAVEN are that proofs should be based on implementation-independent proof rules; that large Ada programs should be validated module by module; that validation by formal proof, informal proof, testing, code walkthroughs, and other means should be integrated; and that the specification of critical properties weaker than correctness should be supported.

Module-by-module validation is made possible by a validation library, analogous to a program library. Modules can be considered individually by distinguishing between their semantic specifications and their bodies. MAVEN imposes validation-order and revalidation requirements analogous to the compilation-order and recompilation requirements imposed by an Ada compiler. Revalidation requirements keep validation current with program revisions made during development or maintenance.

Besides a verifier and a validation-library manager, MAVEN includes reporting tools, tools for building and administering validation plans, tools to help write formal specifications, and tools to retrieve reusable software components based on their formal specifications. These tools provide benefits during all phases of the software life cycle.

Besides the many technical issues associated with formal verification of Ada programs, there is a broader software engineering issue: how formal verification can be applied most effectively throughout the life cycle of Ada software. This paper proposes a set of integrated tools and procedures that support the validation and verification of Ada software on a module-by-module basis. We expect that validation and verification will be accomplished partly through formal proof and partly through other means.

We call the proposed toolset the Modular Ada Validation Environment, or MAVEN. (The Yiddish word maven means an expert, authority, or master in some field.) MAVEN does not yet exist, nor have efforts begun to construct it. Rather, MAVEN is our vision of the context in which Ada formal verification should be applied.

1 Definition of a Validation Environment

When software engineers use the term "validation and verification," they usually do not have formal verification in mind. To avoid confusion, this paper will use the terms validation and verification in two distinct and precise senses:

Verification is the use of formal proof, checked by machine, to establish properties of a program's run-time behavior.

Validation is the process of increasing one's confidence in the reliability of a program. Unlike correctness, reliability is a matter of degree: A program containing even one error is incorrect, but a program containing only a few errors may highly reliable for its intended purpose. There are many methods for validating software, including formal proof, informal proof, code reviews, and testing.

Confusion may also arise from our use of the term environment. Ada Programming Support Environments (APSE's) already exist, and have functions that overlap those we propose for a validation environment. We do not envision MAVEN as a full APSE or as a tool set independent of an APSE. Rather, we view MAVEN as an integrated tool set embedded within an APSE. It can be thought of as a "subenvironment." Many APSE tools, including an Ada compiler, may be used both for validation and for other purposes.

2 Underlying Requirements

Our vision of MAVEN is based on several requirements that we have identified for the validation of Ada programs. These requirements are based on the recognition that Ada programs for mission-critical applications are large, that skilled software engineers are in short supply, that the construction of a verifier is an expensive undertaking, and that the use of a verifier may be time consuming. Our requirements are as follows:

1. Formal proofs should not be based on the behavior of a particular implementation.
2. It should be possible to validate a large program module by module.
3. For typical mission-critical applications, verification will have to be integrated with other forms of validation.
4. It should be easy to request the proof of certain critical properties which, while they do not imply correctness of a module, significantly raise our confidence in its reliability.

The following subsections describe these requirements in greater detail.

2.1 Implementation-Independent Verification

Verification is based on proof rules formalizing the behavior of an Ada program. If the proof rules are based only on the rules that appear in the Ada Language Reference Manual and apply to all implementations, then all proofs will be implementation-independent. Such proof rules are much harder to write, and may be more expensive to use during verification, than proof rules based on the behavior of a particular implementation. Nonetheless, it would be a serious mistake to build a verifier based on the behavior of a particular implementation.

2.1.1 Difficulty of Writing Implementation-Independent Proof Rules

Because the rules of the Ada language provide great latitude for implementation variations, it is difficult to write proof rules that, on the one hand, are consistent with any legal implementation of the language and, on the other hand, are strong enough to provide meaningful information. For example, a perverse but legal Ada implementation could raise `Storage_Error` upon activation of the main program, making it impossible to compute anything. Implementation-independent proof rules must account for this possibility, yet allow meaningful deductions to be drawn about how a program will behave under more reasonable implementations. Other allowable implementation variations include the order of subexpression evaluation, the

disregarding of arithmetic overflow, the use of internal representations that avoid overflow, and the application of optimizations that change the apparent site of an exception.

There are a number of ways to ameliorate the difficulties inherent in writing implementation-independent proof rules for a language with implementation variations:

- The proof rules may apply to a subset of the Ada language that excludes features with no meaningful implementation-independent semantics, such as address clauses and unchecked conversion.
- In some areas, such as order of subexpression evaluation and choice parameter passing mechanism, exhaustive proof rules can be written that cover all possible implementation variations. Such rules have the potential to transform postcondition formulas into exponentially long precondition formulas. In practice, however, the combinatorial explosion can be avoided by simplifications that apply whenever the programmer follows reasonable style guidelines like avoiding expressions with side-effects and disregarding the values left in actual parameters after a subprogram call propagates an exception. (Such guidelines are appropriate in most Ada software and virtually mandatory for portable software.)
- The proof rules can be selectively ignorant. Proof rules might ignore certain aspects of a computation that vary from implementation to implementation rather than trying to reflect all possibilities. For example, the state of variables updated in a frame is uncertain when control reaches a handler in that frame for some predefined exception, because optimizations may have moved the exception-raising operation before or after the variable-updating operation. Rather than trying to account for all possibilities, the proof rule might simply regard the value of the variable as unknown inside the handler.

2.1.2 Drawbacks of Implementation-Dependent Proof Rules

There are three reasons not to build an implementation-dependent verifier. First, the details of an individual implementation's behavior are not constrained by a universally accepted standard like the Ada Language Reference Manual. Second, it is not cost-effective to construct an implementation-independent verifier. Third, an implementation-dependent verifier can verify that erroneous programs (in the narrow sense of Ada Language Reference Manual section 1.6) and programs with incorrect order dependencies are correct, but cannot prove that a program is portable. Let us examine each of these problems more closely.

The language variant verified by an implementation-dependent verifier is not definitively specified. In reasoning about the behavior of a program, such a verifier could use characteristics like the dynamic storage allocation strategy and the size of the internal registers used for intermediate arithmetic results. However, these characteristics are not properly part of a compiler's interface, but part of its internal structure. A compiler

writer is free to change these characteristics from one version of the compiler to another, invalidating the verifier's proof rules.

A verifier that reasons about the behavior of a particular compiler's object code is not cost-effective because it can only be used in conjunction with that compiler. Anyone wishing to verify programs translated by another compiler must build his own verifier. A single implementation-independent compiler can be used in conjunction with all Ada compilers.

The rules of the Ada language forbid programmers to rely on certain characteristics of individual implementations, even though such rules are practically unenforceable. Nonetheless, an implementation-dependent verifier would base its reasoning on just such characteristics. This would encourage departure from the intended use of the Ada language and discourage portable programming. In contrast, a verifier based on implementation-independent language rules automatically proves portability of properties, because only deductions valid for all implementations are used in a proof.

2.1.3A Compromise

Given the technical difficulties presented by implementation-independent proof rules and the inappropriateness of proof rules based on a particular implementation, a compromise may be appropriate. That would be to base proof rules on a large class of implementations that conform to widely agreed-upon restrictions. In particular, substantially simpler proof rules might be obtainable by standardizing a definition of natural Ada semantics. This definition would be consistent with the definition of Ada semantics in the Ada Language Reference Manual, but it would constrain implementations more tightly.

The natural semantics would not specify characteristics like order of subexpression evaluation, which the rules of Ada explicitly forbid a programmer to depend on. Neither would they specify task interleaving or the selection of selective-wait alternatives, since it is intended that the programmer regard such matters as nondeterministic. However, optimizations that change the apparent behavior of a program could be forbidden in the natural semantics. (Optimizations would still be allowed in the presence of pragmas explicitly permitting them, but program units containing such pragmas would not be considered verifiable.) There might also be specific rules making it possible to predict, based on attributes like 'Storage_Size and 'Size and named numbers like System.Memory_Size, when Storage_Error will occur.

If a definition of natural semantics could be agreed upon and widely implemented, and if programmers could be educated to use optimization-enabling pragmas only in the few places they prove to be needed after performance metering, then construction of a verifier based on natural semantics would be reasonable. Like a verifier for implementation-independent Ada, a verifier for natural Ada would be based upon a recognized standard. The verifier could not be used with all compilers, but it could be used with the wide class of compilers implementing natural semantics. Properties proven by the verifier might not hold under all implementations,

but the properties would be portable within the wide class of natural implementations.

2.2 Module-by-Module Validation

The Ada language was designed to facilitate the construction of huge programs. A pervasive theme in the design of the language is the division of a program into units that can be understood individually yet checked for consistency with each other.

There are compelling practical considerations in support of modular validation. First, the amount of time required to validate a large system all at once would be prohibitive. Second, the modular approach allows one unit of a program to be changed and revalidated without revalidating the rest of the program. This is especially important during program maintenance.

The validation of an individual module requires independent specification of that module's desired behavior. Validation of the module establishes to a high degree of confidence that the module meets this specification. Similar specifications must be available for other modules so that the validation can proceed without considering the contents of the other modules.

A fundamental theme in Ada software engineering is the distinction between interface and implementation. MAVEN carries this distinction forward to software validation. A good specification will describe everything about a module's behavior that is of concern to other parts of a program. Once a module M has been validated with respect to its specification, other modules using M can be validated by considering M's specification alone, and not M's implementation.

2.3 Integration of Multiple Validation Approaches

Another reason for validating programs module-by-module is so that different modules can be validated in different ways. There are many software unit validation methods, all of which have been used successfully in the past. These include:

- formal proof generated with machine assistance and checked by machine
- informal proof carried out by hand
- code walkthroughs
- unit testing
- acceptance of a software component as trustworthy, based on experience using the same component in a previous system

It is not necessary for a project to choose one of these validation methods for use throughout a program. Given the right framework, different methods

can be combined in an effective symbiotic relationship to ensure the quality of a system.

While formal verification is the most effective means of ensuring consistency between a program and its specifications, it has limitations. These include the problem of validating that the specifications themselves specify what the customer wants; and the cost -- in both machine time and the time of skilled personnel -- of developing and checking the proof. The manufacture of software, like any manufacturing process, entails a tradeoff between cost and level of quality assurance. In some programs there are modules for which any form of validation less powerful than formal proof would be socially irresponsible. Sometimes the same program also contains many modules for which formal proof would be a wasteful misallocation of resources.

Furthermore, there may be some modules that cannot be verified because they use features of the language for which there are no proof rules. Features may be excluded from the "verifiable subset" of Ada even if there are occasional legitimate uses for such features. Such legitimate uses can be isolated in modules that are validated by some means other than formal proof. In particular, low-level features of the Ada language are inherently machine dependent and thus not characterized by proof rules. Low-level features can be isolated in interface modules, allowing the rest of a system to be validated by formal proof.

Many factors combine to determine the most appropriate form of validation of a module. The cost of formal proof must be compared with the possible impact of an error in the module. Low-level, target-dependent interface modules might best be validated by informal proof. For certain hard-to-specify modules, for example a graphics display builder whose desired output is specified pictorially, testing might be not only the cheapest, but also the most reliable form of validation. For modules that are not particularly critical, and for which test drivers would be difficult to write, code walkthroughs might be most appropriate. Software might simply be trusted (until integration testing) if it has been extracted from a working system in which it has functioned reliably.

To ensure complete coverage, different forms of validation cannot be combined haphazardly. There must be a unifying discipline. One of the functions envisioned for MAVEN is to provide such a discipline.

2.4 Specification of Critical Properties

A formal proof is sometimes incorrectly portrayed as giving absolute assurance of a program's "correctness." In fact, all that can ever be proven about a program is that it is consistent with its formal specifications. If the formal specifications do not correctly and completely reflect the program's intended behavior, then a program proven consistent with those specifications may not behave as intended. Unfortunately, the translation of informal requirements to formal specifications is itself a complex and error-prone process. In particular, it is easy to omit part of the requirements.

Still, it can be quite useful to prove particular properties of a program, even if these properties do not constitute a complete definition of correctness. In fact, we believe that the most promising use of verification in an industrial environment is not to prove that a program will behave correctly, but to prove that a module has certain important properties indicative of correct behavior. A property is a good indicator of an Ada program unit's quality if the attempt to prove the property is likely to uncover many of the faults in the unit.

MAVEN supports the proof of a spectrum of properties that are good indicators of a module's quality. This spectrum ranges from properties that are difficult to specify but provide strong assurances about a module's behavior to properties that are easy to specify but provide less comprehensive assurances. The properties actually proven about a module will depend on the individual module and the sophistication of the user. Some modules may have simple, well-defined specifications while others may not. Some modules may perform functions especially critical to the safety of a system while others may not.

The properties we have identified, ranging roughly from most difficult to specify to least difficult, are as follows:

- the validity of arbitrary logical formulas
- the correctness of a package implementing an abstract data type, with respect to a set of axioms defining the type in terms of its operations
- the correct instantiation of a generic unit
- numeric properties (the range and precision of results)
- the absence of unanticipated exceptions
- the absence of erroneous execution (adherence to potentially unenforced rules of the Ada language)

In time, other properties may be added to this list.

The most easily specified properties require the user of MAVEN to provide little or no information (perhaps a list of exceptions that a module may legitimately propagate). MAVEN automatically generates the logical formulas that must be proven to establish these properties. This makes formal proof, albeit formal proof of properties weaker than correctness, accessible to a larger number of users.

3 Validation Libraries

Module-by-module validation of a large program can be achieved in the same way as module-by-module compilation. Compilation of an Ada program unit consists not only of code generation, but also consistency checking. A unit's syntactic specification is compiled before either the unit's body or any external uses of the unit. This compilation puts information about the syntactic specification into a program library. Later, when either the unit's body or an external use of the unit is compiled, this information is retrieved from the program library and used for compile-time consistency checks. If a unit's body and an external use of the unit are both consistent with the unit's syntactic specification, then they are consistent with each other.

The consistency checks that occur during compilation are limited to the information found in a unit's syntactic specification, such as the number, types, and modes of subprogram parameters. Except for this limitation, however, they are analogous to the checks that occur during unit validation. Just as a unit has a syntactic specification that is checked during compilation, it has a semantic specification that is checked during validation. Just as syntactic specifications are recorded in a program library, semantic specifications are recorded in a MAVEN validation library.

3.1 Semantic Specifications

Different kinds of Ada program units have different kinds of semantic specifications. The semantic specification for a subprogram consists of a set of precondition/postcondition pairs, one for normal termination and one for each exception that the subprogram may raise. The semantic specification of a package consists of the semantic specifications of the subprograms provided by the package. Each of these subprograms may be viewed as having an additional, implicit parameter representing the abstract package state. The package's semantic specification may describe how calls on a package's procedures affect the abstract state of the package and how the abstract state of the package affects the results of the package's procedures and functions. (A package may have many internal states corresponding to the same abstract state.) MAVEN's external view of a task is similar to its external view of a package. A task has an abstract state that is passed as an implicit in out parameter to each entry call. An entry has a logical specification like that of a procedure, consisting of a set of precondition/postcondition pairs. A logical specification of a task type consists of the logical specifications of its entries.

Semantic specifications are textually embedded in syntactic specifications in the form of structured comments like those found in Anna [1]. This unifies the notions of syntactic and semantic specifications. When MAVEN is directed to compile a specification, it invokes the Ada

compiler to place the syntactic specification in the program library. If no compile-time errors are found, the semantic specification is then extracted from the structured comments and added to the validation library. If a specification has already been compiled and only the semantic specification has changed, the user may direct MAVEN to skip the first step when "recompiling" a specification. The effect of such a recompilation is to revise the semantic specification of a program unit but not its syntactic specification. The validation library is updated but leave the program library is left unchanged.

3.2 Validation Order

To facilitate compile-time consistency checks, the Ada language restricts the order in which units may be compiled. MAVEN imposes analogous restrictions on the order of validation. Specifically, a module's semantic specification must be entered into the validation library before the implementation or any use of the module is validated. Then the implementation and each use of the module may be validated in any order. Validation of the implementation establishes that the body fulfills the semantic specification. Validation of a use of the module involves assuming, while validating the using module, that the semantic specification is correctly implemented. This assumption is permitted as soon as the semantic specification is entered into the validation library, even before the body has been demonstrated to fulfill the semantic specification. (This is analogous to the compilation of a subprogram call after the subprogram declaration has been compiled but before the subprogram body has been compiled.) It implies that validation of one unit can proceed considering only the specifications of the units it invokes, without considering their bodies. This is the essence of module-by-module validation.

Consider, for example, the validation of a subprogram. First the subprogram's precondition/postcondition pairs are entered into the validation library. Units that call the subprogram may then be validated. In the case of a formal proof, the precondition/postcondition pairs may be assumed true in verifying the caller. In the case of testing, the precondition/postcondition pairs may be used to construct an appropriate stub.

Similarly, any time after the precondition/postcondition pairs are entered into the validation library, the subprogram body may be validated. The method of validation for the body is independent of the validation methods used for the calling units. In the case of formal proof, it is necessary to show that, when invoked with a precondition true, the subprogram returns with the corresponding postcondition true. In the case of testing, the precondition/postcondition pair may be used to generate test drivers or test data. Of course the validity of a proof about the caller depends on the validity of the specifications for the subprogram, which may be validated by some less rigorous means. Nonetheless, the proof provides strong assurances about the logic of the caller, if not about the behavior of the caller and the subprogram in combination.

Some program units may be validated by fiat. That is, after a code walkthrough or simply on the basis of trust, a unit may simply be decreed to

be "validated." This still must be done explicitly, by a request to MAVEN, and the usual validation order rules must be obeyed. In particular, a unit may not be decreed to be validated before the specifications it is meant to fulfill have been entered into the program library.

3.3 Revalidation Order

Just as the Ada language restricts compilation order, it imposes recompilation requirements to ensure that consistency checks have always been performed on the latest version of a program. If a syntactic specification is recompiled, all consistency checks based on the old syntactic specification are rendered invalid. The corresponding body and all uses of the unit must then be recompiled so that the consistency checks may be repeated with respect to the new syntactic specification.

MAVEN imposes analogous revalidation requirements. If a module's semantic specification is changed, both the implementation and all uses of the module must be revalidated if they have already been validated. This is relevant during program development and program maintenance.

In program development, the following scenario may take place:

1. The semantic specification of package A is entered into the validation library.
2. Subprogram B, which uses package A, is validated with respect to this semantic specification.
3. Attempts to validate the body of A are not successful. Further examination reveals that the validation process is not at fault: A's package body is truly inconsistent with A's semantic specification.

At this point there are two possibilities. First, the validation failure may have revealed an error in the package body. Once this error is corrected, A's body may be successfully validated. Second, the validation failure may have revealed an error in A's semantic specification. The package-body writer may have exploited some valid assumption that was inadvertently omitted from the semantic specification, for example. The solution here is to correct A's semantic specification, perhaps by strengthening its preconditions. This process serves to keep documentation current and complete, since the revised semantic specification now reflects heretofore implicit assumptions. However, B must now be revalidated to ensure that B establishes the strengthened preconditions before invoking A. Just as recompilation of one Ada unit can lead to the recompilation of many other units, so a change to one unit's semantic specification can lead to revalidation of many other units. In this case, if revalidation of B fails, B's semantic specification may have to be revised. Then the body and users of B will have to be revalidated, and so forth.

In program maintenance, revalidation requirements indicate which parts of a large program are potentially affected by a change. This can reduce or eliminate the "ripple effect" typically resulting from a change to a working

program. A change to enhance performance might be accomplished by changing unit bodies only, and leaving each unit's semantic specification intact. Then it would only be necessary to revalidate the revised bodies. A change to enhance functionality might require a change to a unit's semantic interface, requiring revalidation of that unit's body and each of its uses. All possible implications of the change will be flushed out by the ensuing round of revalidations, assuming the revalidation is sufficiently thorough. (If the revalidation is by unit testing, this process amounts to regression testing. Rather than blindly repeating all tests, however, we use validation dependency relationships to identify the tests that might possibly have been affected by the change.)

A unit validated by fiat is subject to the same revalidation requirements as any other unit, even if revalidation consists only of reissuing the decree by which the unit was originally validated. This encourages software engineers to consider whether the original decree is still valid given the new specifications. For example, it may be discovered that an off-the-shelf package originally thought to be applicable to the current application is inappropriate given the revised specifications.

3.4 Other Information in the Validation Library

A validation library contains information besides the semantic specifications of program units. A validation plan can be entered into the library in advance, stipulating how a unit will be validated once it is written. The validation library also records which units have been validated, and according to which validation plans.

Each module may have its own validation plan. The plan specifies the validation method applied to the unit (testing or formal proof, for example) and the details of the validation criteria (which files contain the test driver or test data, algorithms for evaluating test results, or which properties are to be proven, for example). A validation plan may specify several rounds of validation, all of which must succeed for the unit to be considered validated. For example, a plan may call for testing to find and eliminate obvious errors, followed by formal proof to ensure the absence of more subtle errors. No one round of validation need provide complete coverage of the unit's semantic specification. Some parts of a unit's semantic specification may be proven valid, some validated by testing, and some simply assumed to be valid, for example.

Besides allowing MAVEN to enforce validation and revalidation order dependencies, the data kept in the validation library allows MAVEN tools to generate reports on the progress of system validation to date. The reports indicate which units have been validated and how rigorously. During development, validation of units can be tracked and compared with schedules. When an error arises, information about the validation methods applied to each unit and the properties validated for each unit can help pinpoint suspect modules. The revalidation implications of a proposed change can quickly be estimated.

4 Other Components of a Validation Environment

The appropriate home for an Ada verifier is in a validation environment like MAVEN, but a verifier is only one of the tools that such an environment should provide. We have already mentioned the need for a validation library. This implies the need for library management tools, including the report-generation tools discussed above. Other tools can assist in the writing of specifications, the retrieval of reusable software from a large catalogue, and the execution and analysis of tests.

4.1 The Specification-Writer's Assistant

Formal specifications are at the heart of MAVEN, but they are difficult for the typical software engineer to write. Therefore MAVEN must supply tools to help the software engineer express his intent. These tools are collectively called the specification-writer's assistant.

One component of the specification-writer's assistant is a knowledge-based tool that will construct formal specifications based on a dialogue with the user. Libraries of high-level specification primitives will be employed. These might include infinite sets, primitives used in data security specifications, and so forth.

The specification-writer's assistant also includes an interpreter for a logic programming language, similar to PROLOG but providing the higher level of data abstraction found in the Ada language. This tool can be used for rapid prototyping, to test specifications as they are written. Successfully tested specifications are then translated automatically into the MAVEN specification language. (We assume the MAVEN specification language will be too rich to be implemented directly.) Such an approach is suggested by Doyle [2] as a practical way to apply artificial intelligence techniques in software engineering.

4.2 Reusable-Software Retrieval Tool

The Ada language is meant to encourage the reuse of general-purpose software components. This approach can only have a significant impact on software development costs if there is a large corpus of general-purpose software available for reuse; but such a large corpus presents an awesome information-retrieval problem. While software retrieval is not usually thought of as a validation problem, Platek [3] has noted that formal specifications and verification can form the basis of a retrieval tool.

In addition to a validation library, MAVEN might include a catalogue of general-purpose, reusable software components, all of which have been formally specified. Given the semantic specification of a module required in

the design, a MAVEN tool would search the catalogue for reusable components that have matching specifications. Roughly speaking, the specifications will be considered to match if two conditions hold:

1. The preconditions given in the design imply the corresponding preconditions of the reusable component.
2. The postconditions of the reusable component imply the corresponding postconditions of the design.

Both these conditions would be verified.

Such a tool is currently beyond the state of the art. A practical tool will require sophisticated pattern matching, able to look past differences in parameter order, additional functions provided by the reusable component, and so forth. In some cases, the tool will have to recognize that instantiation of a generic unit will produce an instance with matching specifications.

4.3 Testing Tools

Because testing is the most frequently used validation method, MAVEN contains tools specifically supporting testing. These include tools to generate subprogram stubs, tools to generate test drivers, tools to generate test data, and tools to analyze test results. All of these tools can base their outputs at least in part on the semantic specifications found in the validation library. For embedded applications, there should be software simulation tools and tools providing interfaces with hardware mockups.

A related tool would administer tests automatically, based on the validation plans found in the validation library. Such a tool could also revalidate those units validated entirely by testing, whenever revalidation is required. In essence, this automates regression testing.

5 MAVEN and the Software Life Cycle

MAVEN tools are primarily concerned with unit validation. This can lead to the impression that the benefits of MAVEN are primarily reaped during the unit validation stage of the life cycle. In fact, the use of MAVEN imposes a discipline on software development and provides benefits throughout the software life cycle. This section walks through a typical waterfall model of the life cycle and describes the impact of MAVEN on each stage.

5.1 Requirements Analysis

The MAVEN specification-writer's assistant supports the formal expression of requirements. Requirements can be entered into a new MAVEN validation library as the semantic specifications of the main program and of

tasks declared in library packages. These formally stated requirements can be checked for consistency using a verifier. They may later become the basis for design verification and code verification. An integration-testing plan may be derived from the formal requirements and stored in the validation library until software integration time.

5.2 Design

During high-level design, the modular decomposition of a system is determined and the specifications of each module are written. Algorithms for top-level modules may also be written. MAVEN can play four roles at this stage -- design documentation, recording of unit validation plans, software-component retrieval, and design verification.

Design documentation consists of entering the semantic specifications for each design module into the validation library. The specification-writer's assistant again comes in handy here. The semantic specifications entered at this stage become the basis for later verification of module bodies.

Unit validation plans were discussed earlier. The appropriate time to formulate them is just after unit semantic specifications have been identified. Thus unit validation plans are entered into the validation library during design for retrieval during unit development.

One of the responsibilities of an Ada designer is to look, before specifying a new module to be written, for existing software that can be incorporated in a design. As noted earlier, formal specifications might provide the basis for software automated software retrieval. If MAVEN's catalogue of reusable software components contains only verified components, then retrieval of a given component will constitute proof that the component satisfies the specifications in the design. No further validation of that component will be necessary.

Because of its high level of abstraction, the Ada language is frequently used as a program design language. Indeed, executable Ada code would be considered a design-level specification of an algorithm if older implementation languages were to be used. Thus the top-level algorithms of a high-level design are expressed in executable Ada code that can be verified in the same way as lower level modules. At this point, semantic specifications have been written for the main system modules (the main program and tasks declared in library packages) and the high-level modules directly invoked by the main system modules. Using only these specifications, it can be proven that the top-level algorithms correctly implement the system specifications.

5.3 Unit Development

There is not a clear dividing line between design validation and unit validation. The same techniques applied to the top-level modules during design validation are applied to lower-level modules during unit validation.

The unit validation plan placed in the validation library during system design is retrieved and applied. A round of validation is repeated until it is successful, and then the next round specified in the validation plan is begun. The validation plan is restarted from the first round any time a change is made to the unit, its semantic specification, or the semantic specifications of the modules that the unit invokes.

As noted earlier in the discussion of validation order, validation can uncover implicit assumptions that underlie the correct functioning of a module. This is particularly so when validation is by formal verification. Such assumptions must be added to a module's semantic specifications if the module is to be verified. Thus the validation process contributes to the development of complete and up-to-date specifications.

5.4 Integration Testing

The main impact of MAVEN on integration testing will be a drastic reduction in integration problems. The Ada compiler will already have checked all units for syntactic consistency with each other. MAVEN will already have checked all units for consistency with their own semantic specifications and the semantic specifications of the modules they invoke. The few integration problems that remain will arise from incomplete module specifications (for example, specifications that address functional requirements but not performance requirements) and insufficiently rigorous unit validation (for example, use of code walkthroughs as the sole means of validation or the use of tests that do not provide adequate coverage).

5.5 Maintenance

MAVEN will reduce the costs and risks of program maintenance. Both the data MAVEN collects during program development and the discipline MAVEN imposes on program modification will help confine the "ripple effect" of a change. MAVEN will also keep documentation up to date after changes have been made.

The most frequent problem associated with program maintenance is a change that violates an implicit assumption upon which a different part of the program depends. This problem is less likely to arise when using MAVEN for two reasons. First, the validation process applied during program development has served to make implicit assumptions explicit. The documentation will warn the maintenance programmer right from the start that certain changes must be disallowed unless further changes are made in other modules. Second, if the semantic specification of a module is changed, MAVEN will enforce the revalidation of all modules that may be affected by the change. The revalidation dependencies alone clarify the potential impact of a contemplated change. The actual revalidation, which may follow the original unit validation plan created during the initial design, leads the maintenance programmer to discover which potential impacts are truly significant, to revise the affected modules, and to validate the revisions. If the revised modules can themselves affect other modules, revalidation of these other modules will also be required. If sufficiently rigorous,

revalidation anticipates and averts all possible ripple effects.

MAVEN keeps documentation current during program maintenance in the same way that it does so during initial development. Every time a unit's semantic specification changes, MAVEN records the fact. This makes the next round of maintenance easier.

6 Conclusions

A verifier may be constructed as a research tool to explore the technological frontiers of formal reasoning about programs; or as a practical tool meant to be used in the validation of production software. Both goals are worthy. Before undertaking the specification, design, and implementation of a practical tool, however, it is important to consider the context in which the tool will be used. We have described our vision of a Modular Ada Validation Environment, MAVEN, to propose a context in which formal verification can fit into the industrial development of Ada software.

Our vision of MAVEN is based on certain principles. First, formal proof should be based on implementation-independent proof rules, since such rules correspond to a generally accepted standard, are beneficial to users of all compilers, and can be used to prove portability. Second, large Ada programs should be validated module by module. Like module-by-module compilation with static consistency checking, module-by-module validation of run-time behavior is based on the distinction between a module's specification and its implementation, and the recording of module specifications in a library. Third, formal proof is only one form of program validation, and proof of correctness is only one kind of formal proof. Effective industrial use of formal verification requires that it be one weapon in a large arsenal of validation methods.

While proof of correctness is unquestionably the most rigorous and effective form of validation, there are contexts in which it is inappropriate. Specification of correctness may be too difficult or error-prone, in which case there may be weaker properties that it is more appropriate to prove. A module may use implementation-dependent features, making formal proof based on implementation-independent proof rules impossible. Validation is meant to increase confidence in the suitability of a module for its intended purpose; for some modules, greater confidence may be obtained by running test cases than by proving fulfillment of some postcondition. Some modules may not be critical enough to justify the cost of rigorous validation.

MAVEN offers software engineers a continuum of more and less rigorous validation methods. This continuum makes a wider variety of validation methods available to a larger group and applicable to a greater number of modules. MAVEN provides a unifying framework in which different validation methods may be applied to the same program. By exposing software engineers to more rigorous methods than those they may be familiar with, MAVEN

encourages learning and promotes wider use of formal methods in the situations where they are appropriate.

MAVEN includes components that are at and beyond the state of the art. We do not propose that construction of MAVEN in its entirety should start today. Rather, MAVEN can serve as framework for the specification, design, and construction of individual tools, such as a verifier. If such tools are viewed as eventual MAVEN components and if the MAVEN philosophy is kept in mind when the tools are specified, then MAVEN can be assembled over a number of years from independently developed components.

REFERENCES

-
1. Luckham, David C., von Henke, Friedrich W., Krieg-Brueckner, Bernd, and Owe, Olaf. Anna, A Language for Annotating Ada Programs: Preliminary Reference Manual. Technical Report 84-261, Stanford Computer Systems Laboratory, July 1984
 2. Doyle, Jon. Expert systems and the "myth" of symbolic reasoning. IEEE Transactions on Software Engineering SE-11, No. 11 (November 1985), 1386-1390
 3. Platek, Richard. Formal specification. Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada, Alexandria, Virginia, March 1985, paper C

MAVEN: THE MODULAR ADA* VERIFICATION ENVIRONMENT

Norman H. Cohen

SofTech, Inc.

**One Sentry Parkway, Suite 6000
Blue Bell, Pennsylvania 19422-2310**

NCohen@Ada20

***Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).**

vg1518wo 5/12/86

OUTLINE

- DEFINITIONS

- UNDERLYING REQUIREMENTS

- VALIDATION LIBRARIES

- MAVEN TOOLS

- LIFE-CYCLE IMPACT OF MAVEN

- CONCLUSIONS

DEFINITIONS

- VERIFICATION:** USE OF FORMAL PROOF, CHECKED BY MACHINE, TO ESTABLISH PROPERTIES OF A PROGRAM'S RUN-TIME BEHAVIOR
- VALIDATION:** THE PROCESS OF INCREASING ONE'S CONFIDENCE IN THE RELIABILITY OF A PROGRAM (E.G. BY FORMAL OR INFORMAL PROOF, CODE REVIEWS, OR TESTING)
- VALIDATION ENVIRONMENT:** AN INTEGRATED TOOLSET TO SUPPORT VALIDATION, POSSIBLY A SUBENVIRONMENT OF AN APSE
- TOOLSET MAY INCLUDE STANDARD TOOLS ALSO USED FOR OTHER PURPOSES

MAVEN

- **THE "MODULAR ADA VALIDATION ENVIRONMENT"**
- **A VISION OF WHAT A VALIDATION ENVIRONMENT OUGHT TO INCLUDE**
- **NO CURRENT PLANS FOR ITS CONSTRUCTION**

OUTLINE

- DEFINITIONS
- UNDERLYING REQUIREMENTS
- VALIDATION LIBRARIES
- MAVEN TOOLS
- LIFE-CYCLE IMPACT OF MAVEN
- CONCLUSIONS

UNDERLYING REQUIREMENTS

- **FORMAL PROOFS SHOULD NOT BE BASED ON THE BEHAVIOR OF A PARTICULAR IMPLEMENTATION**
- **IT SHOULD BE POSSIBLE TO VALIDATE A LARGE PROGRAM MODULE BY MODULE**
- **FOR TYPICAL MISSION-CRITICAL APPLICATIONS, VERIFICATION MUST BE INTEGRATED WITH OTHER FORMS OF VALIDATION**

MODULE-BY-MODULE VALIDATION

- **VALIDATION OF INDIVIDUAL MODULES REQUIRES SPECIFICATION OF EACH MODULE'S DESIRED BEHAVIOR**
- **SPECIFICATION SHOULD DESCRIBE EVERYTHING ABOUT THE MODULE THAT IS RELEVANT IN VALIDATING OTHER MODULES**
- **MODULE IMPLEMENTATION VALIDATED WITH RESPECT TO ITS SPECIFICATION**
- **VALIDATION MAY REFER TO SPECIFICATIONS OF OTHER MODULES, BUT NOT TO THEIR IMPLEMENTATIONS**

INTEGRATION OF MULTIPLE VALIDATION APPROACHES

- **ALTERNATIVE APPROACHES CAN BE APPLIED TO DIFFERENT
MODULES TO VALIDATE A SYSTEM:**
- **FORMAL PROOF, GENERATED WITH MACHINE ASSISTANCE
AND CHECKED BY MACHINE**
- **INFORMAL PROOF CARRIED OUT BY HAND**
- **CODE WALKTHROUGHS**
- **UNIT TESTING**
- **ACCEPTANCE OF A UNIT AS TRUSTWORTHY, BASED ON
SUCCESSFUL USE OF THAT UNIT IN ANOTHER SYSTEM**
- **A UNIFYING DISCIPLINE IS REQUIRED FOR INTEGRATION**

softech

INTEGRATION OF MULTIPLE VALIDATION APPROACHES (CONT.)

- **CHOICE OF VALIDATION METHODS DEPENDS ON:**
 - **COST OF APPLYING THE METHOD**
 - **IMPACT OF AN ERROR IN THE GIVEN MODULE**
 - **DEGREE OF IMPLEMENTATION DEPENDENCE**
 - **EASE OF FORMAL SPECIFICATION**
 - **NEED TO OBSERVE OUTPUT (E.G. GRAPHICS)**
 - **EASE OF CONSTRUCTING TEST DRIVERS**
 - **PREVIOUS EXPERIENCE WITH THE UNIT**

INTEGRATION OF MULTIPLE VALIDATION APPROACHES (CONT.)

- **LIMITATIONS OF FORMAL VERIFICATION:**
 - **VALIDATING THAT SPECIFICATIONS DESCRIBE THE CUSTOMER'S REAL NEEDS**
 - **TRADEOFF BETWEEN COST OF QUALITY AND ACCEPTABILITY OF IMPERFECTIONS**
 - **LEGITIMATE USES FOR UNVERIFIABLE ADA CONSTRUCTS**

OUTLINE

- DEFINITIONS
- UNDERLYING REQUIREMENTS
- VALIDATION LIBRARIES
- MAVEN TOOLS
- LIFE-CYCLE IMPACT OF MAVEN
- CONCLUSIONS

VALIDATION LIBRARY

- A VALIDATION LIBRARY IS TO UNIT VALIDATION WHAT A PROGRAM LIBRARY IS TO UNIT COMPILATION
- PROGRAM LIBRARIES CONTAIN SYNTACTIC SPECIFICATIONS USED IN COMPILE-TIME CHECKS; VALIDATION LIBRARIES CONTAIN SEMANTIC SPECIFICATIONS USED IN VALIDATING RUN-TIME BEHAVIOR
- SEMANTIC SPECIFICATIONS ARE EMBEDDED AS "STRUCTURED COMMENTS" IN SYNTACTIC SPECIFICATIONS.
 - COMPILING A SYNTACTIC SPECIFICATION ADDS THE EMBEDDED SEMANTIC SPECIFICATION TO THE VALIDATION LIBRARY
 - IF THE SEMANTIC SPECIFICATION IS CHANGED WHILE LEAVING THE SYNTACTIC SPECIFICATION INTACT, THE REVISED SEMANTIC SPECIFICATION CAN BE ADDED TO THE VALIDATION LIBRARY DIRECTLY

SEMANTIC SPECIFICATIONS

- FOR A SUBPROGRAM:
 - A SET OF PRECONDITION/POSTCONDITION PAIRS, ONE FOR NORMAL TERMINATION AND ONE FOR EACH EXCEPTION THAT MAY BE PROPAGATED.
- FOR A PACKAGE:
 - SEMANTIC SPECIFICATIONS OF THE SUBPROGRAMS PROVIDED BY THE PACKAGE
 - EACH SUBPROGRAM HAS AN EXTRA IMPLICIT PARAMETER CONSISTING OF THE "ABSTRACT PACKAGE STATE"
- FOR A TASK:
 - EACH ENTRY HAS A SET OF PRECONDITION/POSTCONDITION PAIRS, LIKE A PROCEDURE
 - THE TASK HAS AN ABSTRACT STATE, LIKE A PACKAGE

VALIDATION ORDER

- MAVEN IMPOSES VALIDATION-ORDER RESTRICTIONS ANALOGOUS TO ADA COMPILATION-ORDER RESTRICTIONS:
 - A MODULE'S SEMANTIC SPECIFICATION MUST BE ENTERED INTO THE VALIDATION LIBRARY FIRST
 - THEN THE MODULE'S IMPLEMENTATION AND THE MODULE'S CLIENTS MAY BE VALIDATED IN ANY ORDER
- VALIDATION OF AN IMPLEMENTATION:
 - ESTABLISH THAT THE BODY FULFILLS THE SEMANTIC SPECIFICATIONS
- VALIDATION OF A CLIENT:
 - ASSUME THE MODULE MEETS ITS SEMANTIC SPECIFICATIONS, EVEN IF THIS HAS NOT YET BEEN VALIDATED

REVALIDATION ORDER

- MAVEN IMPOSES REVALIDATION REQUIREMENTS ANALOGOUS TO ADA RECOMPILATION REQUIREMENTS
- A CHANGE TO A MODULE'S SEMANTIC SPECIFICATIONS MAY RENDER PREVIOUS VALIDATIONS MOOT
 - BODY MUST BE REVALIDATED
 - CLIENTS MUST BE REVALIDATED

SCENARIO 1: REVALIDATION DURING PROGRAM DEVELOPMENT

- SPECIFICATION OF PACKAGE A IS COMPILED
- SUBPROGRAM B, WHICH USES PACKAGE A, IS VALIDATED
- INCONSISTENCIES FOUND TRYING TO VALIDATE A'S BODY
- TWO POSSIBILITIES ARISE:
 - IF A'S BODY CONTAINS A PROGRAMMING ERROR, CORRECT IT AND TRY AGAIN TO VALIDATE THE BODY
 - IF PACKAGE A WAS INCOMPLETELY SPECIFIED (E.G., THE SPECIFICATION OMITTED AN IMPLICIT ASSUMPTION THAT THE BODY EXPLOITED):
 - CORRECT THE SPECIFICATION
 - TRY AGAIN TO VALIDATE A'S BODY
 - TRY TO REVALIDATE B
 - IF NECESSARY, REVISE B'S SEMANTIC SPECIFICATION AND REVALIDATE OTHER UNITS

SCENARIO 2: REVALIDATION DURING MAINTENANCE

- **A CHANGE TO ENHANCE PERFORMANCE MAY REQUIRE ONLY REVISION AND REVALIDATION OF UNIT BODIES**
- **A CHANGE TO ADD FUNCTIONALITY MAY REQUIRE REVISED SEMANTIC SPECIFICATIONS FOR SOME MODULES**
- **BODIES AND CLIENTS OF THESE MODULES MUST THEN BE REVALIDATED**
- **REVALIDATION ANTICIPATES AND MAY AVERT THE "RIPPLE EFFECT" THAT USUALLY FOLLOWS MAINTENANCE**
- **IF VALIDATION IS BY TESTING, REVALIDATION CONSTITUTES REGRESSION TESTING**

IMPLICATIONS OF REVALIDATION REQUIREMENTS

- **IMPLICIT ASSUMPTIONS ARE MADE EXPLICIT AND THE SYSTEM IS CHECKED FOR CONSISTENCY WITH THESE ASSUMPTIONS**
- **DOCUMENTATION (IN THE FORM OF SEMANTIC SPECIFICATIONS) IS KEPT CURRENT DURING PROGRAM DEVELOPMENT AND MAINTENANCE**
- **EVEN MODULES VALIDATED BY FIAT MUST BE REVALIDATED WHEN THEIR SEMANTIC SPECIFICATIONS CHANGE:**
 - **BEFORE REISSUING THE FIAT FOR AN OFF-THE-SHELF MODULE, CONSIDER WHETHER IT IS STILL APPLICABLE, GIVEN THE REVISED SPECIFICATIONS**

VALIDATION PLANS

- **VALIDATION LIBRARY CONTAINS A VALIDATION PLAN FOR EACH UNIT**
- **VALIDATION PLAN SPECIFIES:**
 - **VALIDATION METHODS (E.G., TESTING, FORMAL PROOF)**
 - **VALIDATION CRITERIA (E.G., TEST DATA, TEST RESULT EVALUATION CRITERIA, PROPERTIES TO BE PROVEN)**

VALIDATION PLANS (CONT.)

- **UNIT'S PLAN MAY SPECIFY SEVERAL VALIDATION "ROUNDS":**
 - **EXAMPLE: TESTING TO FIND AND ELIMINATE OBVIOUS ERRORS, FOLLOWED BY FORMAL PROOF TO ENSURE ABSENCE OF MORE SUBTLE ERRORS**
 - **VALIDATION ACHIEVED WHEN ALL ROUNDS SUCCEED**
 - **DIFFERENT ROUNDS MAY VALIDATE DIFFERENT PARTS OF A UNIT'S SEMANTIC SPECIFICATION**

OUTLINE

- DEFINITIONS
- UNDERLYING REQUIREMENTS
- VALIDATION LIBRARIES
- MAVEN TOOLS
- LIFE-CYCLE IMPACT OF MAVEN
- CONCLUSIONS

MAVEN TOOLS

- **VALIDATION-LIBRARY MANAGEMENT TOOLS**
- **VERIFICATION TOOLS (VCG, THEOREM PROVER)**
- **REPORT GENERATORS**
- **SPECIFICATION-WRITER'S ASSISTANT**
- **REUSABLE-SOFTWARE RETRIEVAL TOOL**
- **TESTING TOOLS**

SOFTech

VALIDATION STATUS REPORTS

- **WHICH UNITS HAVE BEEN VALIDATED**
- **VALIDATION METHODS APPLIED TO EACH UNIT**
- **PROJECTED REVALIDATION COSTS OF A CHANGE**
- **COMPARISONS OF ACTUAL VALIDATION PROGRESS AND DEVELOPMENT SCHEDULE**
- **HINTS ABOUT THE POSSIBLE CAUSE OF AN ERROR, BASED ON WHICH PROPERTIES WERE VALIDATED BY WHICH METHODS**

SPECIFICATION-WRITER'S ASSISTANT

- **FORMAL SPECIFICATIONS ARE ESSENTIAL TO MAVEN,
BUT ARE DIFFICULT TO WRITE**
- **COMPONENTS:**
 - **KNOWLEDGE-BASED TOOL TO CONSTRUCT FORMAL
SPECIFICATIONS BASED ON DIALOGUE WITH USER,
USING LIBRARIES OF HIGH-LEVEL SPECIFICATION
PRIMITIVES**
 - **INTERPRETER FOR A LOGIC-PROGRAMMING LANGUAGE
WITH ADA-LIKE DATA ABSTRACTION CAPABILITIES,
FOR RAPID PROTOTYPING OF SPECIFICATIONS**
- **MAVEN ALSO AUTOMATES SPECIFICATION OF CERTAIN
CRITICAL PROPERTIES OF ADA PROGRAMS
(E.G. ABSENCE OF UNANTICIPATED EXCEPTIONS)**

REUSABLE-SOFTWARE RETRIEVAL TOOL

- LARGE CORPUS OF REUSABLE SOFTWARE PRESENTS AN INFORMATION-RETRIEVAL PROBLEM
- MAVEN INCLUDES A CATALOGUE OF FORMALLY SPECIFIED REUSABLE SOFTWARE COMPONENTS
- GIVEN SEMANTIC SPECIFICATION OF A MODULE REQUIRED IN THE DESIGN, A MAVEN TOOL SEARCHES THE CATALOGUE FOR A COMPONENT WITH A "MATCHING" SPECIFICATION
 - PRECONDITIONS GIVEN IN THE DESIGN IMPLY PRECONDITIONS OF THE CATALOGUED COMPONENT
 - POSTCONDITIONS OF THE CATALOGUED COMPONENT IMPLY POSTCONDITIONS GIVEN IN THE DESIGN
- EXTREMELY DIFFICULT PATTERN-MATCHING PROBLEM
- ONCE FOUND, THE MATCH IS VERIFIED BY FORMAL PROOF

TESTING TOOLS

- TOOLS BASED ON SEMIAUTOMATIC ANALYSIS OF SEMANTIC SPECIFICATIONS:
 - STUB GENERATORS
 - TEST-DRIVER GENERATORS
 - TEST-DATA GENERATORS
 - TEST-RESULT ANALYZERS
- AUTOMATIC TEST-ADMINISTRATION TOOLS
 - TOOL TO EXECUTE TEST-BASED VALIDATION PLANS
 - TOOL TO FULFILL THOSE REVALIDATION REQUIREMENTS THAT CAN BE FULFILLED BY TESTING (AUTOMATION OF REGRESSION TESTING)
- TOOLS FOR TESTING EMBEDDED-COMPUTER SOFTWARE
 - SIMULATION TOOLS
 - HARDWARE INTERFACES

OUTLINE

- DEFINITIONS
- UNDERLYING REQUIREMENTS
- VALIDATION LIBRARIES
- MAVEN TOOLS
- LIFE-CYCLE IMPACT OF MAVEN
- CONCLUSIONS

MAVEN AFFECTS EACH STAGE OF THE LIFE CYCLE

- **REQUIREMENTS ANALYSIS**
- **DESIGN**
- **UNIT DEVELOPMENT**
- **INTEGRATION TESTING**
- **MAINTENANCE**

SOFTech

vg1518wo/27 5/12/86

REQUIREMENTS ANALYSIS

- SPECIFICATION-WRITER'S ASSISTANT SUPPORTS FORMAL EXPRESSION AND PROTOTYPING OF REQUIREMENTS
- REQUIREMENTS ENTERED INTO VALIDATION LIBRARY AS SEMANTIC SPECIFICATIONS
 - OF THE MAIN PROGRAM
 - OF TASKS WITHOUT MASTERS (TASKS DECLARED IN LIBRARY PACKAGES)
- CONSISTENCY OF FORMAL REQUIREMENTS CAN BE VERIFIED

DESIGN

- DESIGN DOCUMENTATION: SEMANTIC SPECIFICATION OF EACH DESIGN MODULE ENTERED INTO VALIDATION LIBRARY
- SEMANTIC SPECIFICATIONS USED TO RETRIEVE REUSABLE COMPONENTS FROM VALIDATION LIBRARY
- DESIGN VERIFICATION: TOP LEVEL ALGORITHMS WRITTEN AND VERIFIED IN TERMS OF SEMANTIC SPECIFICATIONS OF LOWER-LEVEL MODULES
- UNIT VALIDATION PLANS DETERMINED AND ENTERED INTO VALIDATION LIBRARY

UNIT DEVELOPMENT

- **DEVELOP MODULE BODIES**
- **DEVELOP LOW-LEVEL MODULE SPECIFICATIONS AND ENTER SEMANTIC SPECIFICATIONS INTO VALIDATION LIBRARY**
- **APPLY UNIT VALIDATION PLANS**
- **AS NECESSARY, REVISE SPECIFICATIONS AND REVALIDATE, MAINTAINING AN UP-TO-DATE AND CONSISTENT SET OF SEMANTIC SPECIFICATIONS IN THE VALIDATION LIBRARY**

INTEGRATION TESTING

- DRASTIC REDUCTION IN INTEGRATION PROBLEMS DUE TO STEPS TAKEN EARLIER
- COMPILER HAS ALREADY CHECKED FOR CONSISTENCY OF EACH MODULE WITH OTHER MODULES' SYNTACTIC SPECIFICATIONS
- MAVEN HAS ALREADY VALIDATED CONSISTENCY OF EACH MODULE WITH OTHER MODULES' SEMANTIC SPECIFICATIONS
- REMAINING INTEGRATION PROBLEMS ATTRIBUTABLE TO
 - INCOMPLETE MODULE SPECIFICATIONS (E.G. FUNCTIONAL SPECIFICATIONS ADDRESSED BUT PERFORMANCE SPECIFICATIONS IGNORED)
 - INSUFFICIENTLY RIGOROUS VALIDATION PLANS

MAINTENANCE

- **CHANGES LESS LIKELY TO VIOLATE IMPLICIT ASSUMPTIONS UPON WHICH OTHER MODULES DEPEND**
- **UNIT VALIDATIONS TEND TO MAKE IMPLICIT ASSUMPTIONS EXPLICIT**
- **REVALIDATION DEPENDENCIES WARN ABOUT THE POTENTIAL SCOPE OF IMPACT OF A CHANGE**
- **ACTUAL REVALIDATION ANTICIPATES AND AVERTS RIPPLE EFFECTS**
- **RECORDING OF REVISED SEMANTIC SPECIFICATIONS KEEPS DOCUMENTATION UP TO DATE FOR NEXT ROUND OF MAINTENANCE**

OUTLINE

- DEFINITIONS
- UNDERLYING REQUIREMENTS
- VALIDATION LIBRARIES
- MAVEN TOOLS
- LIFE-CYCLE IMPACT OF MAVEN

- CONCLUSIONS

SOFTech

BASIC MAVEN PRINCIPLES

- **MODULE-BY-MODULE VALIDATION**
- **SEMANTIC SPECIFICATION OF A MODULE**
- **VALIDATION LIBRARY**
- **INTEGRATION OF MULTIPLE VALIDATION METHODS**
- **VERIFICATION IS THE MOST RIGOROUS METHOD**
- **NOT THE MOST APPROPRIATE METHOD FOR EVERY MODULE**
- **MAVEN PROVIDES A UNIFYING FRAMEWORK FOR APPLYING DIFFERENT METHODS TO THE SAME PROGRAM**

MAVEN AND FORMAL METHODS

- **MAVEN PROVIDES A PRACTICAL ROLE FOR VERIFICATION IN THE INDUSTRIAL DEVELOPMENT OF ADA SOFTWARE**
- **MAVEN MAKES A CONTINUUM OF MORE AND LESS RIGOROUS VALIDATION METHODS AVAILABLE TO THE PRACTICING SOFTWARE ENGINEER**
- **EXPOSURE TO FORMAL METHODS ENCOURAGES LEARNING ABOUT THEM, MAY LEAD TO WIDER USE**

SOFTech

VG1518wo/37 5/12/86

WHERE DO WE GO FROM HERE?

- **DO NOT START BUILDING THE ENTIRE MAVEN ENVIRONMENT**
- **SOME COMPONENTS STILL BEYOND THE STATE OF THE ART**
- **TOTAL COST MAY BE TOO GREAT TO BE FUNDED FROM A SINGLE SOURCE**
- **USE MAVEN AS THE FRAMEWORK FOR THE SPECIFICATION, DESIGN AND CONSTRUCTION OF INDIVIDUAL TOOLS**
- **DESIGN A STANDARD VALIDATION LIBRARY INTERFACE**
- **INCLUDE MAVEN-COMPATIBILITY AS A REQUIREMENT FOR ADA VALIDATION TOOLS**
- **ASSEMBLE MAVEN OVER MANY YEARS FROM INDEPENDENTLY DEVELOPED COMPONENTS**

Software Hazard Analysis and Safety Verification using Fault Trees

Nancy G. Leveson[†]

Information and Computer Science
University of California, Irvine
Irvine, California 92717
(714) 856-5517
e-mail: nancy@ics.uci.edu

Abstract

Contractors for embedded systems are starting to include requirements for software hazard analysis and verification of software safety in their contracts. This paper describes the problem and one possible approach to it — Software Fault Tree Analysis.

Introduction

A system or subsystem may be described as *safety-critical* if a run-time failure can result in death, injury, loss of equipment or property, or environmental harm. In safety-critical systems, it is not unusual to have reliability requirements in the range of 10^{-5} to 10^{-9} probability of failure over a short period of time. Unfortunately, current software engineering technology does not guarantee that such reliabilities can be achieved for software (or, for that matter, even measured). In fact, available evidence indicates that current software reliability figures are, at best, orders of magnitude less than required [3]. Software engineering techniques which attempt to prevent, eliminate, or tolerate software faults may increase the time between failures, but do not provide assurance that catastrophic failures will not occur.

What can be done? One option is not to build these systems or not to use computers to control them. For the most part, however, this option is unrealistic — there are too many good reasons why computers should be used and too few alternatives. Another option is to consider reliability in a less absolute sense. There are many types of failures possible in any complex system, with consequences varying from minor annoyance up to death or injury. It seems reasonable to focus on the failures that have the most drastic consequences. Even if all failures cannot be prevented, it may be possible to ensure that the failures that do occur are of minor consequence or that even if a potentially serious failure

[†]This work has been partially supported by NSF Grant No. DCR-8406532 and by Micro Grants cofunded by the University of California, Hughes Aircraft Co., and TRW.

does occur, the system will "fail safe" (i.e., fail in a manner which will not have catastrophic or serious results).

This approach is useful under the following circumstances: (1) not all failures are of equal consequences and (2) a relatively small percentage of failures lead to catastrophic results. These conditions seem to be true for most realistic safety-critical systems. Under these circumstances, it is possible to augment traditional reliability techniques that attempt to eliminate all failures with techniques that concentrate on the high-cost failures. These new techniques often involve a "backward" approach that starts with determining what are the unacceptable or high-cost failures and then ensures that these particular failures do not occur or at least minimizes the probability of their occurrence. This approach has been used on defense, aerospace, and various types of industrial systems.

Most safety-critical system purchasers are becoming concerned with software risk and are incorporating requirements for software safety analysis and verification in their contracts. In many countries, a formal validation and demonstration of the safety of the computers controlling safety-critical processes is required by an official licensing authority. In the U.S., DoD standards for building safety-critical systems either already include, or are being updated to include, software-related requirements. For example, a general safety standard (MIL-STD-882B) includes tasks for Software Hazard Analysis and Verification of Safety (including software). An Air Force standard for missile and weapon systems (MIL-STD-1574A) requires a Software Safety Analysis and Integrated Software Safety Analysis (which includes the analysis of the interfaces of the software to the rest of the system, i.e. the assembled system). And the U.S. Navy has a draft standard for nuclear weapon systems (MIL-STD-SNS) that requires Software Nuclear Safety Analysis. All of these analyses are not meant to substitute for regular verification and validation, but instead involve special analysis procedures to verify that the software is safe.

It is important to stress that these are *system* problems. When computers are used as components of larger systems, considering the computer software in isolation will be of limited usefulness. Many (if not most) serious accidents are caused by complex, unplanned (and unfortunate) interactions between components of the system and by multiple failures. That is, most accidents originate in subsystem interfaces [4,5]. Software failures and software-induced system failures may be caused by undetected hardware errors such as transient faults causing mutilation of data, security violations, human mistakes during operation and maintenance, errors in underlying or supporting software systems, or interfacing problems with other components of the system including timing errors and specification errors. Therefore, techniques used to build software for embedded systems, especially with respect to analysis and verification, are going to have to consider the system as a whole (especially the interactions between the components of the system or subsystem) and not just the software in isolation.

In fact, after studying actual accidents where computers were involved, safety engineers have concluded that inadequate design foresight and specification errors (i.e., fundamental misunderstanding about the desired operation of the software) are the greatest cause of software safety problems [4,6]. These problems arise from many possible causes including the difficulty of the problem intrinsically, a lack of emphasis on it in software engineering research (which has tended to concentrate on avoiding or removing implementation faults), and a certain cubbyhole attitude that has led computer scientists to concentrate on the computer aspects of the system and engineers to concentrate on the physical and mechanical parts of the system with few people dealing with the interaction between the two [6]. Many hardware-oriented system engineers do not understand software due to the newness of software engineering and the significant differences between software and hardware. The same is true, only vice versa, for software engineers. This has led to system engineers considering the computer as a black box [6,7] while the software engineer has treated the computer as merely a stimulus-response system. This lack of communication has been blamed for several accidents.

One such incident involved a chemical reactor [7]. The programmers were told that if a fault occurred in the plant, they were to leave all controlled variables as they were and to sound an alarm. One day, the computer received a signal telling it that there was a low oil level in a gearbox (see figure 1). The computer reacted as the requirements specified: it sounded an alarm and left the controls as they were. By coincidence, a catalyst had just been added to the reactor and the computer had just started to increase the cooling-water flow to the reflux condenser. The flow was therefore kept at a low value. The reactor overheated, the relief valve lifted, and the contents of the reactor were discharged into the atmosphere. The operators responded to the alarm by looking for the cause of the low oil level. They established that the level was normal and that the low-level signal was false but, by this time, the reactor had overheated.

An obvious conclusion from the above is that system-level methods and viewpoints are necessary. Note that the software itself is not "unsafe." Only the hardware that it controls can do damage. Treating the computer as a stimulus-response system allows verifying only that the computer software itself is consistent or safe; there is no way to verify system correctness or system safety. To do the latter, it must be possible to verify the correctness of the relationship between the input and the system behavior (not just the computer output). Boebert [1] has argued that verification systems that prove the correspondence of source code to concrete specifications are only fragments of verification systems. They do not go high enough (to an inspectable statement of system behavior), and they do not go low enough (to the object code).

Murphy is an experimental methodology being developed to deal with these problems. The goal is to provide procedures and an integrated tool set for building safety-critical real-time software. In general, the following questions are being

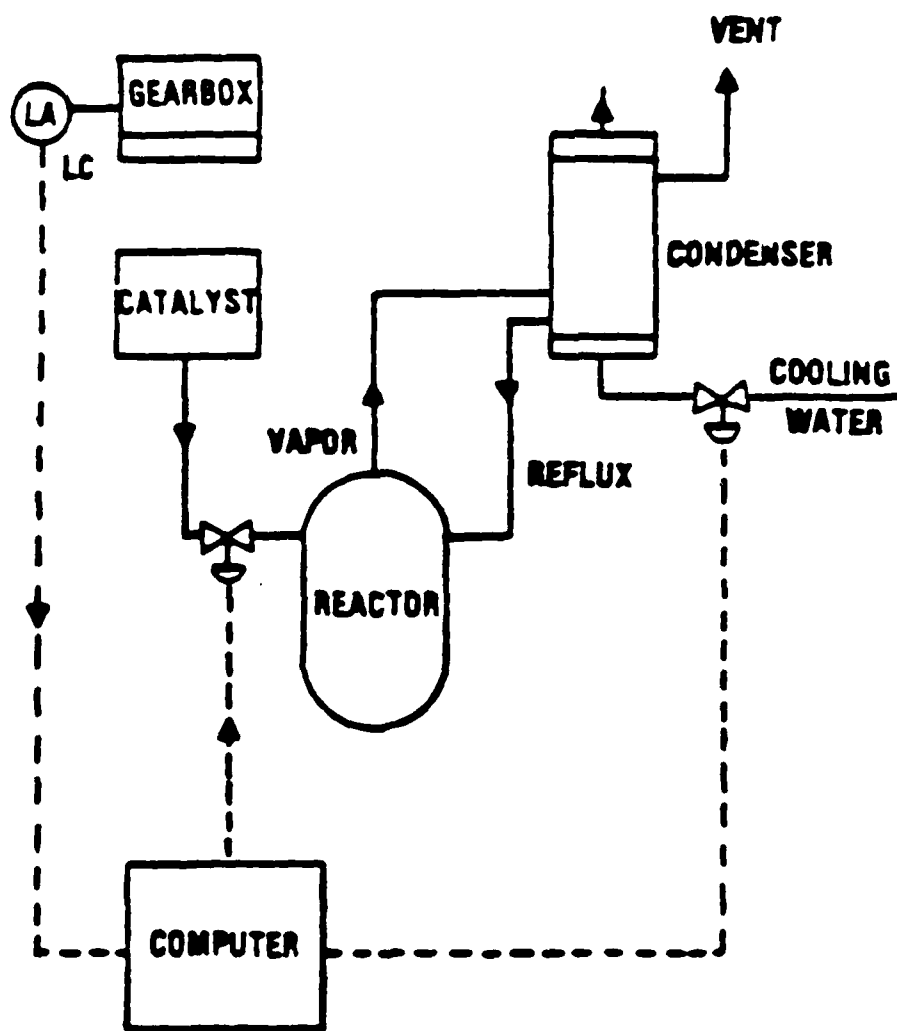


Figure 1. A computer-controlled batch reactor

considered:

- *Software Hazard Analysis and Requirements Specification:* What kinds of system models and analysis tools are most useful? How can software requirements be derived from these system models? How can the models and requirements be analyzed to determine important reliability and safety properties?
- *Verification and Validation:* How can safety properties be identified, specified, and formally verified? What techniques appear the most promising? How can they be implemented so that they can be used in industrial environments and not just in university research labs?
- *Assessment of Safety:* How can the safety of software be accurately measured and assessed? Is this possible? Is this feasible?
- *Software Design and Run-Time Environments:* What techniques and environments are most appropriate for safety-critical software? How can the software detect unsafe states during execution? What types of self-monitoring, external monitoring, fault-tolerance, fail-safe, and other software design techniques can be used to aid in the design of the software especially with regard to handling run-time fault detection and recovery?

Murphy is currently far from a complete methodology. Since it is still in the formative stages, much of the work has involved examining alternative approaches. This paper describes some of the work in software hazard analysis and verification of safety. More general discussion of software safety can be found in [8,10], and a more complete description of Murphy appears in [9].

Software Hazard Analysis

A *mishap* or *accident* is defined as an unplanned event or series of events that results in death, injury, occupational illness, damage to or loss of equipment or property, or environmental harm. Mishaps result from *hazards* or states of the system that when combined with certain environmental conditions can lead to a mishap. The first step in any safety program is to identify hazards and categorize them with respect to *risk* where risk is a function of (1) the probability of the hazardous state occurring, (2) the probability of the hazard leading to a mishap, and (3) the perceived severity of the worst potential mishap that could result from the hazard. This initial identification of hazards is called a *Preliminary Hazard Analysis* (PHA). Potential hazards considered involve normal operating modes, maintenance modes, system failure modes, failures or unusual incidents in the environment, and errors in human performance.

Once the System Preliminary Hazard Analysis is completed, Software Hazard Analysis (SHA) can begin. Software hazards include such things as

- failing to provide a required function, i.e., the function is never executed or no answer is produced,
- performing a function that is not required, i.e., getting the wrong answer or issuing the wrong instruction or doing the right thing but under inappropriate conditions (for example, activating an actuator inadvertently, too early, too late, or failing to cease an operation at a prescribed time),
- timing or sequencing problems, e.g. failing to ensure that two things happen at the same time, at different times, or in a particular order,
- failing to recognize a hazardous condition requiring corrective action,
- producing the wrong response to a hazardous condition.

Once the software hazards have been identified, the next step in SHA is to work backward from the specific hazards for the software under consideration and to locate software faults or paths through the software which could cause the unwanted hazardous conditions or to verify that such paths do not exist. The verification should include the software interfaces including system interfaces and computer hardware interfaces (e.g., hardware failures which could cause the software and hence the system to operate in a hazardous manner). Failures need to be considered along with normal operation.

The final step in SHA is to use the results of the analysis to guide further design and to guide placement and content of run-time checks and software fault tolerance and fail-safe procedures. For example, it may be possible to use the information obtained in the analysis to help write acceptance tests for the software and to determine conditions under which fail-safe procedures should be initiated.

We have been studying ways to accomplish software hazard analysis using Time Petri Nets [14] and Fault Tree Analysis (FTA) [12]. This paper will concentrate on describing the FTA procedures.

Software Fault Tree Analysis (SFTA)

Fault Tree Analysis (FTA) [16] is an analytical technique used in the safety analysis of electromechanical systems. An undesired system state is specified, and the system is then analyzed in the context of its environment and operation to find credible sequences of events that can lead to the undesired state. The fault tree is a graphic model of various parallel and sequential combinations of faults that can result in the occurrence of the predefined undesired event. A fault tree thus depicts the logical interrelationships of basic events that lead to the hazardous event.

The analysis process starts with the categorized list of system hazards that have been identified by the PHA. A separate fault tree must be constructed for

each hazardous event. The basic procedure is to assume that the hazard has occurred and then to work backward to determine its set of possible causes. The root of the fault tree is the hazardous event to be analyzed called the *loss event*. Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are unable to be analyzed for some reason. Figure 2 shows part of a fault tree for a hospital patient monitoring system.

Once the fault tree has been built down to the software interface (as in figure 2), the high level requirements for software safety have been delineated in terms of software faults and failures that could adversely affect the safety of the system. As the development of the software proceeds, fault tree analysis can be performed on the design [11] and finally the actual code [12].

When considering the implemented system, software fault tree analysis procedures can be used to work backward from the critical control faults determined by the top levels of the fault tree through the program to verify whether the program can cause the top-level event or mishap. The basic technique used is the same backward reasoning (weakest precondition) approach that has been used in formal axiomatic verification [2], but applied slightly differently than is common in "proofs of correctness."

The set of states or results of a program can be divided into two sets — correct and incorrect. Formal proofs of correctness attempt to verify that given a precondition that is true for the state before the program begins to execute, then the program halts and a postcondition (representing the desired result) is true. That is, the program results in correct states. For continuous, purposely non-halting (cyclic) programs, intermediate states involving output may need to be considered. The basic goal of safety verification is more limited. We will assume that, by definition, the correct states are safe (i.e., that the designers did not intend for the system to have mishaps). The incorrect states can then be divided into two sets — those that are considered safe and those that are considered unsafe. Software Fault Tree Analysis attempts to verify that the program will never allow an unsafe state to be reached (although it says nothing about incorrect but safe states).

Since the goal in safety verification is to prove that something will not happen, it is useful to use proof by contradiction. That is, it is assumed that the software has produced an unsafe control action, and it is shown that this could not happen since it leads to a logical contradiction. Although a proof of correctness should theoretically be able to show that software is safe, it is often impractical to accomplish this because of the sheer magnitude of the proof effort involved and because of the difficulty of completely specifying correct behavior. In the few SFTA proofs that have been performed, the proof appears to involve much less work than a proof of correctness (especially since the proof procedure can stop as soon as a contradiction is reached on a software path). Also, it is

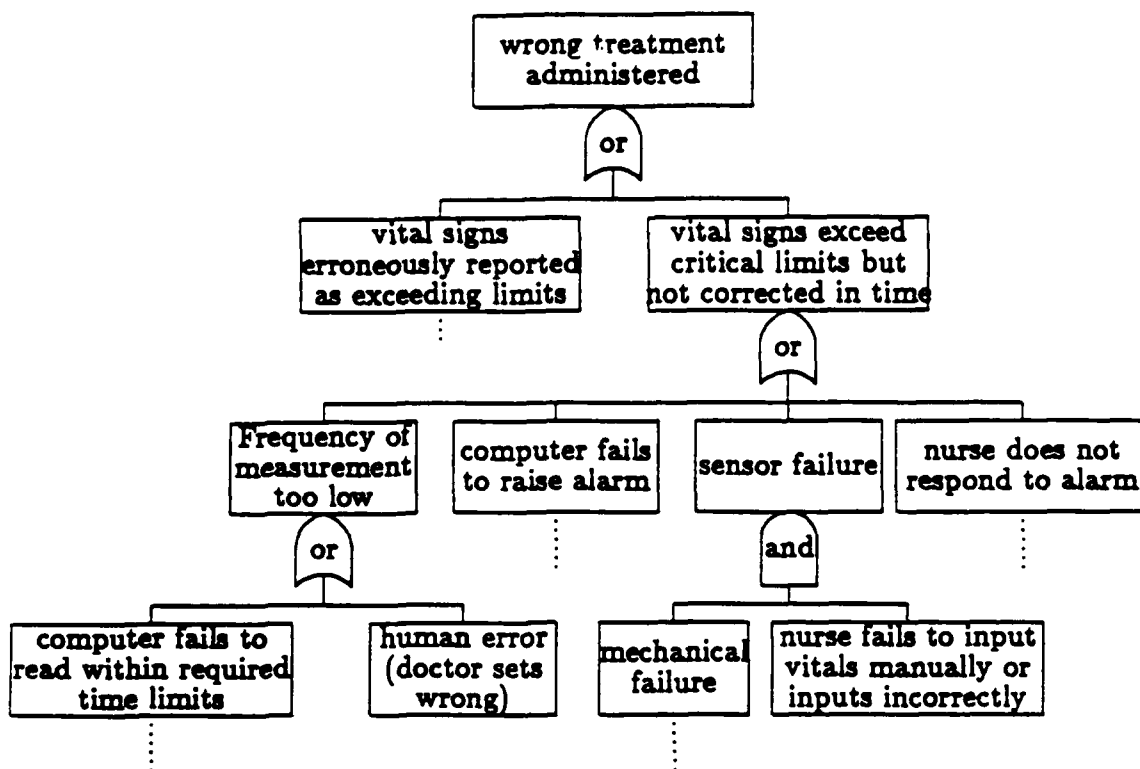


Figure 2: Top Levels of Patient Monitoring System Fault Tree

often easier to specify safety than complete correctness, especially since the requirements may be actually mandated by law or government authority as with nuclear weapon safety requirements in the U.S. Like correctness proofs, the analysis may be partially automated, but highly skilled human help is required.

Software fault tree analysis starts at the software interface of the system fault tree and works back through the logic of the code. Constructs for some structured programming language statements are shown in Figures 3 through 8. In each, it is assumed that the statement caused the critical event. Then the tree is constructed considering how this might occur. An example of the procedure is shown in Figures 9 and 10. An Ada program segment is shown which iteratively solves a fixed point equation. One possible top-level (loss event) for the segment is that no answer is produced in the required time period (and the answer is critical at this point). This loss event corresponds to the while loop executing too long (shown in figure XX as "Max" iterations).

In general, the software fault tree has one or both of the following patterns:

- 1) A contradiction is found as shown in the left branch of figure XX. The building of the software fault tree (at least for this path) can stop at this point since the logic of the software cannot cause the event. This example does not deal with the problem of failures in the underlying implementation of the software, but this is possible. There is, of course, a practical limit to how much analysis can and need be done depending on individual factors associated with each project. It is always possible to insert assertions in the code to attempt to catch critical implementation errors at run-time. This is especially desirable if run-time software-initiated or software-controlled fail-safe procedures are possible. Note that the software fault tree provides the information necessary to determine which assertions and run-time checks are the most critical and where they should be placed. Since checks at run-time are expensive in terms of time and other resources, this information is extremely useful.
- 2) The fault tree runs through the code and out to the controlled system or its environment. In the example of Figure 10, the fault tree shows one possible path to the loss event, and changes are necessary to eliminate the hazard. One appropriate action in this case may be to use run-time assertions to detect such conditions and to simply reject incorrect input or to initiate recovery techniques. Another possibility is to add redundant hardware, e.g. sensors, to eliminate incorrect input before it occurs.

Fault trees can also be applied at the assembly language level to identify computer hardware fault modes (such as erroneous bits in the program counter, registers, or memory) that will cause the software to act in an undesired manner.

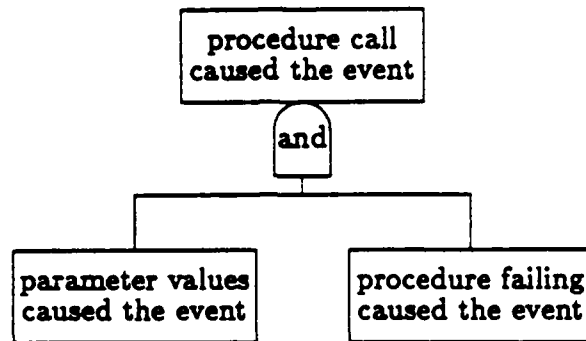


Figure 3 : Fault Tree for a Procedure Call

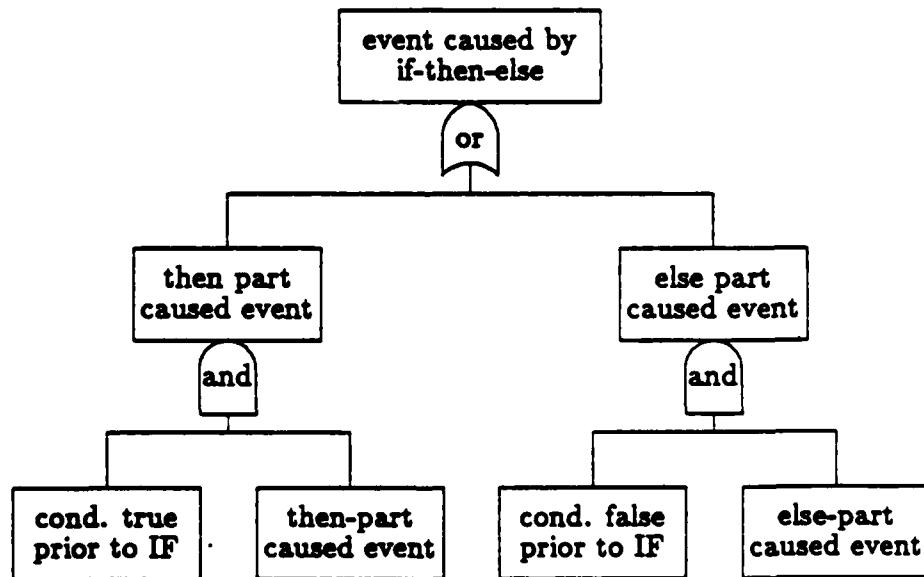


Figure 4 : Fault Tree for an If-Then-Else Statement

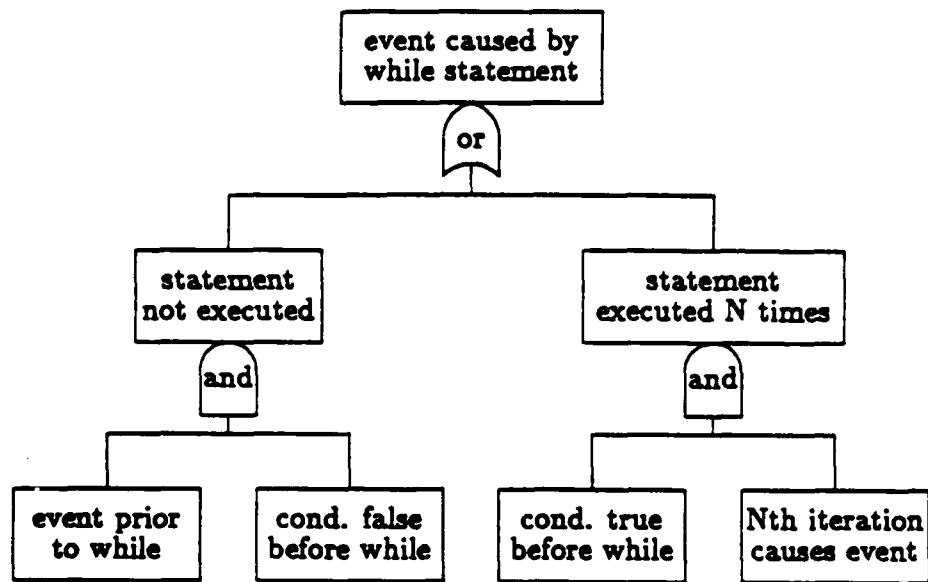


Figure 5: Fault Tree for a While Statement

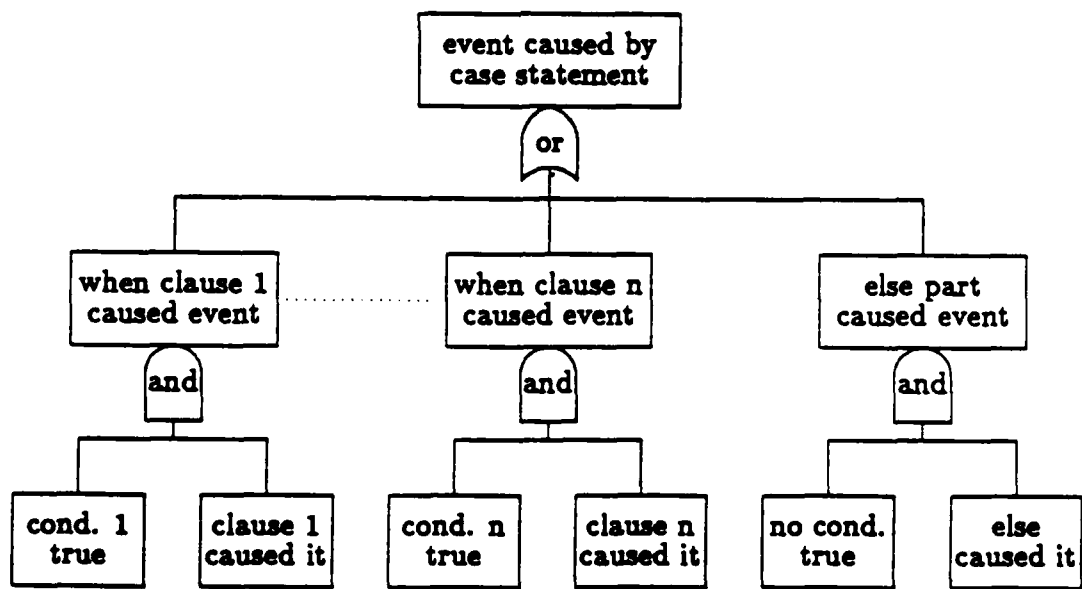


Figure 6 : Fault Tree for a Case Statement

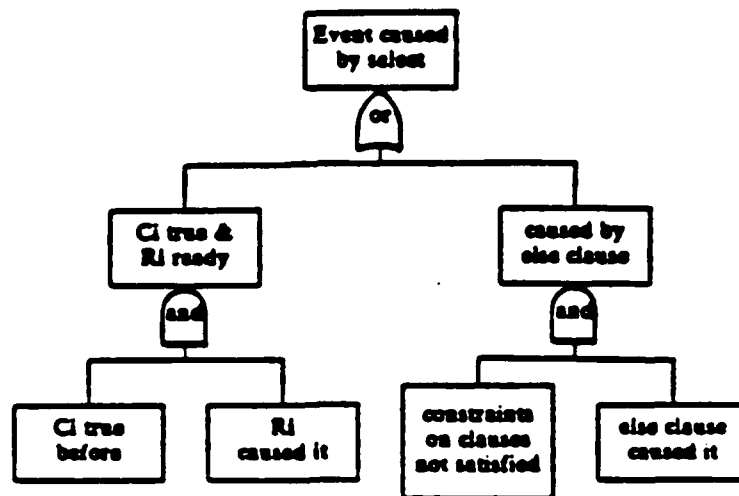


Fig. 7: Fault Tree for a Select Statement.

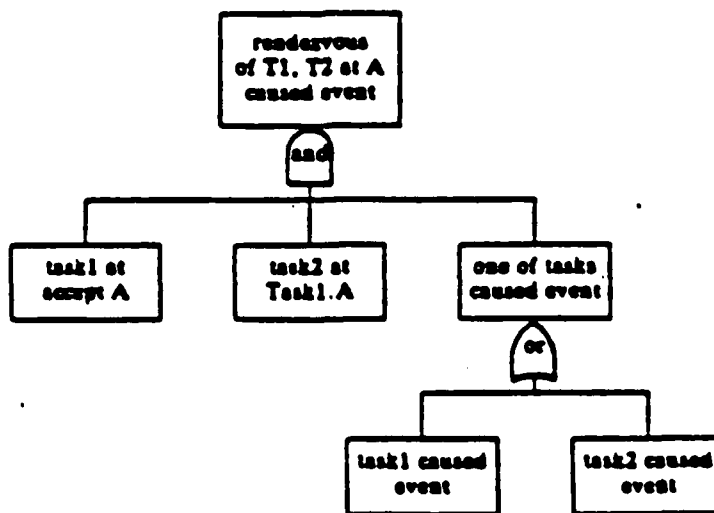


Fig. 8: Fault Tree Construct for a Rendezvous.

```
get (X, Eps);  
  
Err := Eps;  
I := 0;  
  
while Err ≥ Eps loop  
    NewX := F(X);  
    Err := abs(X - NewX);  
    I := I + 1;  
    X := NewX;  
  
end loop
```

Figure 9 : Example of Ada Code

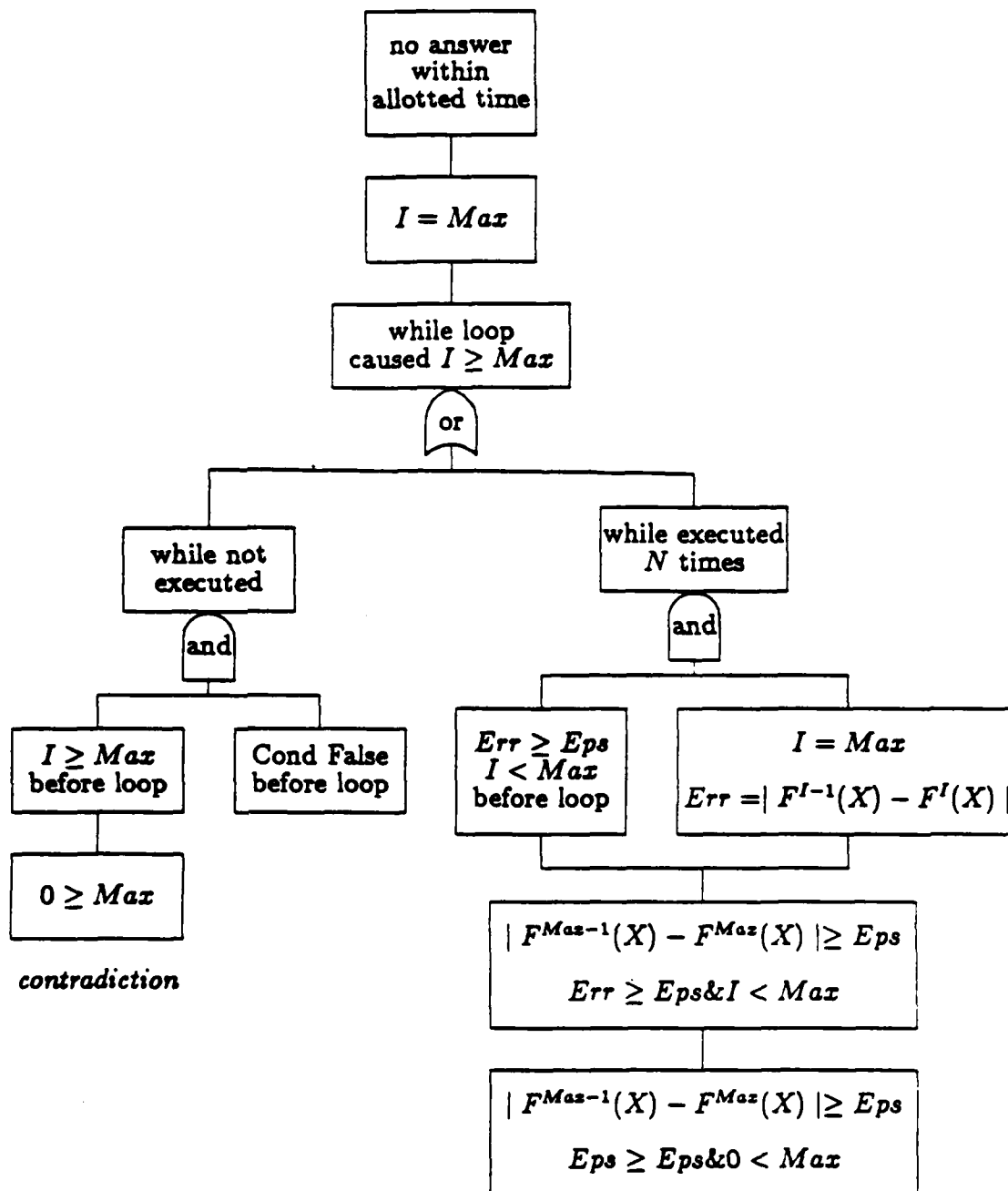


Figure 10: Fault Tree for Code in Preceding Figure

McIntee [15] has used this process to examine the effect of single bit failures on a software fuze. The procedure identified credible hardware failures that could result in the inadvertent early arming of the weapon. This information was used to redesign the software so that the failure could be detected and a "DUD" (fail-safe) routine called.

Experimental evidence of the practicality of SFTA is lacking. Examples of two small systems (approximately 1000 lines of code) can be found in the literature [12,15]. There is no information available on how large a system can be analyzed with a realistic amount of effort and time. But even if the software is so large that complete generation of the software trees is not possible, partial trees may still be useful. For example, partial analysis may still find faults. Furthermore, partially complete software fault trees may be used to identify critical modules and critical functions which can then be augmented with software fault tolerance procedures. They may also be used to determine appropriate run-time acceptance and safety tests [13].

In summary, software fault tree analysis can be used to determine software safety requirements, to detect software logic errors, to identify multiple failure sequences involving different parts of the system (hardware, human, and software) that can lead to hazards, and to guide in the selection of critical run-time checks. It can also be used to guide testing. The interfaces of the software parts of the fault tree can be examined to determine appropriate test input data and appropriate simulation states and events.

Summary

Safety is an important new application area for formal verification. Analyses such as software hazard analysis are now being required for safety-critical software, but the best way to accomplish this analysis is still unknown. This paper has briefly described one possibility — Software Fault Tree Analysis. There is currently work underway to extend the analysis to other Ada programming language constructs and to build an automated tool to aid in the analysis.

References

- [1] Boebert, W.E. "Formal verification of embedded software," *ACM Software Engineering Notes*, vol. 5, no. 3, July 1980, pp. 41-42.
- [2] Dijkstra, E. *A Discipline of Programming*, New York: Prentice Hall, 1976.
- [3] Dunham, J.R. and J.C. Knight (editors). "Production of reliable flight-critical software," *Proc. of Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting*, Research

Triangle Park, North Carolina, Nov. 2-4, 1981, NASA Conference Publication 2222.

- [4] Ericson, C.A. "Software and system safety," *Proc. 5th Int. System Safety Conf.*, Denver, 1981, vol. 1, part 1, pp. III-B-1 to III-B-11.
- [5] Frola, F.R. and Miller, C.O. *System Safety in Aircraft Management*, Logistics Management Institute, Washington D.C., January 1984.
- [6] Griggs, J.G. "A method of software safety analysis," *Proc. 5th Int. System Safety Conf.*, vol. 1, part 1, Denver, 1981, pp. III-D-1 to III-D-18.
- [7] Kletz, T. "Human problems with computer control," *Hazard Prevention R* (The Journal of the System Safety Society), March-April 1983, pp. 24-26.
- [8] Leveson, N.G. "Software Safety: Why, What, and How," Technical Report 86-04, ICS Dept., University of California, Irvine, 1986 (submitted for publication).
- [9] Leveson, N.G. "Building safe software," *Proc. Compass '86*, July 1986, (also available as Technical Report 86-14, ICS Dept, University of California, Irvine, 1986).
- [10] Leveson, N.G. "An Outline of a Program to Enhance Software Safety," *Proc. Safecom '86*, October 1986.
- [11] Leveson, N.G. "The Use of Fault Trees in Software Development," in preparation.
- [12] Leveson, N.G. and Harvey, P.R. "Analyzing software safety," *IEEE Trans. on Software Engineering*, SE-9, no. 5, Sept. 1983, pp. 569-579.
- [13] Leveson, N.G. and Shimeall, T. "Safety assertions for process control systems," *Proc. 13th Int. Conference on Fault Tolerant Computing*, Milan, Italy, 1983.
- [14] Leveson, N.G. and Stolzy, J.L. "Safety analysis using Petri nets," *IEEE Trans. on Software Engineering*, in press.
- [15] McIntee, J.W. Fault Tree Technique as Applied to Software (SOFT TREE), BMO/AWS, Norton Air Force Base, CA. 92409.

- [16] Vesely, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, Jan. 1981.

VERIFICATION OF ADA PROGRAMS FOR SAFETY

Prof. Nancy Leveson
Information and Computer Science
University of California Irvine

Real-Time Safety-Critical Systems

When computers are used to control complex, time-critical mechanical devices or physical processes such as:

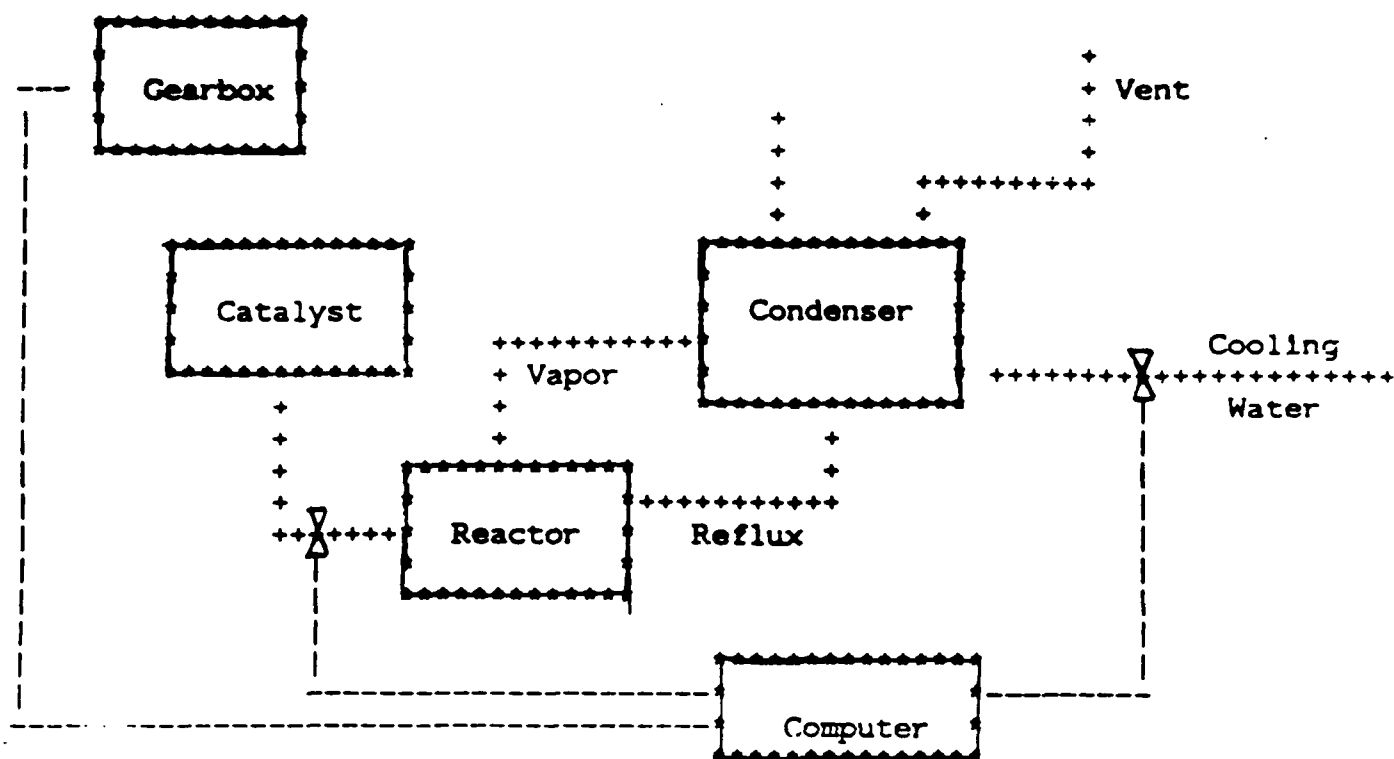
Air Traffic

Nuclear Fission

Hospital Patient Monitoring

Defense and Aerospace Systems

where a run-time error or failure can result in death, injury, loss of property, environmental harm.



Problems:

- **Cannot achieve needed reliability with current techniques**
- **Orders of magnitude less than required**
- **Most accidents can be traced back to inadequate design foresight and requirements specification -- Most software engineering techniques focus on implementation of requirements**

What can be done?

- Don't build these systems or don't use computers to control them
- Take a "system's" viewpoint
 - interface between computer and controlled system
 - interface between software and computer hardware
- Take a less "absolute" view of reliability
 - not all failures are of equal cost
 - minimize risk

Safety Approach to Software Development

- all failures are not created equal
- work backward from highest cost failures
- put effort into eliminating or preparing for high-cost failures.

Implications and Challenges for Software Engineering

- Requirements for software safety analysis and verification being included in contracts and by government licensing agencies.
- New standards for safety-critical software.
- National and international working groups
- Safety involves multiple areas of traditional software research along with safety engineering.

reliability

security

TASK 212

SOFTWARE HAZARD ANALYSIS

212.1 Purpose. The purpose of Task 212 is to perform and document a software hazard analysis to identify hazardous conditions incident to safety critical operator information and command and control functions identified by the PHA, SSHA, SHA, or other efforts.

212.2 Task Description. The contractor shall perform and document software hazard analysis on safety critical software-controlled functions to identify software errors/paths which could cause unwanted hazardous conditions.

212.2.1 Preliminary Software Hazard Analysis. These efforts shall examine software design to identify unsafe inadvertent command/failure-to-command modes for resolution. This effort shall be accomplished by tracing safety critical operator information and commands through flow charts, storage allocation charts, software and hardware specifications, and other applicable documentation.

212.2.2 Follow-on Software Hazard Analysis. These efforts shall examine software and its system interfaces for events, faults, and occurrences such as timing which could cause or contribute to undesired events affecting safety. This effort shall be accomplished by tracing safety critical operator information and commands through source/object code through system simulation and through other applicable documentation. Safety critical programs/modules shall be analyzed for sensitivity to software or hardware failures (bit transformation, register perversion, interface failures, etc.) which could cause the system to operate in a hazardous manner.

212.3 Details to be Specified by the MA (Reference 1.3.2.1).

212.3.1 Details to be specified in the SOW shall include the following, as applicable:

- (R) a. Imposition of Tasks 100 and 212.
- (R) b. Definition of safety critical.
- c. Format, content, and delivery schedule of any data required.
- d. Degree of fault-tolerance for Category I and II hazards.

TASK 212
30 March 1984

Software Safety: involves ensuring that the software will execute within a system context without resulting in unacceptable *risk*.

Risk is defined in terms of *hazards* -- states of the system that when combined with certain environmental conditions *could* lead to a mishap.

$$\text{Risk} = f (\text{Pr} [\text{hazard occurs}], \text{Pr} [\text{hazard leads to mishap}], \\ \text{Severity of worst potential mishap})$$

Safety critical software: software which can directly or indirectly cause or allow a hazardous system state to exist.

MURPHY

Techniques and tools for enhancing safety in real-time systems

- Safety Analysis and Requirements Tools

 - Fault Tree Analysis

 - Timed Petri Net Analysis techniques

- Verification and Assessment Tools

 - Software Fault Tree Analysis

 - Formal Verification

 - Measurement of Safety

- Run-Time Safety Techniques and Environments

 - Safety Assertions

 - Safety Monitor

 - Software Fault Tolerance

hazard = a set of conditions within a state from which there is a path to a mishap.

Goal in designing a safety-critical system.

- eliminate hazards from the design
- if not possible, then minimize risk by altering design so that there is a very low probability of hazard occurring.

Safety Analysis:

- 1) ensure that if design is correctly implemented and no failures occur, operation of system will not result in a mishap.
- 2) eliminate or minimize risk of faults or failures leading to a mishap by using fault-tolerance or fail-safe procedures.

SOFTWARE SAFETY ANALYSIS PROCEDURES

(1) Determine System Hazards (PHA)

(2) Use PHA to determine software hazards (software safety requirements)

- failure to perform a required function
- performing a function not required
- timing or sequencing problems
- failing to recognize a hazardous condition requiring corrective action
- producing wrong response to a hazardous condition

(3) Assume software safety failure and work backwards to determine set of possible causes (if any) or show that cannot be caused by logic of software

(4) Use results of analysis to:

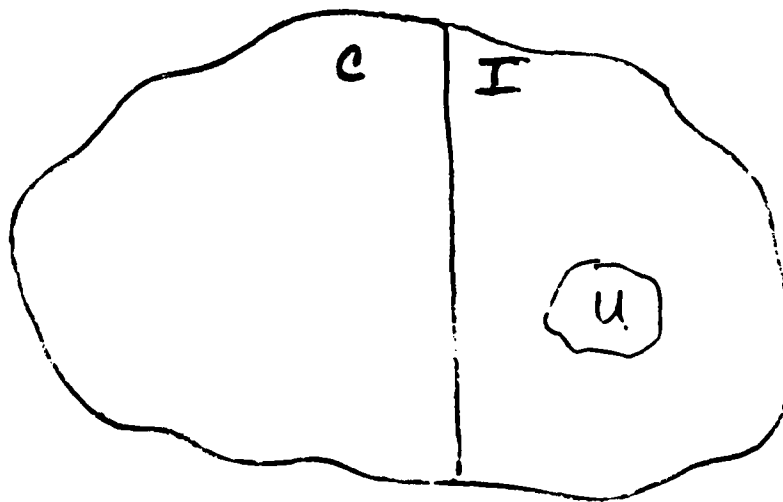
- Guide further design
- Pinpoint critical functions and test cases
- Guide placement and content of run-time checks
- Determine conditions under which fail-safe procedures should be initiated

Proof of "correctness"

- verify that given a precondition which is true for the state before the program executes, then the program will halt and a given postcondition will be true of the state once the program halts. That is, programs result in all and only correct states.

Proof of "safety"

- Divide incorrect states into safe and unsafe and verify that program will never allow an unsafe state to be reached (although says nothing about an incorrect but safe state).
- May be less work and easier to specify.



Fault Tree Analysis

- A graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the occurrence of a predefined undesired event.
- Events can involve hardware failures, human mistakes, software design faults, computer hardware failures, etc.
- Start with list of system hazards (PHA). Assume hazard has occurred, and work backward to determine set of possible causes. Preconditions described with either AND or OR relationships.

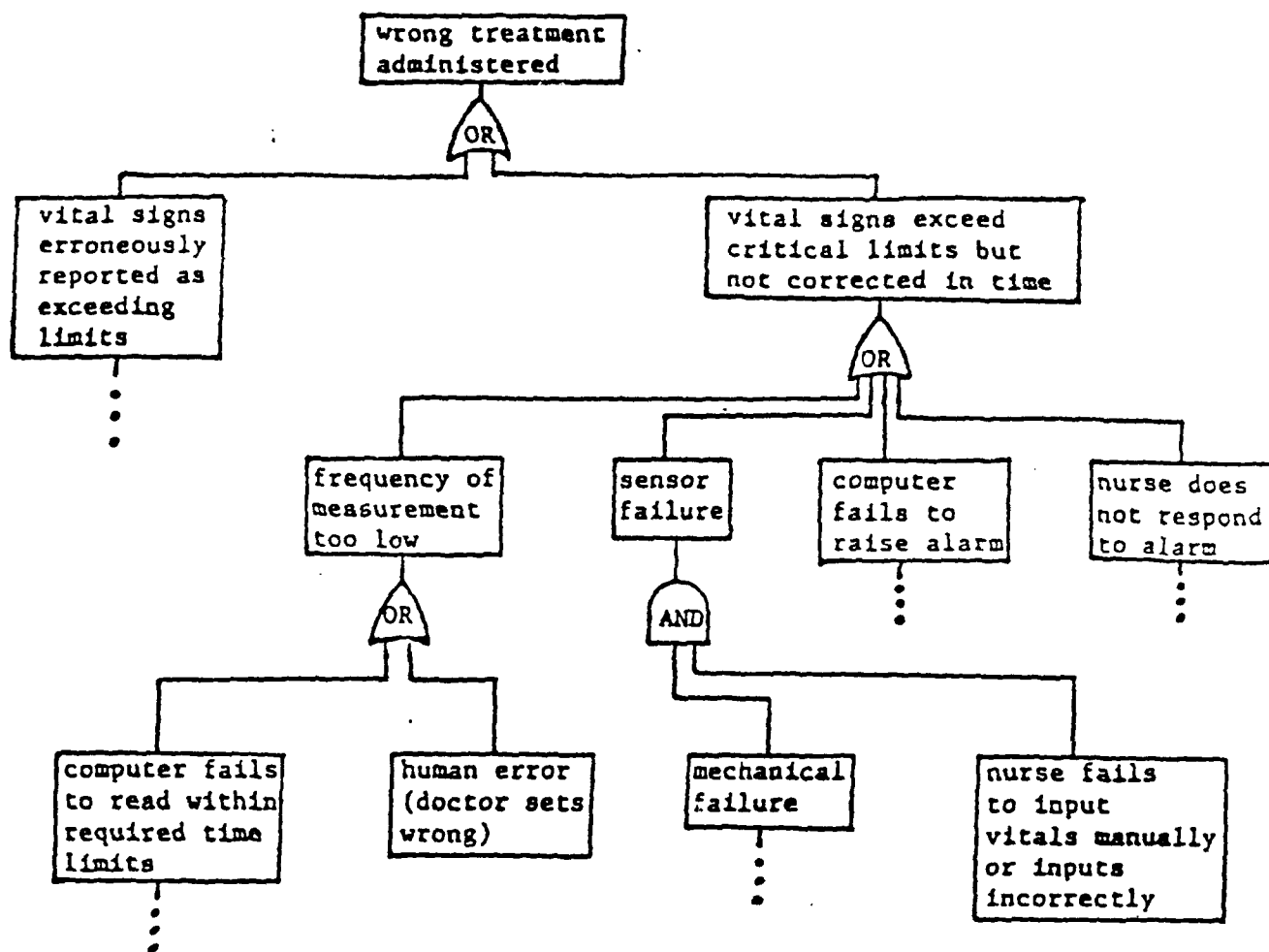


Figure 1. Top Levels of Patient Monitoring System Fault Tree

(1) $A := F(Y)$; (2) $B := X - 5.0$; (3) if $A > B$ then Sub1; end if;

Figure 8: Sample Assignment Statements

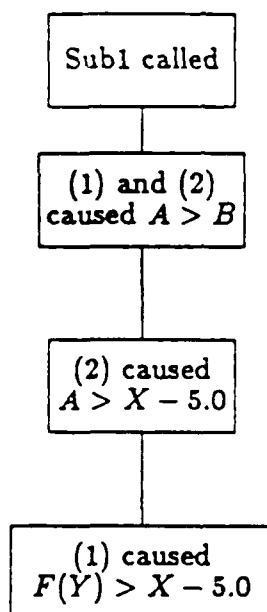
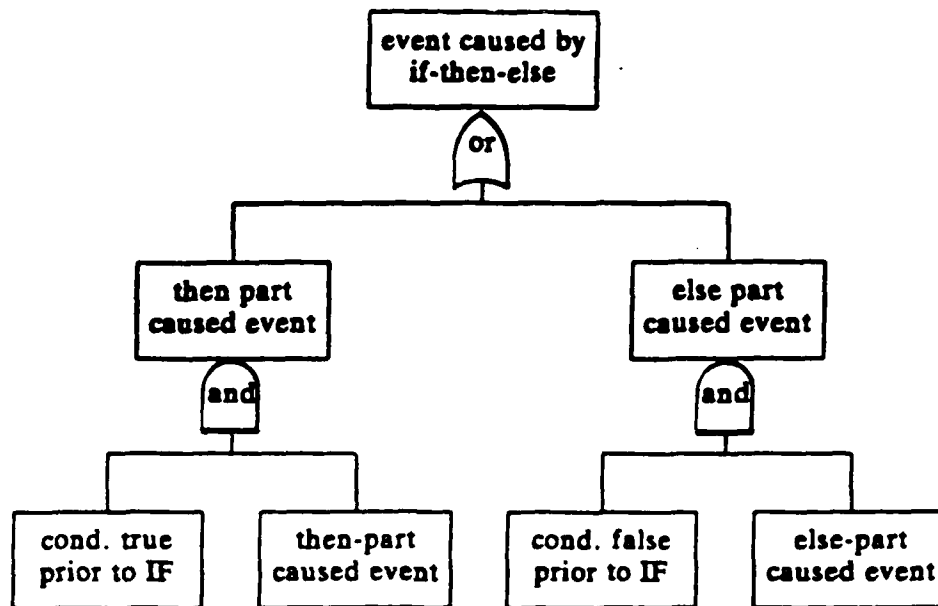
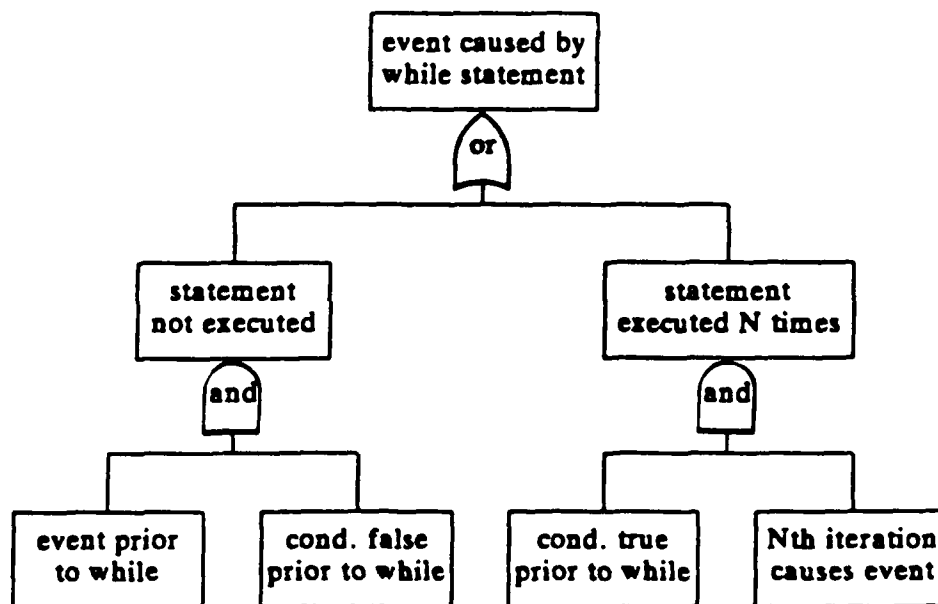


Figure 9: Fault Tree for Assignment Statements



Fault Tree for an If-Then-Else Statement
Figure 1a



Fault Tree for a While Statement
Figure 1b

```
get (X, Eps);
```

```
Err := Eps;
```

```
I := 0;
```

```
while Err ≥ Eps loop
```

```
    NewX := F(X);
```

```
    Err := abs(X - NewX);
```

```
    I := I + 1;
```

```
    X := NewX;
```

```
end loop
```

Figure 14: Example of Ada Code

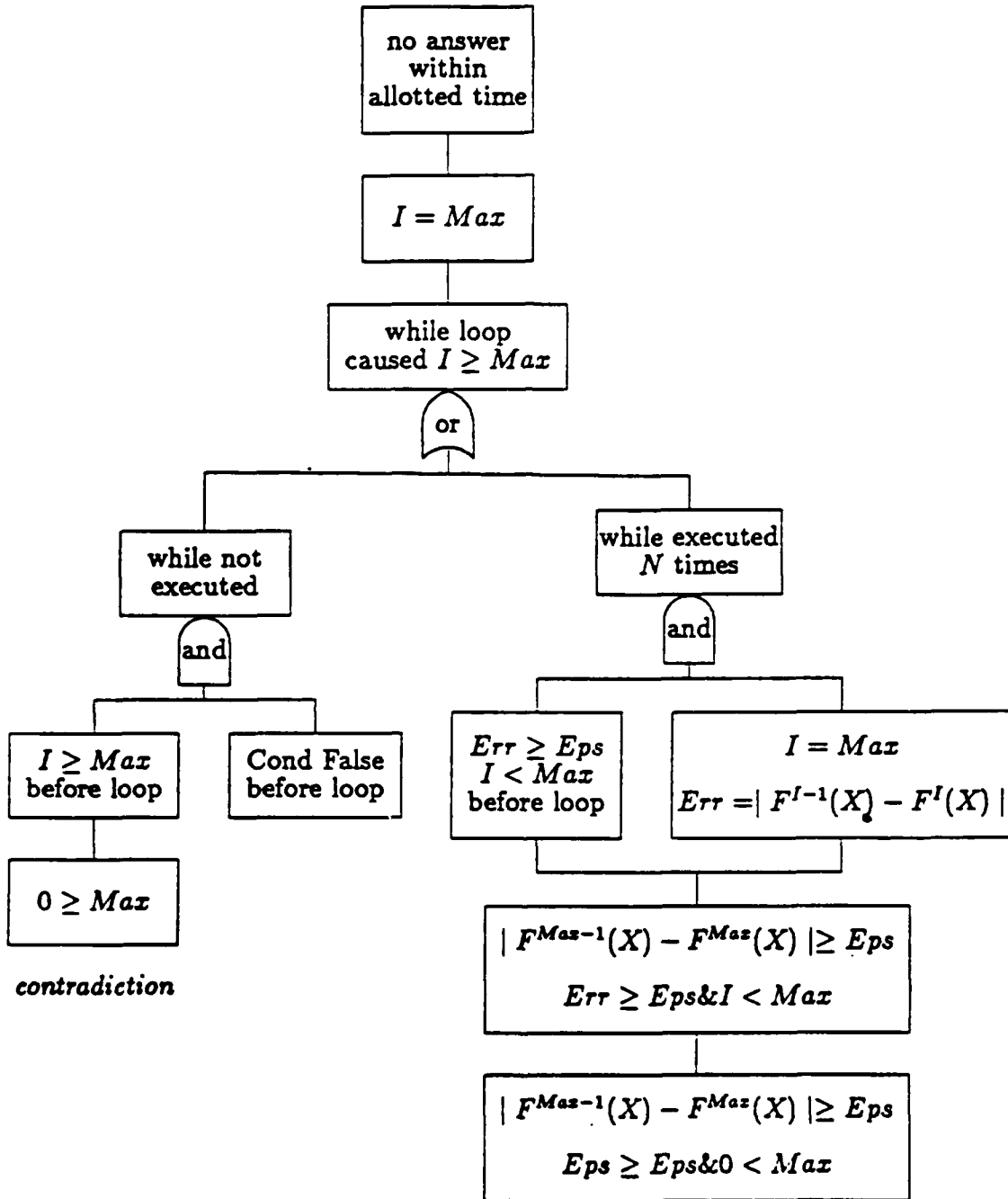


Figure 15: Fault Tree for Code in Preceding Figure

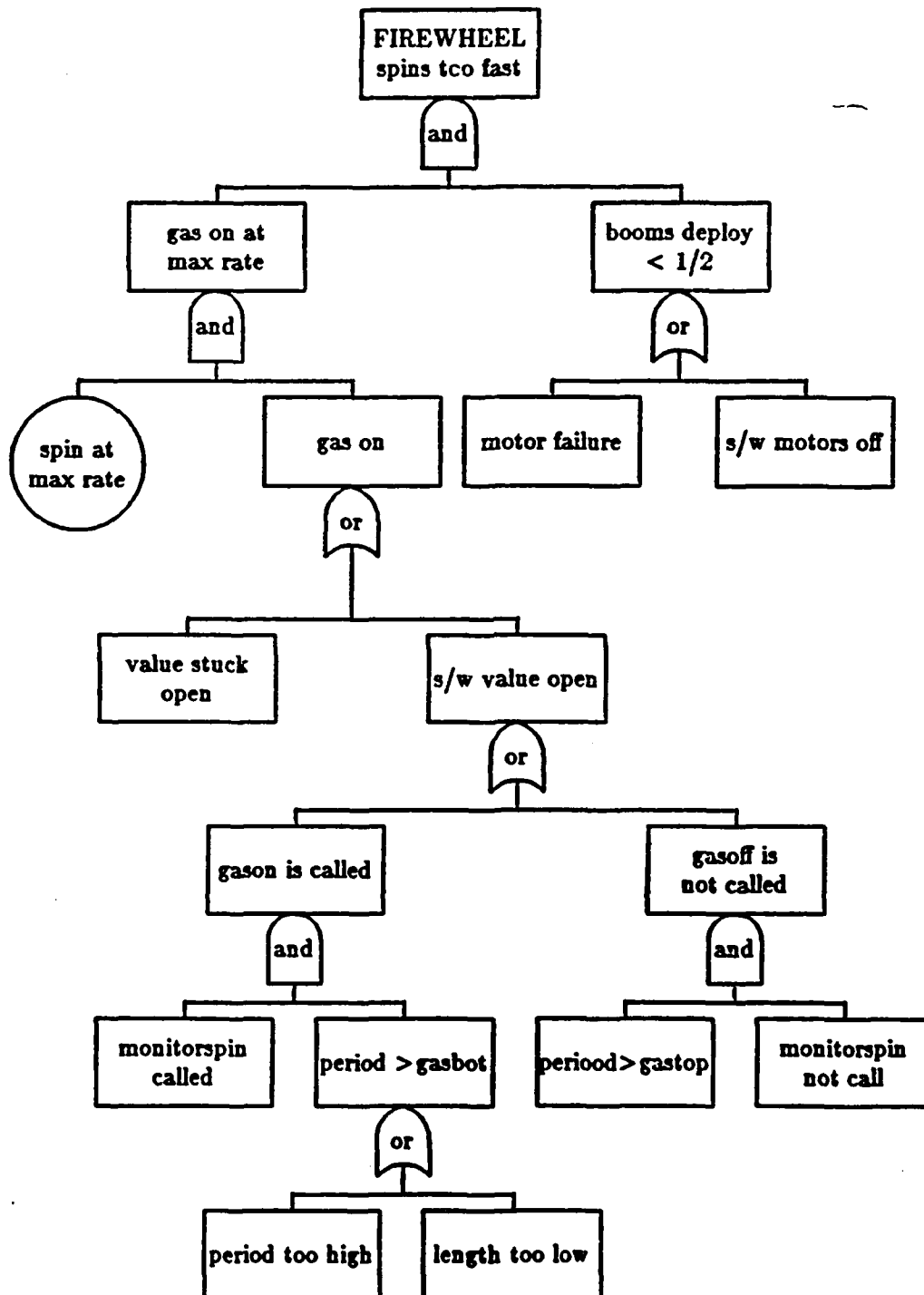
Software fault tree has two possible patterns:

- (1) A contradiction is found.
- (2) Fault tree runs through code and out to controlled system or its environment.

figure 7. Firewheel Spin Control – Software Opens gas Valve

file location: 7

initial fault: FireWheel spins too fast



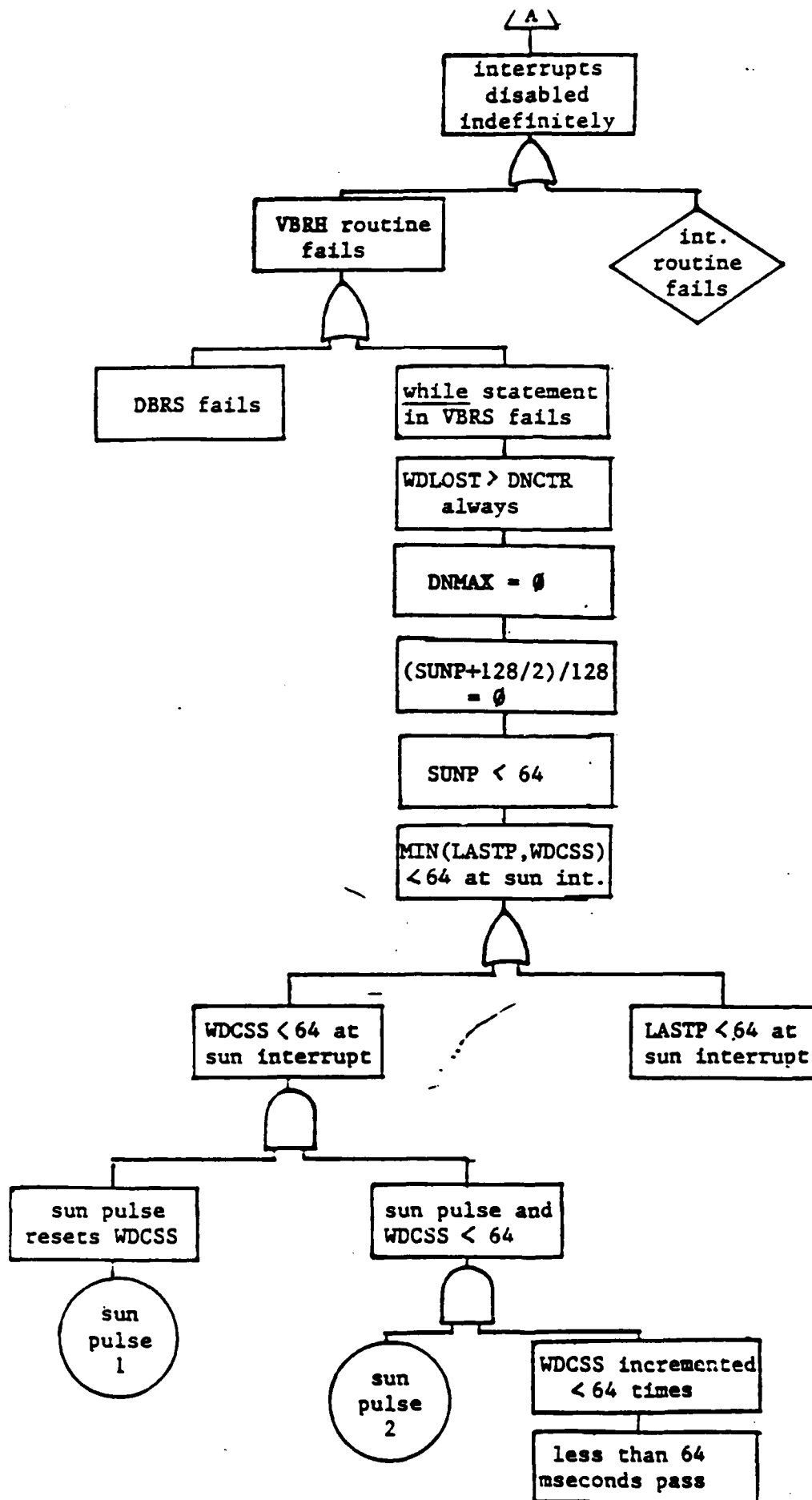
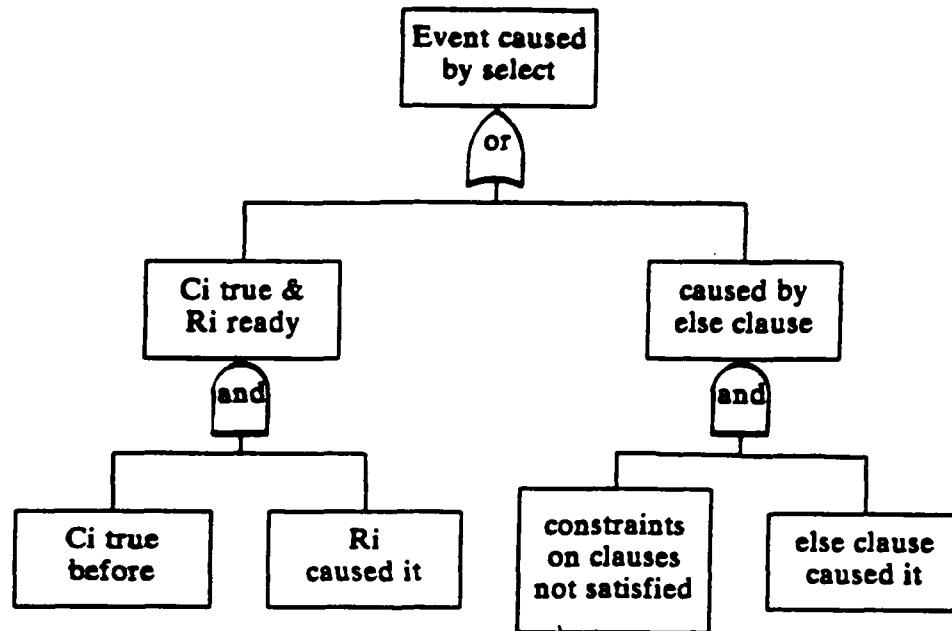


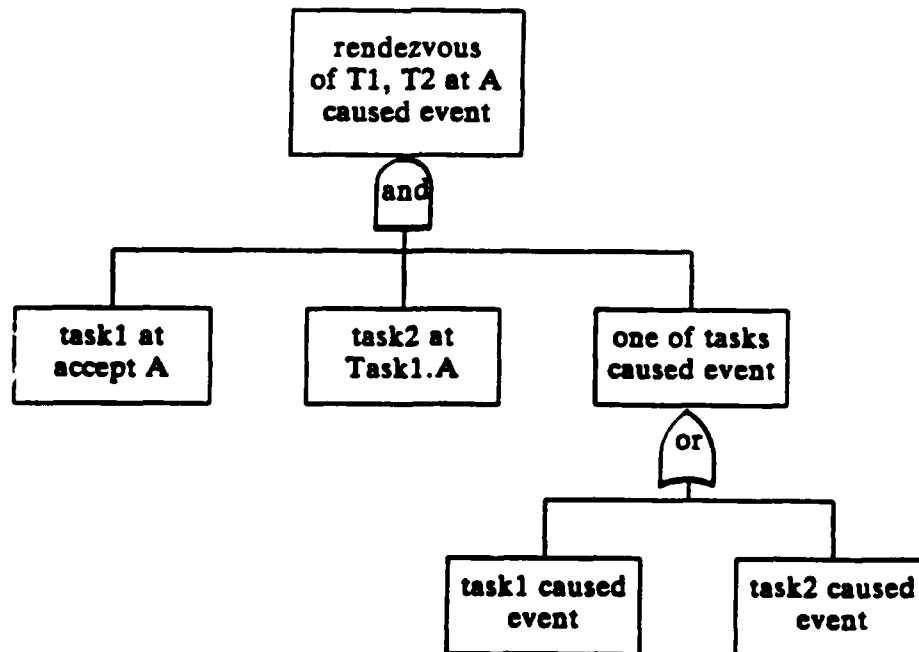
Figure 9b. Boom Length Too Low (continued)

Other issues:

- Concurrency
- Timing
- Computer Hardware Failures



**Fault Tree for a Select Statement
Figure 5**



**Fault Tree Construct for a Rendezvous
Figure 6**

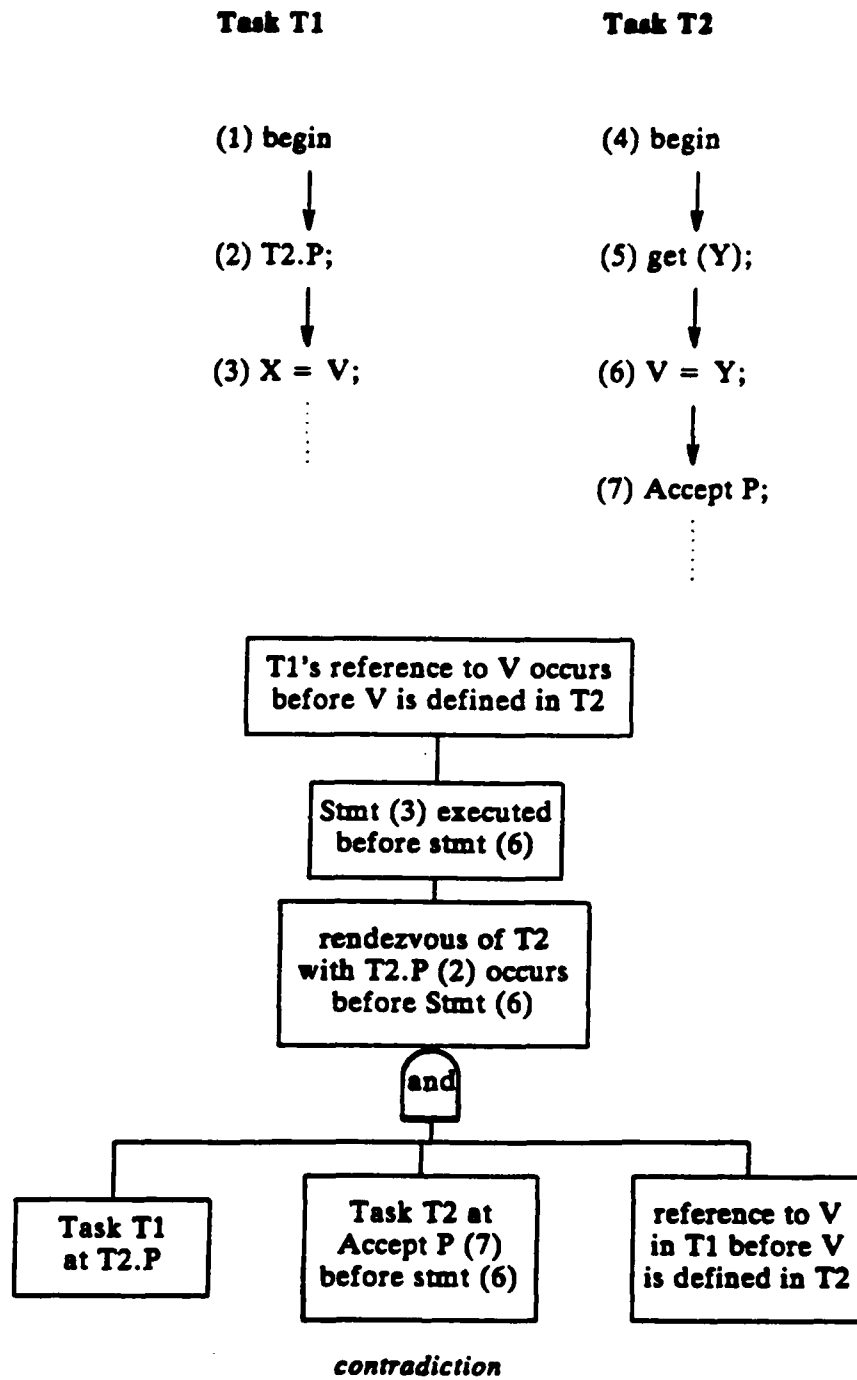


Figure 7

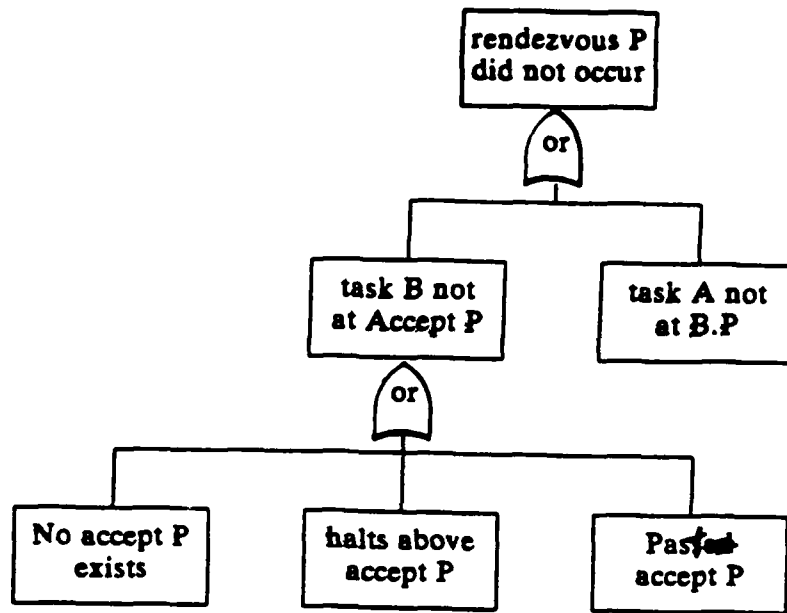


Figure 8

A proof rule for ADA

Ryan Stansifer

Department of Computer Science

Purdue University

West Lafayette, Indiana 47907

Telephone: (317) 494-7281

ryan@Purdue.edu

Abstract: We give a proof rule for a multiple-level exit construct not unlike the loop-exit statement in the ADA* programming language. We give a novel, yet simple, semantics for the loop-exit with which we can prove that the rule is both sound and (relatively) complete in the logic of Hoare triples. Hence, we can be satisfied that the proof rule is sufficient to prove all true Hoare triples using the multiple-level exit statement and is suitable for inclusion in a formal verification system.

Key words: Hoare logics, program verification, ADA programming language, denotational semantics.

* ADA is a registered trademark of the U.S. Government, ADA Joint Program Office.

A proof rule for ADA

Introduction. The programming language ADA has a general loop statement encompassing three different forms (or iteration schemes as they are called in section 5.5 of the reference manual). One form subsumes the other two, that is, the other forms can be derived from it. It is this most general case that we consider here. The syntax of this loop construct looks like:

`l : loop S end loop l;`

where *l* is the label of the loop. Execution of the loop statement proceeds by repeatedly executing the statements in *S* until a statement of the following form is encountered:

`exit l when B;`

When the boolean condition *B* evaluates to true, the execution of the loop labeled *l* is ended and the next statement in sequence after the loop is executed. If the condition is false, execution of the loop continues with the next statement in sequence after the exit statement. This exit statement can be encountered while nested inside of more than one loop. This is the cause of much travail in the denotational description of such constructs.

In this paper we give a denotational semantics for a simple language containing an ADA-like loop-exit construct. For this language we present an axiom system for deriving assertions about the correctness of programs. From this axiom system it is possible to determine what verification condition must be generated by a verification system for ADA. Although the language focuses on the loop-exit construct, certain generalizations are immediate (like the inclusion of the conditional construct). We show that this axiom system is sound using the denotational semantics given here. This is the least we can expect of the axiom system, and it insures that we can safely use it. We also show that this axiom system is (relatively) complete. Completeness guarantees that anything that

is true about the loop language does have a proof in the axiom system we give. This is important because it means we can stop looking for a more comprehensive set of proof rules. The proof of completeness requires a slightly different type of definition for the semantics of the loop language than the traditional one, but the definition integrates easily into the traditional definition. This permits the incorporation of yet other generalizations from the literature of denotational semantics.

We assume the reader is familiar with Hoare triples [Hoare, 1969], and somewhat familiar with denotational semantics [Stoy, 1977] and the classical soundness and completeness results for Hoare logic, for example [Loeckx et al., 1984].

The while loop. We begin by considering the proof rule for the while loop, which is a special case of the loop statement. The proof rule shows how to derive Hoare triples from other Hoare triples. Hoare triples are statements of the form $\{ P \} S \{ Q \}$, where P is an assertion called the precondition, S is a program segment, and Q is an assertion called the postcondition. A Hoare triple $\{ P \} S \{ Q \}$ asserts that starting the program S in a state satisfying P will result in a state satisfying Q (if S terminates). Here is the familiar rule for the while construct:

$$\frac{\{ I \& B \} S \{ I \}, \quad (I \& \neg B) \Rightarrow Q}{\{ I \} \text{ while } B \text{ loop } S \text{ end loop; } \{ Q \}}$$

This rule permits one to derive Hoare triples concerning the while loop, if one can derive the two premisses. The assertion I is called the loop invariant. Cook proved that this rule was sound and (relatively) complete in a computational semantics for while programs [Cook, 1978]. A similar result was proved by de Bakker who used denotational semantics to assign meaning to the program segments [de Bakker, 1980]. In denotational semantics each statement of the language is denoted by a function that transforms states to states. We use the symbol C for the traditional mapping of programs to their denotations. For example, the denotation of the while loop with condition B and body S is the recursive function f_{wh} defined as follows (we use σ as a formal parameter for states):

$$f_{\text{wh}}(\sigma) = \text{if IsTrue } (\mathcal{E}[B]\sigma) \text{ then } f_{\text{wh}}(C[S]\sigma) \text{ else } \sigma$$

where $\mathcal{E}[B]\sigma$ is the value of the boolean expression B in state σ , and $C[S]\sigma$ is the denotation of program segment S applied to the state σ (in other words, the resulting state obtained after executing S beginning in state σ). This is just one case, the case for the while loop, in the recursive definition of the function C . We write this case in the definition of C as:

$$C[\text{while } B \text{ loop } S \text{ end loop;}] \sigma = f_{\text{wh}}(\sigma)$$

Using this definition of the meaning of the while loop, the interpretation given to the Hoare triple $\{P\} \text{ while } B \text{ loop } S \text{ end loop; } \{Q\}$ is

$$\forall \sigma. P \text{ is true in } \sigma \ \& \ \sigma' \neq \text{error} \Rightarrow Q \text{ is true in } \sigma'$$

where $\sigma' = f_{\text{wh}}(\sigma)$. If $f_{\text{wh}}(\sigma)$ does not terminate, we set $\sigma' = \text{error}$. We have defined here a logic for partial correctness, since termination is a hypothesis in the implication above.

The loop-exit construct. The proof rule for the loop statement is reminiscent of the rule for the while statement. Here is the informal presentation of the rule.

$$\frac{\{I\} S \{I\}}{\{I\} l : \text{loop } S \text{ end loop } l; \{Q_l\}}$$

Like the while rule, the rule for the loop statement has an invariant assertion which we have called I in the rule above. Execution of S , the body of loop, must maintain the assertion I . It is interesting to note that the loop rule requires just one invariant assertion despite the possibility of multiple exit statements in the body of the loop. The rule for the exit statement has the premiss that if the “when” condition is true, the postcondition of the loop holds.

$$\frac{(Q \ \& \ B) \Rightarrow Q_l}{\{Q\} \text{ exit } l \text{ when } B; \{Q \ \& \ \neg B\}}$$

Here Q_l is the postcondition of the loop labeled l .

Now we check to see if we can derive certain obviously true Hoare triples from these rules. These rules would be inadequate if we could not use them to derive even simple Hoare triples concerning the loop and exit statements. Our purpose at present is to strengthen the plausibility of these rules. Later we will prove that these rules derive only true Hoare triples, and that all true Hoare triples can be proved using these rules.

For example, if the loop does not terminate we expect to conclude any postcondition Q (since this is a property of partial correctness logics). So we believe intuitively that the following Hoare triple is true and should be derivable:

$$\{P\} l : \text{loop null; end loop } l; \{Q\}$$

Indeed, this is derivable by a single application of the loop statement rule, if the Hoare triple $\{P\} \text{null}; \{P\}$ is derivable. This last Hoare triple can be taken as the meaning of the null or skip statement.

Another way the loop may not terminate is if the guard on the exit is always false, as in the program segment of the next Hoare triple:

$$\{P\} l : \text{loop exit } l \text{ when false; end loop } l; \{Q\}$$

The previous Hoare triple should be true, regardless of the assertions P and Q . This is derivable using the loop statement rule, if the Hoare triple $\{P\} \text{exit } l \text{ when false; } \{P\}$ is derivable. This follows from

$$\frac{(P \ \& \ \text{false}) \Rightarrow Q}{\{P\} \text{exit } l \text{ when false; } \{P \ \& \ \text{true}\}}$$

which is an instance of the exit rule.

Finally we expect the Hoare triple

$$\{P\} l : \text{loop exit } l \text{ when true; end loop } l; \{P\}$$

to be true, since this program segment acts like a no-op instruction. The Hoare triple above is derivable, since

$$\frac{(P \ \& \ true) \Rightarrow P}{\{P\} \text{ exit } l \text{ when } true; \{P \ \& \ false\}}$$

is an instance of the exit rule. (To obtain the postcondition P we must use the rule of consequence and the fact that $P \ \& \ false \Rightarrow P$.)

A simple language. To be specific we give the syntax of the language we wish to consider. This language is very simple; it contains hardly more than the loop-exit construct.

$$S ::= \text{assign}; \mid S_1 \ S_2 \mid l : \text{loop } S \text{ end loop } l; \mid \text{exit } l \text{ when } B;$$

We shall assume that loops are labeled uniquely in all program segments in this language. We call the program segment *closed* if all statements of the form $\text{exit } l_0 \text{ when } B$ are nested inside a loop statement labeled l_0 . Clearly only the closed program segments are meaningful when considered in isolation—a requirement an ADA compiler must check.

The denotations we give to program segments differ slightly from the typical denotations. Instead of transformations from states to states ($States \rightarrow States$), we use transformations from states to pairs of labels and states ($States \rightarrow (Labels \times States)$), where *Labels* is the set of possible identifiers for loops. We must add a special designator *ne* (for normal exit) to the set of labels. This designator indicates the normal sequential execution has been followed.

The denotations we give to program segments of the simple loop language are functions from states to pairs of labels and states. The semantic function that maps program segments to their denotations is denoted $\mathcal{X}$, and it has the functionality:

$$\mathcal{X} : S \rightarrow States \rightarrow ((ne + Labels) \times States)$$

Next we give the four cases in the recursive definition of the semantic function χ . The first case is that of assignment. We assume the assignment statement modifies the state in some matter.

$$\chi[\text{assign};]\sigma = (ne, \sigma')$$

The state σ' is the resulting, modified state after the assignment. The details of the modification are of no importance to the present discussion.

The next case is for the sequential execution of two program segments. If the execution of S_1 proceeds normally, then the program segment S_2 is executed in the resulting state. Otherwise, the execution of S_2 is skipped.

$$\begin{aligned} \chi[S_1 S_2]\sigma = \\ \text{let } (j, \sigma') = \chi[S_1]\sigma \text{ in} \\ \quad \text{if } j = ne \text{ then } \chi[S_2]\sigma' \text{ else } (j, \sigma') \\ \text{end} \end{aligned}$$

The remaining two rules are for the remaining two types of statements in the language, which together constitute the loop-exit construct. The exit statement is the one statement whose execution can result in initiating a path of executing that is not the "normal" sequential path represented by the pair (ne, σ) . But this occurs only if the guard is true, in which case the result is (l, σ) where l is the label of the loop being exited. Notice that the exit statement does not, in any case, change the state.

$$\begin{aligned} \chi[\text{exit } l \text{ when } B;]\sigma = \\ \text{if IsTrue } (\mathcal{E}[B] \sigma) \text{ then } (l, \sigma) \text{ else } (ne, \sigma) \end{aligned}$$

As in the case of the while statement, the denotation of the loop statement is a recursively defined function. We have called the function f_{lp} below.

$$\chi[l : \text{loop } S \text{ end loop } l]\sigma = f_{lp}(\sigma)$$

```

where rec  $f_{lp}(\sigma) =$ 
  let  $(j, \sigma') = \mathcal{X}[S]\sigma$  in
    if  $j = ne$  then  $f_{lp}(\sigma')$ 
    else if  $j = l$  then  $(ne, \sigma')$ 
    else  $(j, \sigma')$ 
  end
end

```

The function f_{lp} keeps calling itself recursively until $j \neq ne$. If j is the label of the current loop then the loop statement exits normally. This is the only instance in the definition of $\mathcal{X}$ that a subcomponent, in this case the body of the loop, exits with $j \neq ne$ and the language construct transforms it to a normal termination $j = ne$. The final case in the definition of f_{lp} is when $j \neq ne$ and $j \neq l$. In this case the exit of the loop labeled j is propagated, presumably to be caught by loop j .

For the purposes of defining which Hoare triples are true, we first define what we mean by an assumption. An *assumption* is a pair consisting of a label and an assertion. It is intended that the assumption (l, Q_l) represent the fact that Q_l is the postcondition of the loop labeled l . We shall be interested in sets of assumptions in which a label occurs at most once. We will call these sets *proper*. Sets of assumptions that are not proper indicate that a loop has more than one postcondition, and we have no use for this. This technique is inspired by a similar construction for goto statements [de Bruin, 1980].

We say that the Hoare triple $\{P\} S \{Q\}$ is *valid* with respect to a proper set of assumptions Φ (we will write this as $\Phi \vdash \{P\} S \{Q\}$) whenever

$$\forall \sigma. P \text{ is true in } \sigma \ \& \ \sigma' \neq error \Rightarrow Q_j \text{ is true in } \sigma'$$

where $(j, \sigma') = \mathcal{X}[S]\sigma$ and the assertion Q_j is defined as follows:

$$Q_j = \begin{cases} Q, & \text{if } j = ne; \\ Q_l, & \text{if } j = l \text{ for some } (l, Q_l) \in \Phi; \\ false, & \text{otherwise.} \end{cases}$$

Intuitively the notion of validity means that if P is true in σ and S exits normally, then Q is true in the resulting state, and if S exits a loop l , then Q_l is true in the resulting

state. If S exits some loop l_0 that is not in the set of assumptions, then the Hoare triple is automatically false.

Whenever S is closed (i.e., when $j = ne$ for all σ), the definition of validity corresponds to the usual one, because Q_j is always Q . Thus the semantics for the loop-exit construct presented here can be easily integrated into the usual semantics with the following definition:

$$\begin{aligned} C[S]\sigma = \\ \text{let } (j, \sigma') = \chi[S]\sigma \text{ in} \\ \quad \text{if } j=ne \text{ then } \sigma' \text{ else error} \\ \text{end} \end{aligned}$$

Thus, the denotations of closed statements can be viewed as state transformations like in the classical approach. This is important in fitting together these rules for the loop-exit construct with results on other constructs like procedure call rules.

The details. Next we give the precise rules for the loop-exit construct. This requires making the previous rules relative to proper sets of assumptions. Here is the rule for the loop statement:

$$\frac{\Phi \cup (l, Q_l) \vdash \{I\} S \{I\}}{\Phi \vdash \{I\} l : \text{loop } S \text{ end loop } l; \{Q_l\}}$$

We have discharged the assumption (l, Q_l) by enclosing S in the loop labeled l . The exit rule introduces the assumption (l, Q_l) .

$$\frac{(Q \ \& \ B) \Rightarrow Q_l}{(l, Q_l) \vdash \{Q\} \text{exit } l \text{ when } B; \{Q \ \& \ \neg B\}}$$

We list the remainder of the axiom system to show the effect of relativizing the usual rules.

$$\overline{\emptyset \vdash \{P'\} \text{assign}; \{P\}}$$

$$\frac{\Phi \vdash \{P\} S_1 \{Q\}, \quad \Psi \vdash \{Q\} S_2 \{R\}}{\Phi \cup \Psi \vdash \{P\} S_1 S_2 \{R\}}$$

$$\frac{P_1 \Rightarrow P_2, \quad \Phi \vdash \{P_2\} S \{Q_1\}, \quad Q_1 \Rightarrow Q_2}{\Phi \vdash \{P_1\} S \{Q_2\}}$$

With the precise statement of the rules of inference for the loop-exit constructs of the simple programming language, it is now possible to give the proof of soundness and (relative) completeness. We do not give the whole proof as that would not be illuminating. We give the cases of the proof for the loop and exit statements only, beginning with the exit statement, since it is easier.

Soundness of the exit statement. Suppose that $(Q \& B) \Rightarrow Q_l$. We are to show that the Hoare triple $\{Q\} S \{Q \& \neg B\}$ is valid with respect to (l, Q_l) , where S is the program segment `exit l when B`; . Let σ be such that Q is true in σ . The proof breaks into two cases. Either B is true in σ , in which case $\mathcal{X}[S]\sigma = (l, \sigma)$ and Q_l is true in σ by the hypothesis, or B is false in σ , in which case $\mathcal{X}[S]\sigma = (ne, \sigma)$. So, $Q \& \neg B$ is true in σ by assumption. Either way, the Hoare triple is valid.

Completeness of the exit statement. We are given that the Hoare triple $\{P\} S \{Q\}$ is valid with respect to (l, Q_l) , where S is the program segment `exit l when B`; . We must show we can derive this Hoare triple. We can derive this triple in two steps using the exit rule and the rule of consequence, if we can show that $(P \& B) \Rightarrow Q_l$ and $(P \& \neg B) \Rightarrow Q$.

$$\frac{\frac{(P \& B) \Rightarrow Q_l}{(l, Q_l) \vdash \{P\} \text{exit } l \text{ when } B; \{P \& \neg B\}}, \quad (P \& \neg B) \Rightarrow Q}{(l, Q_l) \vdash \{P\} \text{exit } l \text{ when } B; \{Q\}}$$

Assume first that P and $\neg B$ are true in σ . Then $\mathcal{X}[S]\sigma = (ne, \sigma)$. Since $\{P\} S \{Q\}$ is valid, Q is true in σ . In other words, $(P \& \neg B) \Rightarrow Q$. Now assume P and B are true in σ . In this case $\mathcal{X}[S]\sigma = (l, \sigma)$. Since $\{P\} S \{Q\}$ is valid with respect to (l, Q_l) , Q_l is true in σ . Hence both assertions hold.

Soundness of the loop statement. Suppose the Hoare triple $\{I\} S \{I\}$ is valid with respect to $\Phi \cup (l, Q_l)$. Let σ be some arbitrary state such that I is true in σ (and $f_{lp}(\sigma)$ terminates). Now define the (finite) sequence

$$\sigma_0, (ne, \sigma_1), (ne, \sigma_2), \dots, (ne, \sigma_{n-1}), (j, \sigma_n)$$

where $\sigma_0 = \sigma$ and $(j_i, \sigma_i) = \mathcal{X}[S]\sigma_{i-1}$. The sequence stops when $j_i \neq ne$, corresponding to when the execution of the loop halts and f_{lp} terminates. By induction it holds that I is true in σ_{n-1} . And thus, if $j = l$, then Q_l holds. If j is some other label in Φ , then the appropriate assertion holds as well. Hence,

$$\{I\} l : \text{loop } S \text{ end loop } l; \{Q_l\}$$

is valid with respect to Φ . This concludes the proof of soundness.

In the proof of completeness for the loop statement we will need the definition of the weakest precondition. The *weakest precondition* of S and Q (relative to proper set of assumption Φ), denoted $wp(S; Q)$, is that assertion such that

$$\Phi \vdash \{wp(S; Q)\} S \{Q\}, \quad \text{and} \quad P \Rightarrow wp(S; Q)$$

for all P such that $\Phi \vdash \{P\} S \{Q\}$. We assume that the weakest precondition is always expressible in the language of the assertions.

Completeness of the loop statement. Suppose that $\{P\} L \{Q\}$ is valid with respect to Φ , where L is the program segment $l : \text{loop } S \text{ end loop } l;$. We pick Q_l to be Q and I to be $wp(L, Q)$. So by definition $\{I\} L \{Q\}$ is valid with respect to Φ . By definition of the weakest precondition $P \Rightarrow I$, so from $\Phi \vdash \{I\} L \{Q\}$ we can derive $\Phi \vdash \{P\} L \{Q\}$ using the rule of consequence. Therefore, it remains to be proved that $\Phi \vdash \{I\} L \{Q\}$ is derivable. By the induction hypothesis we know we can derive this Hoare triple, if $\{I\} S \{I\}$ is valid with respect to $\Phi \cup (l, Q)$.

Suppose I is true in σ . Now execute the loop L beginning in this state. If we execute the body but "half" a time, then we must have $\mathcal{X}[S]\sigma = (j, \sigma')$ and $j \neq ne$. Since $\Phi \vdash \{I\} L \{Q\}$, $j = l$ or $j = l'$ for some $(l', Q_{l'}) \in \Phi$. If $\mathcal{X}[S]\sigma = (l, \sigma')$ then $\mathcal{X}[L]\sigma = (ne, \sigma')$, and since $\{I\} L \{Q\}$ we have Q is true in σ' . Hence $(l, Q) \vdash \{I\} S \{I\}$. If $j \neq l$, then

$$\Phi \cup (l, Q) \vdash \{I\} S \{I\}$$

follows because $(j, Q_j) \in \Phi$.

On the other hand, executing the loop may execute the body completely at least once.
In other words:

$$\Phi \cup (I, Q) \vdash \{I\} L \{Q\} \equiv \Phi \cup (I, Q) \vdash \{I\} S L \{Q\}$$

Since I is the weakest precondition of L we have $\{I\} S \{I\}$ (we leave the justification to the following lemma). This completes the proof.

Lemma. Suppose $\Phi \vdash \{P\} S_1 S_2 \{Q\}$ then $\Phi \vdash \{P\} S_1 \{R\}$ where R is $wp(S_2, Q)$. The proof is by contradiction. Suppose $\Phi \vdash \{P\} S_1 S_2 \{Q\}$, but $\{P\} S_1 \{R\}$ is not valid with respect to Φ . Then there is some state σ for which P is true in σ , but R is not true in σ' , where (j, σ') is $\chi[S_1]\sigma$. (If $j \neq ne$, then there is an immediate contradiction.) Then in no case is it possible to arrive at a state σ'' in which Q is true, as that contradicts the assumption that R is the weakest precondition.

Conclusion. Using a somewhat different semantics we have given a proof rule for an ADA-like loop-exit construct which is sound and complete. This rule can be safely included in a system to formally verify the correctness of ADA programs. The loop rule is no harder to use and understand than the rule for the while statement. The bookkeeping necessary for associating loop labels and the appropriate postconditions is straightforward. The only detail that prevents the rule from being applied mechanically is the discovery of the loop invariant. This, of course, is not surprising. What is surprising is that only one invariant must be found regardless of the number of exit statements.

References.

- Cook, Steven A. "Soundness and completeness of an axiom system for program verification." *SIAM Journal on Computing*, volume 7, number 1, February 1978, pages 70-90.
- de Bakker, Jacobus Willem. *Mathematical Theory of Program Correctness*. Prentice-Hall International series in computer science. Prentice-Hall International, London, 1980.
- de Bruin, Arie. "Chapter 10: Goto statements." In *Mathematical Theory of Program Correctness*, Prentice-Hall International, London, 1980.

Hoare, Charles Antony Richard. "An axiomatic basis for computer programming." *Comm. of the ACM*, volume 12, number 10, October 1969, pages 576-580, 583.

Loeckx, Jacques J. C., Kurt Sieber and Ryan D. Stansifer. *The Foundations of Program Verification*. Wiley-Teubner series in computer science. Teubner, Stuttgart, 1984.

Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massachusetts, 1977.

United States Department of Defense. *Reference Manual for the ADA Programming Language*. United States Government Printing Office, 1983.

A Proof Rule for Ada

Ryan Stansifer

D Structures

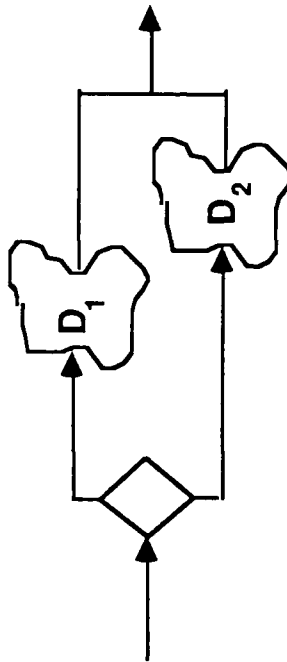
Basic Actions



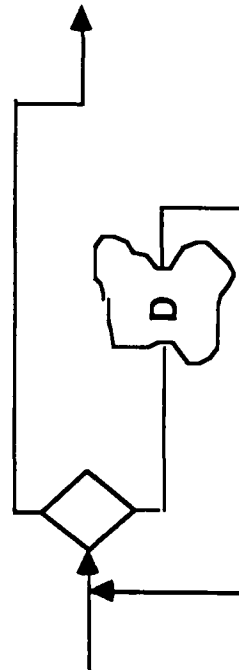
Composition



Conditional



White Loops



Context Free Grammar

$D :: =$

$a;$

$D_1 D_2$

if B then D_1 else D_2 end if;

while B loop D end loop;

SEMANTICS FOR D STRUCTURES

$\mathcal{D}: D \rightarrow (\text{States} \rightarrow \text{States})$

$\mathcal{D} [[a;]] \sigma = \sigma$

$\mathcal{D} [[D_1 D_2]] \sigma = \mathcal{D} [[D_2]] (\mathcal{D} [[D_1]] \sigma)$

$\mathcal{D} [[\text{if } B \text{ then } D_1 \text{ else } D_2 \text{ end if; }]] \sigma =$
 of IsTrue ($\mathcal{C} [[B]] \sigma$) then $\mathcal{D} [[D_1]] \sigma$ else $\mathcal{D} [[D_2]] \sigma$

$\mathcal{D} [[\text{while } B \text{ loop } D \text{ end loop; }]] = f_{wh}$
 where $\text{rec } f_{wh} (\sigma) =$
 if IsTrue ($\mathcal{C} [[B]] \sigma$)
 then $f_{wh} (\mathcal{D} [[D]] \sigma)$
 else σ
 end

ew7886/3

BJ_n STRUCTURES

```

loop
  exit when B1;
  S1
  exit when B2;
  .
  .
  .
  exit when Bn;
  Sn;
end loop;

```

1) n exits
2) can only exit one loop

context free grammar

```

BJ:: = a; BJ BJ
      if B then BJ else BJ end if;
      loop exit when B; BJ and loop;
      loop exit when B; BJ exit when B; BJ end loop;

```

SEMANTICS FOR BJ_n STRUCTURES

$\mathcal{B}[[\text{loop exit when } B_1; BJ, \text{ exit when } B_2; BJ_2 \text{ end loop;}]] =$

f_{bj}

where $\text{rec } f_{bj}(\sigma) =$

if $\text{IsTrue}(\mathcal{C}[[B_1]]\sigma)$

then σ

else let $\sigma' = \mathcal{B}[[BJ_1]] \sigma$ in

if $\text{IsTrue}(\mathcal{C}[[B_2]]\sigma)$

then σ'

else $f_{bj}(\mathcal{B}[[BJ_2]])$

end

ew7886/5

RE_n STRUCTURES

```

l: loop
.
.
.
m: loop
.
.
.
exit l when B;
.
.
.
and loop m;
and loop l;

```

1. Arbitrary no of exits in loop
2. Can exit nested_n deep
3. May exit within conditional

B[[m: loop exit l when B; end loop m;]] =

depends on context

SEMANTICS FOR RE STRUCTURES

$\mathcal{K}: S \rightarrow (\text{States} \rightarrow (\text{Labels} \times \text{States}))$

$\mathcal{K}[[a;]] \sigma = (me, \sigma')$

$\mathcal{K}[[S_1 S_2]] \sigma =$

let $(j_1 \sigma') = \mathcal{K}[[S_1]] \sigma$ in

if $j \neq ne$ then $\mathcal{K}[[S_2]] \sigma'$ else $(j_1 \sigma')$

end

$\mathcal{K}[[\text{exit } l \text{ when } B;]] \sigma =$

if $l \text{ is True } (\mathcal{K}[[B]] \sigma)$ then (l, σ) else (me, σ)

$\mathcal{K}[[l: \text{ loop } S \text{ end loop } l;]] = f_{lp}$

where $\text{rec } f_{lp}(\sigma) =$

let $(j_1 \sigma') = \mathcal{K}[[S]] \sigma$ in

if $j \neq ne$ then $f_{lp}(\sigma')$

else if $j \neq l$ then (ne, σ')

end else (j, σ')

end

ew7886/7

S is closed whenever S is properly nested

Fact: For closed S $j \neq e$ for all σ

integrate with "usual" state transformation semantics

$\mathcal{C}[[S]] d =$
let $(j, d_1) = \mathcal{X}[[S]] d$ in
if $j \neq e$ then $\partial \cdot$ else d error

end

Admission: neither semantics appropriate for Ada

ew7886/R

HOARE CALCULUS

$$\frac{}{\{P_1\} a; \{P\}}$$

$$\frac{\{P\} S_1 \{Q\} \{Q\} S_2 \{R\}}{\{P\} S_1 S_2 \{R\}}$$

$$\frac{\{P \& B\} S_1 \{Q\} \{P \& B\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ end if; } \{Q\}}$$

$$\frac{\{I \& B\} S \{I\}, (I \& B) \Rightarrow Q}{\{I\} \text{ while } B \text{ loop } S \text{ end loop; } \{Q\}}$$

rule of
consequence

$$\frac{P_1 \Rightarrow P_2, \{P_2\} S \{Q_1\}, Q_1 \Rightarrow Q_2}{\{P_1\} S \{Q_2\}}$$

PROOF RULE FOR ADA LOOP

$$\frac{\{I\} S \{I\}}{\{I\} \text{ loop } S \text{ and loop } I; \{Q\}}$$

$$\frac{(Q \& B) \Rightarrow Q_I}{\{Q\} \text{ exit } I \text{ when } B; \{Q \& B\}}$$

$$\frac{(P \& \text{True}) \Rightarrow P}{\{P\} \text{ exit } I \text{ when true; } \{Q \ P \ \& \ \text{false}\}, P \ \& \ \text{false} \Rightarrow P}$$

$$\frac{\{P\} \text{ exit } I \text{ when true; } \{P\}}{\{P\} \text{ loop exit } I \text{ when true; end loop } I; \{P\}}$$

ew 7886/10

type rec S =

loop of Label x Formula x S |

exit of Formula x Label x Formula |

...

VCs (loop (I, Q, s), post, as) =
{Implies (Q, Pre (s))} u VCs (s) Q, (I, post) u as)

VCs (exit (P, I, B), post, as) =
{Implies (and (P, Not B), Implies (And (P, B), QI))}
when (I, QI) E as

ew7886/11

Define assumption to be (I, Q_I) representing
that Q_I is past condition of loop labeled I

$$\begin{array}{l} \Phi \cup (I, Q_I) \vdash \{I\} \quad S \quad \{I\} \\ \hline \Phi \vdash \{I\} \quad I: \text{ loop } S \text{ end loop } L; \{Q_I\} \\ \\ (Q \& B) \Rightarrow Q_I \\ \hline (I, Q_I) \vdash \{Q\} \text{ exit } I \text{ when } B; \{Q \& B\} \end{array}$$

ew 7886/12

Define: Validity $\Phi \vdash \{P\} S \{Q\}$

$\psi \delta$ P is true in σ & $\sigma' \neq \text{error}$
 $\Rightarrow Q_j$ is true in σ'

where $\langle j_1 \sigma' \rangle = X [[S]] \sigma$
 and

$$Q_j = \begin{cases} Q & \text{if } j = ne \\ Q_1 & \text{if } j = l \text{ for } (l, Q_1) \in \phi \\ \text{false} & \text{ow} \end{cases}$$

COMPLETENESS OF EXIT STATEMENT

Given $(I, Q_I) \vdash \{P\} S \{Q\}$

$(P \ \& \ B) \Rightarrow Q_I$

 $(I, Q_I) \vdash \{P\} \text{ exit } I \text{ when } B; \{P \& B\}, (P \& B) \Rightarrow Q$

$(I, Q_I) \vdash \{P\} \text{ exit } I \text{ when } B; \{Q\}$

ew7886/14

SOUNDNESS OF THE LOOP STATEMENT

Given: $\Phi \cup (L, Q_l) \vdash \{I\} S \{I\}$

Let σ s.t. I is true in σ (and $f_{lp}(\sigma)$ terminates)

$\sigma = \sigma_0 (ne, \sigma_1), (ne, \sigma_2), \dots, (ne, \sigma_{n-1}), (j_1 \sigma_n)$

By induction I is true in σ_{n-1}

if $j=1$ then Q_l is true in σ_n

if $j \neq 1$ then $(j, Q_j) \in \phi$

ew7886/15

COMPLETENESS OF THE LOOP STATEMENT

Given: $\Phi \vdash \{P\} \vdash \{Q\}$

Show: $F \cup \{I_1 Q_1\} \vdash \{I\} S \{I\}$
 where $Q_1 = Q$ and $I = wp(L, Q)$

"half" $\nexists [S] \quad \sigma = (j_1 \sigma') \quad j \neq ne, j = l \sigma(j, Q_j) \in \Phi$
 if $j = l \quad \nexists [L] \quad \sigma = (ne, \sigma_1)$

ow

$\Phi \cup \{I, Q\} \vdash \{I\} L \{Q\}$

=

$\Phi \cup \{I, Q\} \vdash \{I\} S L \{Q\}$

ew7886/16

ATTRIBUTE GRAMMARS

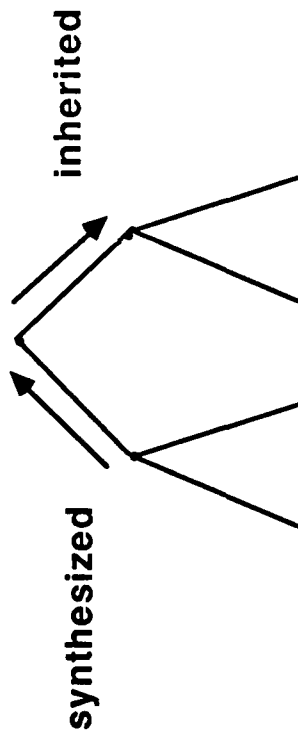
CFG Augmented

Semantic Attributes

Semantic Rules

Two Classes of Attributes

Inherited and Synthesized



1. Easy to Describe VCG
2. Generators Exist
3. Integrates with Compiler Technology

VC GENERATION

$P::= S\{Q\}$

$S::= a;$

assert Q;

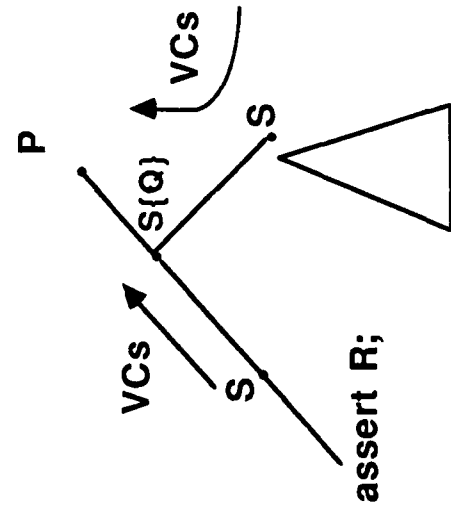
S S

if B then S else S end if;

inv Q while B loop S end loop;

P (program) has synthesized attribute VCs
set of verification conditions

S has inherited attribute post and synthesized
attributes pre and VCs



VC GENERATION FOR D STRUCTURES

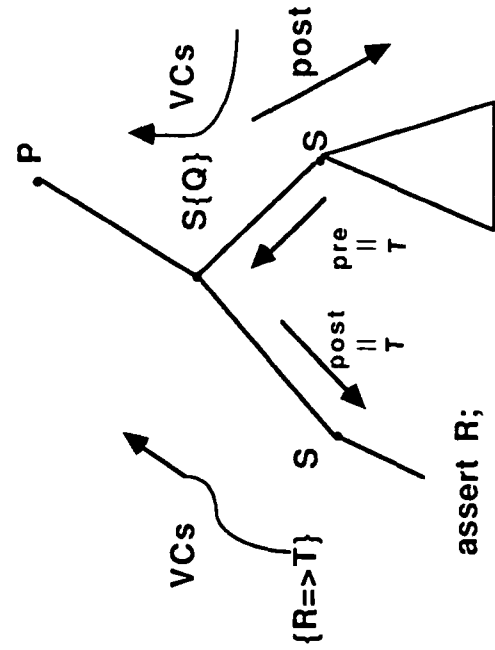
```

P ::= S {Q}
P.VCS := S.VCs
S.post := Q

S ::= a;
S.VCS := ∅
S.pre := f (S.post)

S ::= assert Q;
S.VCS := {Q ==> S.post}
S.pre := Q

S1 ::= S2 S3
S1.VCS := S2.VCS U S3.VCS
S1.pre := S2.pre
S2.post := S3.pre
S3.post := S1.post
    
```



VC GENERATION FOR LOOP CONSTRUCT

Add inherited attribute as for the list of assumptions

$S_1 :: = l: \text{inv } Q \text{ loop } S_2 \text{ end loop } l;$

$S_1.\text{pre} : = Q$
 $S_1.VCs : = \{Q \Rightarrow S_2 \text{ pre}\} \cup S_2 \text{ VCs}$
 $S_2.\text{post} : = Q$
 $S_2.\text{as} : = (l, S_1.\text{post}) \cup S_1 \text{ as}$

$A :: = \text{pre } P \text{ exit } l \text{ when } B;$
 $S.\text{pre} : = P$
 $S.VCs : = \{(P \& \neg B)\} \Rightarrow S.\text{post}, (P \& B) \Rightarrow Q_l\}$
 where $(l, Q_l) \in S.\text{as}$

ew7886/20

SIMULATING AG EVALUATION

- 1. General Method**
 - 2. Requires Recursive Types**
-
- 1. CFG is rendered a type definition**
 - 2. Mutually Recursive Functions Written for Each Synthesized Attribute**
 - 3. Inherited Attributes are Passed as Arguments to Functions.**

REVISITING AXIOMATIC EXCEPTION PROPAGATION

by

Timothy E. Lindquist
Arizona State University
Tempe, Arizona 85287

(602) 965-2783

Lindquis%asu@csnet-relay

May 1, 1986

Abstract

Ada* exception mechanisms have recieved much attention in evaluations of the language's design. In part, critics focus on the dynamic structure used to propagate a raised exception. Axiomatic proof rules for exceptions have already been proposed using an abstraction approach similar to that used for procedures and iteration. The approach augments a program to include specifications describing the particulars of exception declaration and propagation. The goal of this paper is to present an axiomatic system for exceptions that does not require augmented specifications. The reason for constructing such a description is not to ease proofs of programs using exceptions, but is instead to better represent the semantics of exceptions, particularly propagation.

Index Terms -- Ada, exceptions, verification, semantics (CR categories - 4.22, 5.24)

*Ada is a registered Trademark of the U.S.Government, Ada Joint Program Office.

1. INTRODUCTION

Program verification methods are generally useful in areas other than proving software. For example, Dijkstra [1] has shown the benefit in constructing a program, rather than as an analytic tool for an existing program. Further, proof techniques developed for various language features have helped language designers and implementors produce conceptually clearer languages and more uniform implementations.

In this paper we consider axiomatic rules for Ada exception mechanisms. Of special interest are rules to accommodate exceptions raised in a procedure and propagated to the calling environment. In part, our purpose is to demonstrate that axiomatic descriptions might be differently formulated to represent the semantics of exception features than when used in proofs. Our rules are more closely aligned to the semantics of exceptions than to proofs. Since Ada uses the dynamic path through the program to bind a raised exception to a handler, the mechanisms we present elaborate possible execution paths through a program. Thus they provide the basis for describing dynamic identifier binding such as seen in languages APL, SNOBOL, and LISP.

We present mechanisms that are analogous to those presented by Dijkstra in [1], in which concrete mechanisms are first presented and then generalized to allow for proofs. For example, Dijkstra first presents transformers for iteration which don't require induction, and he then generalizes these for arguing about more realistic programs. We present mechanisms which describe exceptions. Luckham and Polak [2] have already described exception mechanisms axiomatically in a form that may be used to argue about programs. They augment programs with specifications describing:

- For every handler, an assertion that will be true whenever one of the exceptions in a when clause is raised, and
- Exception propagation; each exception propagated by a subprogram must be specified by an assertion that is true when the exception is propagated.

Our goal is to have proof rules that don't require additional specifications. Although we realize that augmenting the program with assertions is necessary for verification, a rudimentary description better demonstrates the semantics of exceptions.

1.1 Ada Exceptions

Programs are often required to display normal behavior in the presence of unexpected conditions or failures. So long as the failure does not indicate a flaw in the program logic, termination of the program may not be necessary and processing may continue. Exceptions provide this capability for Ada programs.

Ada Exceptions are declared in the same manner as any other object. A handler, or service routine, must also be defined, and it may exist at the end of any Ada block. When an exceptional condition is signaled, normal execution is diverted to the appropriate handler. Handlers for an exception may be defined in different blocks depending on the scope and visibility rules of Ada. Determining the handler to be used in servicing an exception is done based on the dynamic flow of execution through the program. Further, the point at which execution continues after an exception has been handled depends on which handler is used.

User-defined exceptions are placed in the declarative part of a program unit as in the example:

```
STACK_OVER, STACK_UNDER : exception;
```

In contrast to user-defined exceptions, several are predefined as a part of the standard environment for a program. Built-in exceptions include **constraint_error**, **numeric_error**, **program_error**, and **tasking_error**. The programmer must supply handlers for built-in exceptions, but they are raised automatically by the supporting runtime system (unless suppressed). Exceptions may be explicitly asserted with the **raise** statement. For example:

```
raise ACCESS_VIOLATION;
```

Since handlers are placed only at the end of a block, the handler that will be used for any exception raised in a block does not change during execution of the block. This is one of the key differences between conditions in PL/1 and Ada exceptions. The PL/1 ON-statement may be used to alter the handler for a raised condition at any point during execution.

Determining the handler to be used for a raised exception in Ada and the point where control returns after the handler completes depends on where the exception is raised and whether a handler exists in the block. In the simple case, an exception, say E, is raised in the statement part of the block and a handler (or **others** clause) exists in the exception part. The statements following the **raise** are abandoned in this situation and control returns to the calling program unit.

If there is not a handler or **others** clause in the raising block then the block is abandoned and the exception is raised in the calling block (in the case of a subprogram). Propagation, as this is called, uses the dynamic calling structure of the program to bind a raised exception to a handler. Other cases exist, in which an exception may be raised in the elaboration of the declarative part of a

program, in a rendezvous between tasks or in the execution of a handler. In these instances, the exception is also propagated. Propagation of exceptions is analogous to the dynamic identifier binding strategy used in several interpreted languages.

1.2 Binding Exceptions to Handlers

When an exception is propagated, the nesting of procedure calls determines the order of search for a handler. Suppose the execution of procedure P causes E1 to be raised, but P does not contain a handler for E1. E1 is propagated to all subroutines nested in the call leading to P's execution. Each active procedure, from the most recent backwards, is examined for an appropriate handler. While executing a program, it is common to have more than one active handler for an exception. Further, if a procedure on separate invocations raises the exception E1, it may be handled differently each time it is raised.

Verification of exceptions is complicated by propagation. For example, consider the program L in Figure 1. The procedure R is invoked twice, and each time it raises the exception T. R does not contain a handler for T, and in fact, on successive calls to R propagation results in different handlers being used. Two handlers for the exception T exist the second time the statement

raise T;

in procedure R is executed. One of these handlers is created when execution of L begins; the other when procedure Q is called from L. The procedure demonstrates how different handlers may be used to service a raise statement on different executions of the encompassing procedure.

```

procedure L is;
  T : exception;
  procedure R is;
    begin raise T;
  end R;
  procedure Q is;
    begin call R;
    exception when T => put ("Q's handler used");
  end Q;
  begin call Q;
    call R;
    exception when T => put ("L's handler used");
end L;

```

Figure 1. Dynamically binding exceptions to handlers.

1.3 Proof Strategy

Ideally, a predicate transformer should be formed for each procedure used in a program. In constructing a proof, the transformer should be applied each time the procedure is called, as determined by the proof rule for procedure call. When used in this manner, a procedure's transformer serves as a description of the effect of invoking the procedure, and it appropriately means that the effect of the procedure is the same, withstanding parameters, each time it is called. Axiomatic rules, adhere to the ideal that a procedure is a parameterized abstraction, but this is done through PRE and POST assertions rather than a predicate transformer. When used properly, procedures have the same meaning each time they are called. With propagated exceptions, however, a procedure's meaning may change each time it is called. Aside from parameters, the exception handlers may be different for each call. The concept of PRE and POST assertions for a procedure must be extended to accommodate the changing meaning that propagation may cause.

The question naturally arises: How can a static description method accurately treat exception binding that relies on the dynamic nature of the program? Ada is defined in such a manner that static analysis of code can determine all possible procedure nestings when an exception is raised. The procedure execution history is a sequence of nested procedure calls leading to the statement causing the exception. The history may be determined through static path analysis of the program. Properties of the language allowing this are:

- procedure variables are not permitted in the language,
- procedure references are resolved using scope and visibility rules which are based on the textual structure of the program,
- A single Ada block may define only one handler for any given exception. Although PL/1, in which any number of handlers may be defined in a block, may be statically analyzed, the analysis is simplified by this restriction.

2. BINDINGS FOR AXIOMATIC DESCRIPTIONS

To simplify the rules which follow, we restrict the language in a manner to allow for exceptions in the absence of several other language features. Rules are presented based on a language framework including procedures (without global references), assignment, scalar types and if-then-else selective statements. Exceptions are raised explicitly with the **RAISE** statement, and handlers are defined according to the language.

The rules presented do not deal with exceptions that are raised in elaboration of a declarative region. Further, we assume that a handler will be found for a raised exception. The rules assume that pragma suppress is not used and that built-in and user defined exceptions are enabled.

2.1 Substitution Operator and General Axioms

Simultaneous substitution $R(F_1/X_1, \dots, F_n/X_n)$ is the predicate R with F_i replacing all free occurrences of X_i . The substitutions take place simultaneously; that is, each substitution is not affected by any other, provided that all X_i are distinct.

The general axioms presented here are also adapted from Floyd [3]. These are:

If $\vdash P \{S\} Q$ and $\vdash P' \{S\} Q'$
then Ax1. $\vdash P \text{ and } P' \{S\} Q \text{ and } Q'$
Ax2. $\vdash P \text{ or } P' \{S\} Q \text{ or } Q'$

2.2 Predicates and Continuations

The predicates used in an axiomatic description of a language are first order logic assertions whose free variables are the user-defined identifiers from the program. The predicates characterize the state space of the program. To these predicates we add a characterization of the continuation. A continuation predicate accompanies a statement's post condition to indicate how execution proceeds. The predicate, which is similar to the continuation function of a denotational description [4], characterizes one of three possible continuations:

CONT -- continue

ABAN -- abandon

PROP T -- propagate the exception named T

Predicates over the identifiers of the program characterize continuation states. For example,

$$X + 5 \leq \text{integer}'\text{last}$$

is one of the conjuncts in a predicate we call

$$\text{NOEXCEPT}(X + 5)$$

The predicate characterizes the CONT continuation state for evaluating the expression $X+5$. Post assertions are augmented to include a conjunct characterizing the continuation state. We have constructed the rules in such a way as to allow the post assertion for a command to include possibly many disjuncts. Each disjunct describes a separate execution path through the statement. Disjuncts are made up of a predicate describing the state space for the path and a predicate describing the continuation state.

2.3 Axiom Scheme for Assignment

To provide the distinction between execution paths through the program we use Floyd's strongest verifiable consequent (svc) rule [3] and apply it to axiomatics. The rule takes the form:

$$\vdash P \text{ and } \text{NOEXCEPT}(f(x,y)) \{x:=f(x,y)\}$$

$$\exists x_0(x=f(x_0,y) \text{ and } P(x_0/x)) \text{ and CONT}$$

With the simplified NOEXCEPT given above, we might have as an axiom:

$$\vdash x>5 \text{ and } x+5 \leq \text{integer}'\text{last} \{x:=x+5\}$$

$$\exists x_0(x=x_0+5 \text{ and } x_0>5) \text{ and CONT}$$

which can be simplified to:

$$\vdash x>5 \text{ and } x+5 \leq \text{integer}'\text{last} \{x:=x+5\} \quad x>10 \text{ and CONT}$$

When it is convenient, we will drop the predicate NOEXCEPT from preconditions for brevity and clarity.

2.4 Explicitly Raising an Exception

The predicate transformer for a raise statement that may result in propagating the exception to a nonlocal handler is now presented. To determine which handler of the possibly many active ones will be used, a set of active handlers could be retained for each execution path. Although this approach would have the advantage of reflecting the dynamic declaring environment existing at the

point of raising the exception, Ada semantics define propagation in terms of abandoning the current frame and raising the exception in the calling procedure. To do this our rule for raising an exception checks whether the exception is defined in the block or whether it must be propagated outside the block. Accordingly the continuation state is set.

To determine whether a handler is locally declared, we add an indication of the exception handlers to the language of the system. Thus, if we have:

```
begin
    S0
exception
    when E1 => S1; ...
    when En => Sn;
end;
```

We represent this in proofs as: $P\{S_0 \Delta E_1 \Rightarrow S_1 \dots E_n \Rightarrow S_n\}Q$ and we will abbreviate $E_1 \Rightarrow S_1 \dots E_n \Rightarrow S_n$ as just E_H . If S_0 is compound then we keep E_H for each command in S_0 . If the set of handlers does not enter into the rules for the statement, then it is dropped from the rule. The rules for raising an exception using the set of local handlers as:

if T handled in E_H by S_i and $\vdash P\{S_i \Delta \text{no-handle}\}Q$
 then $\vdash P\{\text{RAISE } T \Delta E\_H\}Q$ and ABAN

According to the rule, if a handler for T is local to the block, then the state space is effected by executing the handler. The continuation state in this case indicates that the remainder of the block is to be abandoned. This state will be passed through any remaining statements in the block. Note that **no-handle** is the local set of handlers for executing the handler S_i . That is to say, if an exception is encountered in handling an exception, then the exception is propagated to the invoking environment.

Alternatively, another rule is needed when the handler is not contained in the excepting block. The following rule causes the exception to be propagated to the calling environment.

if T not handled in E_H
 then $\vdash P\{\text{RAISE } T \Delta E\_H\}P$ and PROP T

When the exception must be propagated to the calling environment the precondition P must also serve as the precondition (possibly modified by environment changes) for the handler eventually found.

Proof Rules for Propogated Ada Exceptions

2.5 Sequential Composition

Sequential composition is no longer sequential composition, at least not referring to sequential execution of statements. That is, when a sequencer is introduced to the language, the semi-colon no longer implies sequential execution. One approach would be to consider commands to be separate from sequencers, and to allow sequential composition to be applied to commands only. The problem with this approach is that implicitly raised exceptions turn a command into a sequencer. Thus, the normal rule for composition must be modified for Ada even without explicit exceptions. Here we present three rules for composition, one for each continuation state. When an exception has been noted (by an ABAN or PROP continuation state), subsequent statements in the block are ignored by the rule.

A. Abandoned execution paths:

If $\vdash P \{S_1\} Q$ and ABAN
then $\vdash P \{S_1; S_2\} Q$ and ABAN

B. Propagate an exception along the path:

If $\vdash P \{S_1\} Q$ and PROP T
then $\vdash P \{S_1; S_2\} Q$ and PROP T

C. Continued statements:

If $\vdash P \{S_1\} Q$ and CONT and $\vdash Q \{S_2\} R$ and CONT
then $\vdash P \{S_1; S_2\} R$ and CONT

2.6 A Selection Rule: if-then-else

The rule for if-then-else is modified to characterize the separate execution paths as done by Floyd [3]. The rule takes the form:

If $\vdash P$ and B $\{S_1\} Q$ and $\vdash P$ and $\sim B \{S_2\} R$
then $\vdash P \{ \text{if } B \text{ then } S_1 \text{ else } S_2 \} Q \text{ or } R$

2.7 An Example of a Locally Handled Exception

Thus far, the rules presented are sufficient to see how explicitly raised exceptions are bound to local handlers. The following example is constructed to demonstrate a block which can conclude after handling an exception or normally.

```
begin
  PRE(x > 5);
  if x = 7 then RAISE T;
             else x := x + 2; endif;
  x := x + 1;
exception
  when T => x := x + 4;
end;
```

We want to show:

$\vdash x > 5 \quad \{\text{if-then-else; } x := x + 1\}$
 $(x > 8 \text{ and } x \neq 10 \text{ and CONT}) \text{ or } (x > 9 \text{ and } x = 11 \text{ and ABAN})$

To argue this we show a proof of each path through the program and then combine them using axiom Ax2.

Since $\vdash x > 5 \text{ and } x = 7 \{x := x + 4 \Delta \text{no-handle}\} x > 9 \text{ and } x = 11 \text{ and CONT}$
we have: $\vdash x > 5 \text{ and } x = 7 \{\text{RAISE T} \Delta \text{E\_H}\} x > 9 \text{ and } x = 11 \text{ and ABAN}$
and for the if: $\vdash x > 5 \text{ and } x = 7 \{\text{if-then-else}\} x > 9 \text{ and } x = 11 \text{ and ABAN}$
(We have removed the "or FALSE and CONT" from the else clause). Applying the ;
 $\vdash x > 5 \text{ and } x = 7 \{\text{if-then-else; } x := x + 1\} x > 9 \text{ and } x = 11 \text{ and ABAN (1)}$

Following a similar argument for the else path we get:

$\vdash x > 5 \text{ and } x \neq 7 \{\text{if-then-else; } x := x + 1\} x > 8 \text{ and } x \neq 10 \text{ and CONT (2)}$

Combining (1) and (2) using axiom Ax 2 provides the desired theorem.

2.7 Propagating Exceptions and Procedure Calls

As can be seen with the above argument, proofs are constructed based upon possible execution paths through the program. Although the rules for procedure invocation do not explicitly show this proof approach (neither does the rule for if-then-else), one can see that we rely on the availability of path information. The path provides access to the nesting structure of procedure calls; thus allowing raised (and propagated) exceptions to be bound to the most recently created handler (dynamic binding). A rule for procedures is now presented to demonstrate handling

propagated exceptions. The rule is simplified by excluding nonlocal references from the procedure and problems arising from aliases created through parameters. Procedures are abstracted through PRE and POST assertions, and we assume that $\text{-- PRE } \{ S \} \text{ POST}$ where S is the statement body of the procedure. POST is assumed to be in the form:

$$(D_1 \text{ and } C_1) \text{ or } (D_2 \text{ and } C_2) \text{ or } \dots \text{ or } (D_n \text{ and } C_n)$$

where: D_i is a predicate characterizing the data state of the computation and

C_i is a predicate characterizing the continuation state.

If we have a call to R , where R is defined with parameter x ;

call $R(a)$;

then the following rule applies:

$$\text{-- } P \text{ and PRE}(a/x) \{ \text{call } R(a) \Delta E\_H \} Q_1 \text{ or } Q_2 \text{ or } \dots \text{ or } Q_n$$

where each Q_i is derived from D_i, C_i according to:

If C_i is **PROP T**

then if T is handled in E_H at j and

$\text{-- } P \text{ and PRE}(a/x) \text{ and } D_i(a/x) \{ S_j \Delta \text{no-handle} \} R_i$

then Q_i is **(R_i and ABAN)**

else if T is not handled in E_H

then Q_i is **(P and $\text{PRE}(a/x)$ and $D_i(a/x)$ and **PROP T**)**

otherwise (C_i is **ABAN** or **CONT**)

Q_i is **($D_i(a/x)$ and **CONT**)**

2.8 An Example of Exception Propagation

We modify the previous example to contain a procedure, R , that does not handle exception T :

```

procedure  $R(x)$  is;
  PRE ( $x > 5$ );
  POST ((  $x > 8$  and  $x \neq 10$  and CONT) or (  $x = 7$  and PROP T));
  begin if  $x = 7$  then RAISE  $T$ ;
        else  $x := x + 2$ ; endif;
         $x := x + 1$ ;
  end R;

```

If we have a call to R as in the block below then the T raised in R should be propagated and handled in the calling block.

```
begin PRE ( a > 6 );
      . . .
      call R ( a );
      . . .
exception
  when T => a := a + 3;
end;
```

We show:

$\vdash a > 6 \{ \text{call } R(a) \Delta E\_H \} (a > 8 \text{ and } a \neq 10 \text{ and } \text{CONT}) \text{ or } (a = 10 \text{ and } \text{ABAN})$

Applying the procedure call rule:

Q_1 is obtained by substituting on the post condition POST as:

$(a > 6 \text{ and } a \neq 10 \text{ and } \text{CONT})$

Q_2 is obtained from the first half of the rule since C_2 is **PROP T** and T is handled

locally: $\vdash a > 6 \text{ and } a > 5 \text{ and } a = 7 \text{ and } \text{NOEXCEPT}(a+3) \{ a := a+3 \Delta \text{no-handle} \}$
 $a = a_0 + 3 \text{ and } a_0 > 6 \text{ and } a_0 > 5 \text{ and } a_0 = 7 \text{ and } \text{CONT}$

i.e., $\vdash a = 7 \{ a := a+3 \Delta \text{no-handle} \} a = 10 \text{ and } \text{CONT}$

So, Q_2 is $(a = 10 \text{ and } \text{ABAN})$ providing our desired result.

3. SUMMARY

An axiomatic approach to describing exception propagation in Ada must be based on the execution paths through a program since a raised exception is bound to the most recently created handler. This paper presents rules which are applied based on execution paths. Predicates describing the continuation state of a command are used to augment post conditions. The continuation state information is propagated through execution paths along with predicates characterizing the data space of a program. An adaptation of Floyd's Strongest Verifiable Consequent (SVC) is used to generate post conditions.

When commands, such as assignment statements, may cause exceptions to invoke specific actions rather than abort in error, typical rules for sequential composition of statements must be modified. In Ada and other languages, such as PL/1, commands (and expressions) that may

generate exceptions must be considered to be sequencing statements. With this possibility, we can no longer define the semi-colon occurring between two commands to imply sequential composition. It must instead revert to being a statement separator (or terminator). In this paper we present rules for commands that provide continuation semantics along with functional semantics. The rule for a semi-colon is modified to account for the possible continuation states resulting from commands (and sequencers).

4. REFERENCES

- [1] Dijkstra, E.W., A Discipline of Programming, Prentice Hall, 1976.
- [2] Luckham, D.C. and W. Polak, "Ada Exception Handling: An Axiomatic Approach," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, April 1980.
- [3] Floyd, R.W. "Assigning Meanings to Programs," *Proceedings of Symposium of Applied Mathematics: Mathematical Aspects of Computer Science*, AMS, Providence RI (1967).
- [4] Tennent, R.D., Principles of Programming Languages, Ed. Hoare, Prentice Hall, 1981.

Revisiting Axiomatic Exception
Propagation

May 14, 1986

presented by:

Timothy E. Lindquist

Arizona State University

(csnet: Lindquis@asu)

Goal:

To have proof rules that don't require additional specifications and which conceptually represent the semantics of exception propagation.

- Propagation uses the dynamic calling structure of the program to bind a raised exception to a handler.
- Propagated exceptions consequently possess many of the properties of dynamically bound identifiers.

Proof Strategy

To allow for the dynamic nature of propagation, we present rules that aid in identifying the execution paths through a program (strongest verifiable consequent).

Continuations for Axiomatic Systems:

Augment predicates characterizing the state space of a program with predicates characterizing the continuation state.

We are concerned with three:

- CONT continue
- ABAN abandon
- PROP T propagate T

Assignment:

$\vdash P \text{ and NOEXCEPT}(f(x,y)) \{x := f(x,y)\}$

$\exists x_0 (x = f(x_0, y) \text{ and } P(x_0/x)) \text{ and CONT}$

NOEXCEPT (f) is a predicate over program

identifiers characterizing the data
states in which f may be evaluated
without generating an exception.

Example:

One of the conjuncts in **NOEXCEPT(X+5)**

is:

$$X + 5 \leq \text{integer 'last}$$

So we might have as an example of assignment:

$\vdash x > 5 \text{ and } \mathbf{NOEXCEPT} (X+5) \{X := X+5\}$

$\exists X_0 (X = X_0 + 5 \text{ and } X_0 > 5) \text{ and } \mathbf{CONT}$

Raising and Exception:

if T handled in E_H by S_i and

$\vdash P \{ S_i \Delta \text{no-handle} \} Q$ and CONT

then

$\vdash P \{ \text{RAISE } T \Delta E\_H \} Q$ and ABAN

if T not handled in E_H

then

$\vdash P \{ \text{RAISE } T \Delta E\_H \} P$ and PROPT

WHEN RAISING an EXCEPTION:

1. E_H is an abbreviation for:
exception when $E_1 \Rightarrow S_1$. . .
 when $E_n \Rightarrow S_n$

end

2. When T is handled in E_H we show:

$\vdash P \{S_i \ \Delta \text{ no\_handle} \} Q \text{ and CONT}$

Which says that if an exception occurs in handling an exception, then propagate it.

3. We abandon the remainder of the statements in any event by
PROPT or ABAN

**WHEN A LANGUAGE PROVIDES IMPLICIT
EXCEPTIONS, SEQUENTIAL COMPOSITION
MUST BE MODIFIED**

**A COMMAND BECOMES A SEQUENCER
WHEN AN IMPLICIT EXCEPTION (THAT
WILL BE HANDLED) IS RAISED**

Sequential Composition:

- a. if $\vdash P \{ S_1 \} Q$ and ABAN
then $\vdash P \{ S_1 ; S_2 \} Q$ and ABAN
- b. if $\vdash P \{ S_1 \} Q$ and PROP T
then $\vdash P \{ S_1 ; S_2 \} Q$ and PROP T
- c. if $\vdash P \{ S_1 \} Q$ and CONT and
 $\vdash Q \{ S_2 \} R$ and CONT
then $\vdash P \{ S_1 ; S_2 \} R$ and CONT

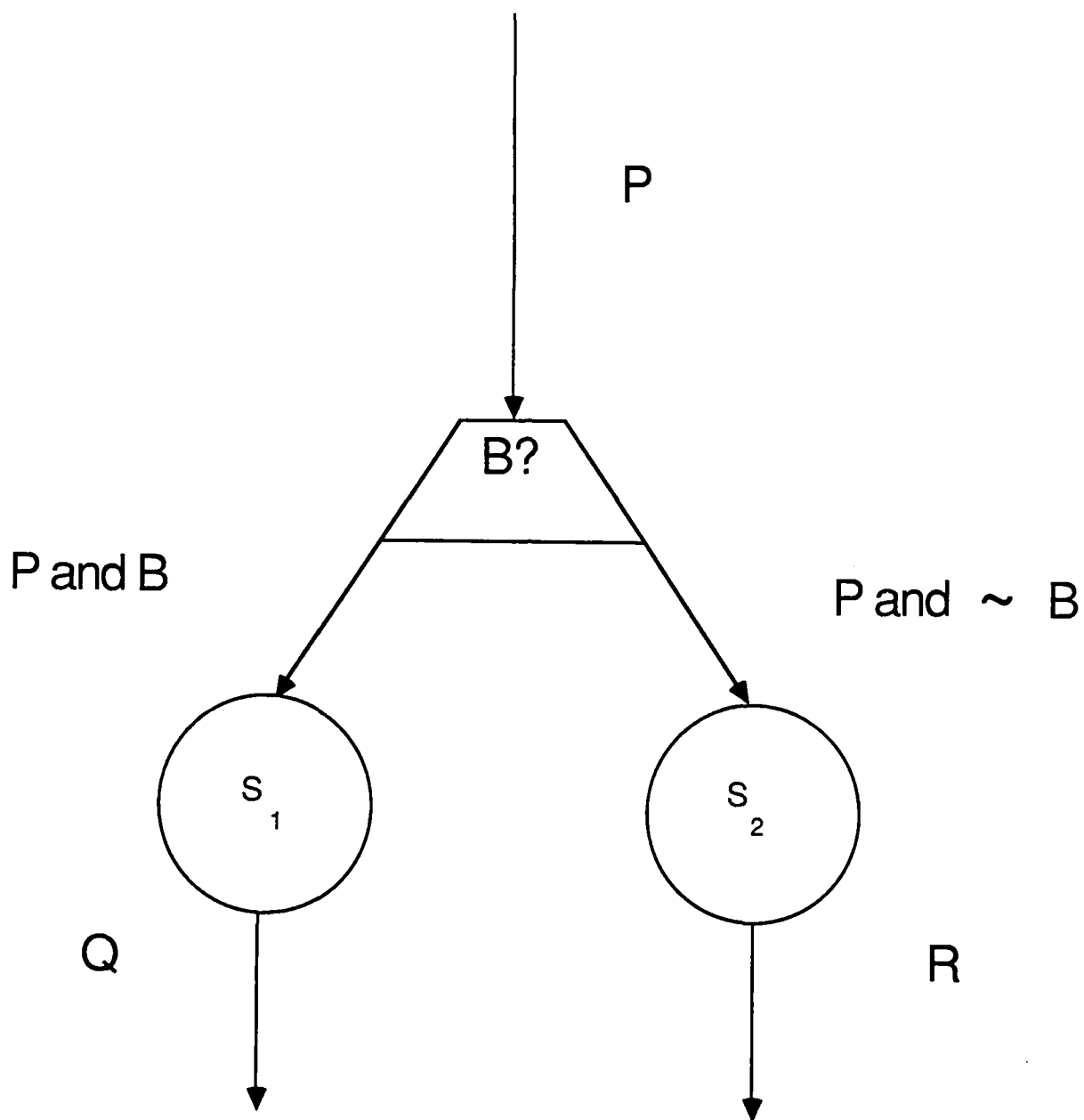
If-then-else

if $\vdash P \text{ and } B \{ S_1 \} Q$ and

$\vdash P \text{ and not } B \{ S_2 \} R$

then

$\vdash P \{ \text{if } B \text{ then } S_1 \text{ else } S_2 \} Q \text{ or } R$



Lindquist (AZ State)

Procedure Calls and Propagation:

Abstract procedures through PRE and POST assertions. Also assume that POST is of form:

$(D_1 \text{ and } C_1) \text{ or } \dots \text{ or } (D_n \text{ and } C_n)$

where: D_i characterizes the data state,
 C_i characterizes the continuation

if we have a call to procedure R with parameter x then the following rule applies:

$\vdash P \text{ and } \text{PRE}(a/x) \{ \text{call } R(a) \Delta E\_H \}$

$Q_1 \text{ or } Q_2 \text{ or } \dots \text{ or } Q_n$

where each Q_i is derived from D_i, C_i as:

if C_i is PROP T

then if T is handled in E_H at j and

$\vdash P \text{ and } \text{PRE}(a/x) \text{ and } D_i(a/x)$

$\{ S_j \Delta \text{no-handle} \} R_i$

then Q_i is $(R_i \text{ and ABAN})$

else if T is not handled in E_H

then Q_i is $(P \text{ and } \text{PRE}(a/x) \text{ and}$

$D_i(a/x) \text{ and PROP T})$

otherwise (C_j is ABAN or CONT)

Q_j is ($D_j(a/x)$ and CONT)

Summary

- sequential composition must change when commands become sequencers as in Ada.
- strongest verifiable consequent allows easier identification of execution paths — thus providing dynamic propagation

Program Development by Specification and Transformation

(PROSPECTRA)

European Strategic Programme for Research in Information Technology

copyright 1985 by

P R O S P E C T R A P r o j e c t

Universität Bremen

Universität Dortmund

Universität Passau

Universität des Saarlandes

University of Strathclyde

SYSECA Logiciel

SYSTEM KG Dr. Winterstein

**under the ESPRIT Programme of the
Commission of the European Communities**

CONTENTS

1	Title	6-7
2	Summary	6-7
3	Objectives	6-8
4	Compliance with ESPRIT	6-10
5	State of the Art	6-12
6	Project Description	6-16
6.1	Methodological Framework.....	6-16
6.2	Formal Basis.....	6-20
6.3	Transformation in Ada/Anna.....	6-20
6.4	System Overview.....	6-21
6.5	User Interaction with the System.....	6-22
6.6	System Development Components.....	6-25
6.7	Transformation Development Components.....	6-29
6.8	Ada/Anna Development Components.....	6-29
6.9	Control Components.....	6-31
7	Project Consortium	6-32
7.1	Project Structure.....	6-32
7.2	Contribution of each Partner.....	6-33

Sections 1 to 6 of this PROSPECTRA Project Summary are taken from the Technical Annex of the contract between the Partners and the Commission of the European Communities (ESPRIT Project # 390).

1 Title

PROgram development by SPECification and TRAnsformation.

2 Summary

The PROSPECTRA project shall provide a rigorous methodology for developing correct software and a comprehensive support system.

The **methodology** shall allow the integration of program construction and validation during the development process. Customer and implementor start with a formal specification, the interface or "contract". This initial specification can then gradually be transformed into an optimized machine-oriented executable program. The final version is obtained by stepwise application of transformation rules. These are carried out by the system, guided interactively by the implementor or automatically by compact transformation tools.

The final version is correct by construction: only the applicability of transformation rules needs to be validated at each step, assisted by the system.

Transformation rules are proved correct once and for all. They shall form the nucleus of an extendible knowledge base, the method bank, together with pre-fabricated program components, previous program versions, and entire development histories that can be replayed.

The strict methodology of Program Development by Transformation shall be completely supported and controlled by the system, enabling the construction of "a priori" correct programs from formal specifications. However, the system shall also allow other program development styles where the user assumes responsibility for unguarded development transitions. Moreover, it shall be possible to integrate existing program components based on their specification, and to develop them further.

The **system** shall comprise basic components for the application of individual transformation rules and of compact development methods described as transformation scripts; these will provide its real power. Any kind of system activity is conceptually and technically regarded as a transformation of a "program" at one of the system layers. This provides for a uniform user interface, reduces system complexity, and shall allow the construction of system components in a highly generative way.

Choosing Ada/Anna as a standard language, and standard tool interfaces (DIANA, CAIS, PCTE), shall ensure portability of the system as well as of the newly developed software.

3 Objectives

Correct Programs

Current software developments are characterized by ad-hoc techniques, chronic failure to meet deadlines because of inability to manage complexity, and unreliability of software products. The major objective of the PROSPECTRA project is to provide a technological basis for developing correct programs.

This will be achieved by a methodology that starts from a formal specification and integrates verification into the development process. Complexity is managed by abstraction, modularization and stepwise transformation. Programs need no further debugging; they are correct by construction w.r.t. the initial specification. Adaptive maintenance is greatly facilitated by replay of developments.

Rigorous Methodology

The envisaged methodology for program development shall be sufficiently rigorous, on a solid formal basis, to allow validation of correctness during the complete development process. It is deemed to be more realistic than the conventional style of a posteriori verification: the construction process and the validation process are broken down into manageable steps; both are coordinated and integrated into an implementation process by stepwise transformation that guarantees a priori correctness w.r.t. the original specification. Efficiency considerations and machine-oriented implementation detail come in by conscious design decisions from the implementor when applying pre-conceived transformation rules. A long-term research aim is the incorporation of goal orientation into the development process. In particular, the crucial selection in large libraries of rules has to reflect the reasoning process in the development.

Formal Specification

Formal specification shall be the foundation of the development to enable the use of formal methods. Existing specification techniques shall be consolidated and made amenable to mechanical verification. High-level development of specifications and abstract implementations (a variation of "logic programming") is seen as the central "programming" activity in the future.

Uniform Language Spectrum

Development by transformation receives increased attention world-wide. However, it has mostly been applied to research languages. Instantiating the general methodology and the support system to Ada and Anna (its complement for formal specification) shall make it realistic for systems development including concurrency aspects. Ada/Anna taken together cover the complete spectrum of language levels from formal specifications and applicative implementations to imperative and machine-dependent representations. Uniformity of the language enables uniformity of the transformation methodology and its formal basis. It is hoped that the complexity of Ada itself will also become more manageable. Stepwise transformations synthesize Ada programs such that many detailed language rules necessary to achieve reliability in direct Ada programming are obeyed by construction and need not concern the program developer.

In its present extent, the project will concentrate on specification and implementation development at the applicative level and the generation of imperative versions. Optimization transformations at the imperative level, analogous to those of a conventional optimizing compiler, will be of less concern.

Method Bank as Knowledge Base

Individual transformation rules, compact automated transformation scripts and advanced transformation methods shall be developed for Ada/Anna to form the kernel of an extendible method bank. It shall thus embody some formalized

"programming" knowledge and expertise analogously to a handbook of physics. It is also expected to provide initial support for goal-oriented selection of rules and scripts. Presently, the PROSPECTRA project does not yet incorporate the development of a full scale expert system for program development; this may well be an objective for the future.

Comprehensive System Support

Support of the methodology by a powerful system is essential. It shall consist of a comprehensive set of interrelated components forming an advanced Ada Program development Support Environment. Existing environments only support the conventional activities of edit, compile, execute, debug. Existing transformation systems are mostly experimental and hardly have production quality in user interface, efficient transformation or library support. Conventional verification systems are monolithic and only support a posteriori verification. The support of correct and efficient transformations is seen as a major advance in programming environment technology.

Reduction of Systems Complexity

The central concept of system activity shall be the application of transformations to trees. Generator components shall be employed to incorporate the hierarchical multi-language approach, to construct transformers for individual transformation rules and to incorporate the hierarchical multi-language approach. Generators increase flexibility and avoid duplication of efforts; thus the overall systems complexity is significantly reduced.

Perspicuous User Interface

Reduction of system complexity shall also imply a reduction of the diversity of user interactions. A small number of uniformly applicable commands shall form the basis for communication with the system. The user interface shall benefit from high-resolution display technology to present a hierarchy of rules, program versions, catalogues, commands, and development histories in a perspicuous form on appropriate windows manageable in parallel. Each such object shall have a uniform internal representation as a tree with different classes of attributes. However, the user shall always operate on the particular intuitive external form in which the tree is paraphrased (a text, a menu).

Efficient Transformation

Transformers shall be generated for (classes of) rules and scripts. In analogy to LALR parser generators, the tree transformer generator shall analyze the properties of a rule in the context of other rules to compute application strategies etc. at transformer generation time. This allows a significant increase in efficiency at transformer execution time, in particular for scripts, i.e. sets of rules to be applied together. In existing systems, applicability conditions for rules are either expressed purely syntactically or as verification conditions about the context to be proved on the side. In the proposed system, applicability conditions shall be described in terms of semantic attributes. This is appealing from a conceptual point of view since it relates to the well-understood notion of attribute grammars. On the implementation side, a considerable increase in efficiency of transformation application can be expected, as context information is available locally and can be incrementally updated.

Transformation Scripts

The transition from collections of individual rules to scripts is a major step forward in the mechanization of transformation descriptions. Rules shall be described uniformly, whether used individually or incorporated into a script. Optional application strategies shall be furnished separately. Scripts can be seen as a structured breakdown of monolithic optimizers; they can be applied individually under the methodology, guided interactively by the user.

4 Compliance with ESPRIT

Engineering Discipline

The PROSPECTRA project aims at making software development an engineering discipline. In the development process, ad hoc techniques shall be replaced by the proposed uniform and coherent methodology, covering the complete development cycle. Programming techniques shall be formalized as transformation rules and methods with the same rigor as engineering calculus and construction methods, on a solid theoretical basis. Rules in the method bank shall be proved correct once and for all and shall thus allow a high degree of confidence. Since the methodology can be completely controlled by the system, reliability is significantly improved and higher quality can be expected.

Specification

Formal specification of requirements, interfaces and abstract designs (including concurrency) shall relieve the programmer from unnecessary detail at an early stage. Detail comes in by gradual optimizing transformation, but only where necessary for efficiency reasons. Validation by formal verification is integrated into the construction process. Specifications are the basis for adaptations in evolving systems, with possible replay of the implementation.

Programming Language Standard

Ada will become central for a common European technology base. Dedicated to embedded information systems, it allows the integration of HW and SW solutions. Complemented by Anna, it covers the complete spectrum from non-imperative specification and design to machine-oriented representations where required.

Research Consolidation

Research in language design and methodology has traditionally come from Europe; strong expertise in formal methods is concentrated here and has had considerable international influence. It is of strategic importance to encourage research and retain its potential in Europe. The PROSPECTRA project shall contribute to the technology transfer from academia to industry by consolidating converging research in formal methods, specification and non-imperative "logic" programming, stepwise verification, formalized implementation techniques, transformation systems, and human interfaces.

Industry of Software Components

The portability of Ada allows pre-fabrication of software components. This is explicitly supported by the methodology. A component is catalogued on the basis of its interface. Formal specification in Anna gives the complete semantics as observable by the user; the implementation is hidden and may remain a company secret of the producer. Ada/Anna and the methodology emphasize the pre-fabrication of generic, universally usable components that can be instantiated according to need. This will invariably cut down production costs by avoiding duplicate efforts. The production of perhaps small but universally marketable components on a common technology base will not only foster a European market but also assist smaller companies in Europe.

Tool Environment

Emphasis on the development of a comprehensive support system is mandatory to make the methodology realistic. The system can be seen as an integrated set of tools based on a minimal Ada Program Support Environment. As such the system shall be compatible by adhering to emerging interface standards such as DIANA, CAIS and the planned ESPRIT Portable Common Tool Environment. Because of the generative nature of system components, adaptation to future languages shall be comparatively easy.

In relation to the ESPRIT 1984 Workplan, the following R&D topics are covered:

2.1: general, and type B projects:

- practical and disciplined system development methods: a coherent methodology as well as individual methods, e.g. interface transformation,
- effective methods of software production and maintenance: a coherent methodology covering software production as well as maintenance; the latter does not require debugging, but allows adaptation and evolution of systems by changes to the original specification and replay of developments, also including:
 - use of existing components in new developments: based on Ada packages and Anna specifications
 - formal semantics of interfaces: algebraic specification in Anna related to formal semantic model
 - (validation and) verification: of implementations against specification; of applicability conditions
 - capturing of requirements: as (incomplete) algebraic requirements specifications
 - development of notations with well-founded semantics: multi-language paraphrasing of Ada/Anna programs, transformation rules, etc.
 - reliability of specifications: formal specifications
 - specification of sequential and concurrent systems: Ada and extension of Anna for specification of concurrent systems
 - decomposition, integration, compatibility of HW/SW subsystems ...: via Ada/Anna package specifications
 - HW/SW migration: possible based on specification
 - system optimization: via optimizing transformations
- representation and transformation tools (major emphasis here), includes multi-level, attribute oriented paraphrasing on screen
- verification and (validation) tools: verification of Anna Logic and verification conditions,
- component library support tools: for Ada/Anna program components and also for transformation rules, scripts, methods and development histories,
- configuration management tools: interface to PCTE for version management etc.
- documentation tools: implicitly, since specifications, transformations, in fact whole developments are recorded

5 State of the Art

Methodological Framework

Although program development by stepwise transformation has attracted considerable interest and substantial work has been carried out by various groups (e.g. Darlington/Burstall, CIP/Munich, PECOS/Barstow, ZAP/Feather, POPART/Wile) no production-level system to support this method has yet appeared (see "Program Transformation Systems" by H. Partsch, R. Steinbrüggen, *Computing Surveys* 15:3).

Experiments with prototype systems showed clearly that the problem of systematically using a large collection of transformation rules has to be solved. The problem is to structure the transformation bank in such a way as to reflect the systematic, goal-oriented reasoning necessary to select a transformation. It should then be possible to automatically support each development step in an effective way without abolishing the guiding intuition of the programmer.

The problem of structuring collections of transformation rules combined with an appropriate strategy for application is related to the work on optimizing compilers (e.g. PQCC/Wulf). No notation for rule scripts has yet been developed that combines the elegance of individual rules with efficient applicability.

As concerns specification development, in particular development methods, the work on CLEAR (putting theories together) by Burstall, Goguen is most relevant as an attempt to make the specification activity a constructive development process.

Formal Basis

In the past, considerable progress has been made in the development of a formal basis for software construction. Denotational semantics, assertion logics, algebraic specifications, transformational semantics have been major areas of successful research (cf. Formal Methods Appraisal study for the EC). Practical outcome of the theoretical work are program support systems connected with functional programming languages, program verification, specification languages and prototypers, and program transformation systems. Not all aspects of programming and program development, however, are equally well developed. In particular, the semantics of concurrent communicating programs is a topic of current research. It seems, however, as if research in this area is just beginning to converge towards a few basic principles such as observability concepts for streams of communication actions. Moreover, Hehner and Hoare have recently proposed methods for specification of such communicating processes.

What is missing at present is a consolidation of the various formal models into a formal framework that can serve as a uniform basis for a practical software development system that provides support for all development phases from the initial specification to final implementation. The correctness of transformation rules can then be verified against the formal model framework.

Transformation in Ada/Anna

Some catalogues of transformations have been assembled for various high-level languages. Of particular interest is the structured approach of the CIP group. The program development language CIPL is formally defined by transformational semantics, mapping all constructs in the wide spectrum of the language to an applicative language kernel that is defined denotationally. These basic transformation rules have an axiomatic nature: compact rules for program development

can be derived from them in a formal way.

Most of Anna is defined by transformational semantics mapping elaborate annotations into simple assertions.

Generative Approach to Program Transformation

The system as proposed needs a generative approach to the implementation of transformations. Transformation rules as given by the designers and incrementally added by the programmer will automatically be implemented by an adequate set of generators.

For a clear interface, several description mechanisms are needed. Firstly, the correspondence between string and tree representations of programs has to be described to allow for generation of parsing and pretty-printing. Secondly, the basic transformation rules as target for the translation of program development steps have to be supported by a generative system.

State-of-the-art generative techniques for the automatic implementation of transformers offer:

- efficient checking for rule applicability
- efficient restoring of consistency after transformation.

At least the following three systems are relevant:

- the MENTOR system at INRIA
- the Cornell synthesizer generator and
- the OPTRAN system at Saarbrücken.

The MENTOR system was designed for (PASCAL)-program development, but has later on been extended to a generating system (which is also the case for the Cornell system). Transformation rules must be explicitly "programmed" by the user who takes care of the collecting of context information and the resolution of conflicts for rule application.

The Cornell synthesizer generator produces program development tools with much emphasis on editing. It has no description mechanism for arbitrary program transformations.

The OPTRAN system is the basis for the work done in the PROSPECTRA project. It has been designed to provide generative support for transformations on programs represented as attributed trees. It contains generators for efficient tree pattern matching and attribute reevaluation. The tree pattern matcher generator works incrementally and thus allows for addition of rules at any time. It also provides for transformation scripts ("T-units"), i.e. collections of transformation rules with an application strategy. So far, these have only been implemented in a rudimentary form.

OPTRAN was designed for batch mode application (apply rules from a given set as long any rule is applicable, restricted only by a user supplied strategy), while MENTOR and the Cornell synthesizer offer an interactive user interface.

User Interaction

The development of powerful personal workstation computers with high-resolution graphic display and pointing devices (e.g. Star, Dorado, Lisa, Lillith) together with highly interactive user interfaces and program environments (Smalltalk (TM), Lisa Tool Kit, Modula-2 Environment, XS3) has set new standards of user interaction. Common features are windows for pursuing distinct activities in parallel, and uniform commands to edit data, evoke system activities, select parameters of commands, manipulate windows, etc. Some systems, e.g. Nievergelt's Modula-2 environment XS3, are even designed by relating user interaction to the underlying principles of computation. Others support language-dependent generation of user interfaces (MENTOR, Cornell Program Synthesizer, ALOE/Feiler).

Important aspects are not covered by today's systems. Firstly, a program system, if it is designed according to the principles of modularity and information hiding, is hierarchically structured. This means that user commands have arguments that may result from computations performed on lower levels of the system. The principles of abstraction and information hiding should be reflected in the system's ways of guiding user interaction through the levels in the hierarchy. Secondly, in an interactive system, the user manipulates the internal state of the system by inputting new data or evoking commands. The presentation of the resulting system state (input parameters of commands, results of calls, internal module state) is, however, the responsibility of each single module. Uniformity of externally presenting internal data in a way that suits their intuitive meaning is therefore not guaranteed. (In the following we will call the process of external representation of internal data **paraphrasing**.) An interactive framework for a program system should therefore provide for mechanisms that generate paraphraser for internal representations of data from high-level descriptions. These descriptions have to be based on the general principle of computation that underlies the various system activities.

Generators for Editors and Paraphraser

As far as syntactic issues are concerned, the generation of syntax oriented editors from context-free grammars is well-understood. Editing of semantic information represented in terms of attributes in abstract syntax trees is not supported by conventional systems, in particular, if attributes may themselves contain program fragments of other languages represented as trees. For example, an Ada/Anna program would contain Anna fragments to annotate the underlying Ada program as tree attributes of this Ada program. An important technical issue is that of incrementally updating the formatted representation during editing. Techniques of incremental attribute evaluation as employed in the Cornell Program Synthesizer seem to be sufficiently powerful to deal with this problem.

Editing of trees is always closely connected to paraphrasing (formatting), as the user shall be allowed to act (select nodes and operations) in terms of an intuitive external representation of the tree - its paraphrased version. Yet it is a separate logical process. The way in which editing commands are recognized as user activation of input devices, or in which edited trees are paraphrased on the screen by evoking operations of a virtual I/O-driver is completely independent of the editing process itself. This conceptual distinction is not clearly made in existing editor/formatter generators (MENTOR, Cornell Program Synthesizer). As a consequence, traversing of trees to select nodes can be quite awkward in these systems. On the other hand, decoding of input activities into editor commands and "encoding" trees into their 2D representation are dual to each other. A paraphrasing description should therefore contain enough information to allow the system to generate both the decoder and the encoder (formatter, paraphraser) in parallel.

Verifiers

Automatic verification systems do exist but the most successful of these are not distributed due to severely restricted circulation because of the commercial or national security considerations resulting from the obvious benefits in terms of reliability.

Existing systems tend to be designed along the following lines. Starting from a given specification and a given program, a verification condition generator will produce a set of logical expressions whose truth would guarantee the correctness of the program. The processing of these logical expressions is then accomplished by a theorem prover (perhaps using an algebraic simplifier) in an attempt to establish the truth of the logical expressions. Failure to do so may result in changes to the program or to the specification. An iterative process then commences, culminating hopefully in establishing correctness.

There is general agreement that these early systems had considerable deficiencies, partly due to the difficult theoretical problems which dictate that the verification process cannot be completely automated. Further shortcomings stem from the fact that the user interfaces were traditionally very poor: should the theorem prover fail to prove that a verification condition is true, a user would typically be confronted with an awesome expression which bears little relation to the original program; small changes to a program would require to starting the complete verification process again.

The advances of a verifier integrated into the process of program development by transformation would be several. Firstly, since verification conditions are produced by means of a sequence of transformations and since the system possesses a mechanism for remembering which transformations have been performed, it becomes a simple matter to relate errors back to the original programs in a user-friendly manner. Secondly, an adjustment to one section of the program will result in the reprocessing of only that section and other affected sections. Thirdly, theorem provers that interact to enlist the help of the user offer substantial advantages both in terms of their efficiency and their capability over alternative approaches.

6. Project Description

Before the particular research and development subjects and phases in the project are described in the work plan (section 7), an overview shall be presented here, giving some background and motivation.

6.1 Methodological Framework

The Development Process

Consider a simple model of the major development activities in the life of a program:

(1) pre-development phase:

ANALYSIS of requirements and informal problem definition

(2) development phases:

SPECIFICATION of the problem (formal requirement specification)

- interface between customer and implementor, the "contract"
- prototype modelling allows informal presentation to the customer
- formalization allows rigorous validation of implementations
- restriction to necessary requirements leaves design choices open

IMPLEMENTATION

by decomposition ("top-down" hierarchy)

- design:
definition of a model; specification of components
- validation:
of model design specification against interface
- construction:
by (recursive!) implementation of components
- installation:
of components by integration

by instantiation ("bottom_up")

- design:
selection of pre-fabricated components from stock
- construction:
by specialization / parameterization
- validation:
of instantiated component specification against interface
- installation:
of instantiated component

(3) post-development phase:

EVOLUTION in response to changes in requirements

- evaluation, inducing changes in requirements
- leads to re-development, starting with changes in the specification
- requires re-implementation, possibly by replay and adaptation of previous implementations or of previously discarded variants

In a rigorous methodology based on formal specification and formal validation, explicit testing of implementations is not required; it is substituted by verification of correctness. Note, however, that the formal specification needs to be accepted by the customer. Prototyping is a way to provide for acceptance tests early in the development process.

Dimensions of Development

In adaptation of the conventional view of a "life-cycle", one can distinguish several dimensions along which the program development activities take place (see fig. 1):

(1) revision (global "cycle")

change of a specification/implementation to adapt to new requirements

(2) variation (local "cycle")

alternate implementation for the same interface specification

(2a) decomposition (hierarchy of recursive developments)

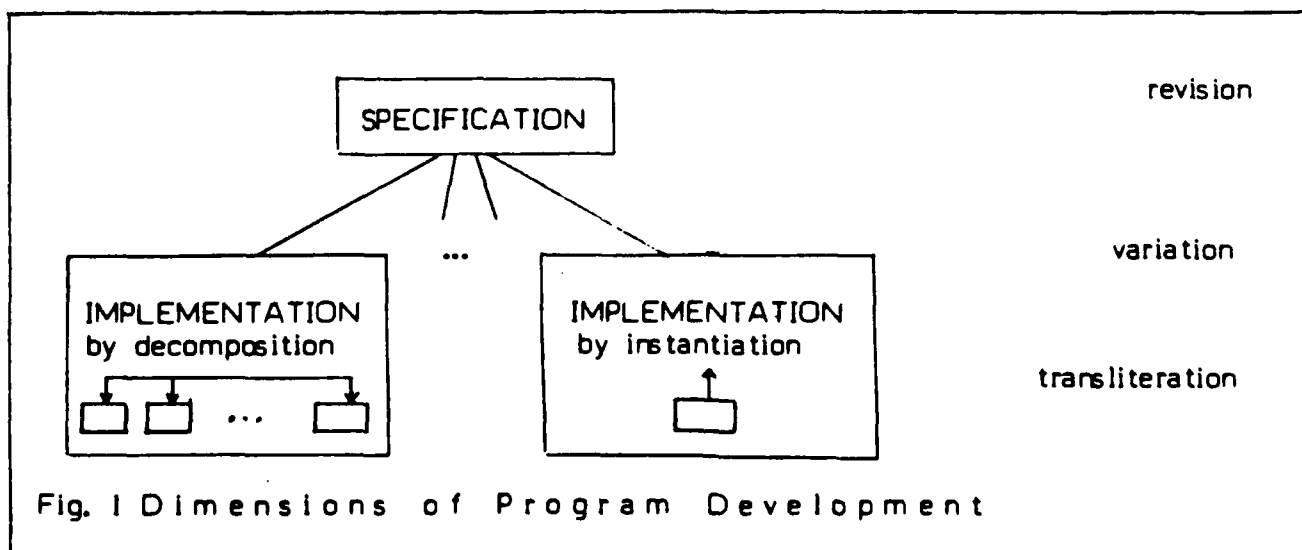
specification and implementation of components

(2b) abstraction/instantiation (pre-fabrication and use)

generalization/parameterization of components to/from stock

(3) transliteration (orthogonal local "cycle")

transformation, possibly to different language style



Although transliteration is, in a sense, a form of variation of an implementation, a conceptual difference is made here. Transliteration does not invalidate previous design decisions as a variation would. It may make the design more detailed, and translate into a more machine-oriented language style or a different implementation language. A conventional compilation is a transliteration in this sense. Similarly, a specification can be transliterated into a different language style.

As a consequence of this view of development activities, one can distinguish several relationships between program versions:

- revision of specification/implementation
- variation of implementation for fixed specification
- component of decomposed implementation
- instance of specification/implementation
- abstraction of specification/implementation (converse)
- transliteration of specification/implementation

Development by Transformation

Each transition from one program version to another can be regarded as a transformation in an abstract sense. It has a more technical meaning here: a transformation is a development step producing a new program version by application of an individual transformation rule, a compact transformation tool, or, more generally, a transformation method invoking these. Before we come to the latter two, the basic approach shall be described in terms of the transformation rule concept.

A transformation rule is a schema for an atomic development step that has been pre-conceived and is universally trusted, analogously to a proof rule in mathematics. It embodies a grain of expertise that can be transferred to a new development. Its application realizes this transfer and formalizes the development process.

Not only is the program construction process formalized and structured into individual mechanizable steps, but the validation process is structured as well and becomes more manageable. If transformation rules are correctness-preserving, then only the applicability of each individual rule needs to be verified step by step. Thus a major part of the validation, the verification of the correctness of each rule, can be done once and for all. Validation reduces to verification of the applicability of a rule, and program versions are correct by construction (w.r.t. the correctness of the original version). The design activity consists in the selection of an appropriate rule, oriented by development goals, for example machine-oriented optimization criteria. Attention is confined to those portions where further development seems to be worthwhile.

The approach of program development by transformation has its obvious use in the transliteration activity. The programmer designs a first, high-level, abstract program version and proves its correctness w.r.t. the specification. This proof can be expected to be easier than the proof of the final version by orders of magnitude. The implementor then refines this version step by step through the application of transformation rules until the development goals are

satisfied. This way, the program may well be transliterated from a recursive version in an applicative language style via a procedural version with loops and variables to a final machine-oriented version involving jumps and registers.

It is likely that the approach will also significantly support the variation activity by formalizing implementations, and choices thereof, as abstract data type transformations.

It is less obvious to what extent the transformational approach can be used to assist the revision activity. Can the development of specifications be formalized by transformation rules? For example the "merge" of two specifications? This will be a subject of research.

Transformation Rules and Transformation Scripts

The methodology of program development by transformation needs powerful system support to be realistic, as shall be described below in sections 6.4 to sections 6.8. The programmer should be able to guide the transformation process interactively, with full attention to detail. However, explicit application of a lot of individual rules is much too tedious in general.

One direction for automation is the development of compact transformation tools that mechanize complex transformations. These may be explicitly programmed for their task. It is more desirable to generate them from transformation scripts, that is a description of a collection of transformation rules complemented by a strategy for their application. The definition of a language for defining transformation scripts will be a R&D task.

In addition, the support of the design activity in this context, that is the goal-oriented selection of transformation rules from a library, is an important research item. More generally, the long term research goal is to develop transformation methods that relieve the programmer from considerations about individual rules to concentrate on the goal oriented design activity. Some of these methods may be quite application oriented, for example to develop programs with strong concern for properties of concurrency and real time.

The Method Bank

Such transformation methods shall be collected in a method bank. Initially, it will contain an extendable library of transformation rules and tools that embody the expertise about the program development process gathered so far. One can compare it to an encyclopedia for mathematical methods for engineers. It can not be expected that a universal closed method will be found, just as there is no single closed formula for the solution of differential equations.

Apart from this general portion of the Method Bank there shall be individual portions for archiving program versions and histories of previous developments. Replay of developments may make an adaptation of previous versions possible during a revision, depending on the nature of the changes. Analysis of development histories may also allow a suitable abstraction and generalization of a development to a method for future use. It will be a matter of research to what extent this is possible.

6.2 Formal Basis

Semantic Foundation of the Programming Methodology

A coherent programming development methodology that employs formal methods such as specification, transformation, verification has to be based on a common semantic basis. Such a semantic model framework has to be selected. Its appropriateness will have to be demonstrated, as well as extendability to concurrency. Algebraic specifications have to be embedded into the model. Relations to the formal semantics of Ada/Anna will be established. Moreover, the correctness of transformation rules and proof rules will be demonstrated in the model.

Algebraic Specification

Algebraic specifications of data structures, models or theories consist of a description of a signature (a family of sorts and operation symbols) and of a number of axioms. Algebraic specifications have proved to be a rather flexible and powerful tool. The extensive theoretical work in this area has to be consolidated. An adaptation to the overall program development methodology is required. The semantics of Ada/Anna constructs, in particular that of "package specifications" has to be defined in terms of the thus obtained algebraic model. Further required is the definition of a implementation concept that provides for the construction and verification of concrete implementations for algebraic specifications. Moreover, frequently used abstract data types, among them the predefined Ada types, shall be constructed once and for all.

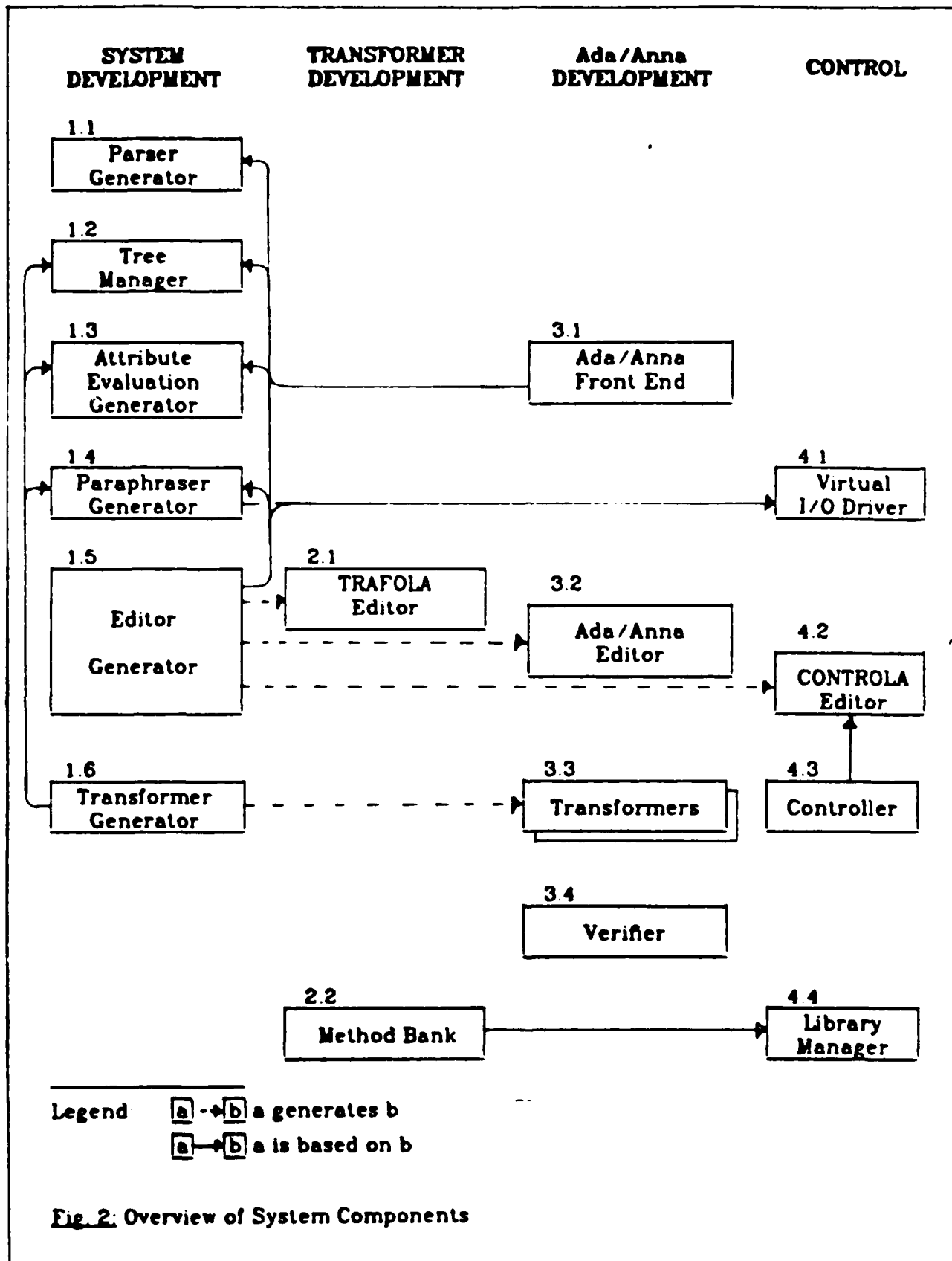
Concurrency

Concurrent communicating systems add a further dimension to programming. The possible combinations of communication actions can be combinatorially complex, errors may in general not be reproduced. Therefore, absence of formal models and techniques leads to a much higher degree of unreliability in the software produced. On the other hand, concurrent systems are often embedded in environments where reliability is of utmost importance (process control etc.). The formal foundation of concurrent systems is much less advanced than that of sequential programs. Fortunately, progress has been made in this area recently. There is hope that these results can be applied to formally support a methodology of concurrent system design.

In the project, existing approaches have to be consolidated and refined. A conceptual model has to be defined on which specification concepts can be based. A notation for specification of concurrency has to be defined as an extension of Anna. Furthermore, the semantics of specialized Ada/Anna constructs will be defined in terms of the model, for example restricted forms of tasks such as monitors, that is those that can be safely developed from specifications.

6.3 Transformation in Ada/Anna

A stock of basic Ada/Anna transformation rules and methods has to be the basis for program development in the system. Some phases and methods of program development (e.g. recursion removal) are well-understood. Work in these areas will therefore concentrate on the adaptation of well-known basic transformation rules and methods to Ada/Anna from other languages. Whereas basic transformation rules have to be correct in terms of the underlying formal semantic model of specification or computation, the correctness of derived transformation rules can be inferred from the correctness of the basic ones. These derived rules /scripts will then be used in program development by transformation.



6.5 User Interaction with the System

The system as proposed provides for various kinds of user activity. The principal goal is to develop and implement a uniform concept of user interaction. As the methodology for program development in the system will be that of transforming programs and specifications by applying transformation rules, it seems natural to view user interaction on each system level as invocation of manipulations on (attributed) trees. It seems that the concept of transformation (as tree manipulation) can be generalized to all processes in the system initiated by user interaction, at various nested levels. This view subsumes command "sequences" in the conventional sense.

Levels of Interaction

A particular system level is, then, characterized by four kinds of objects:

- T. A language T of trees specifies the class of trees to which transformations can be applied on that level.
- A. The atomic elements (leaves) A in these trees represent the interface to the next lower levels in the system. Atomic elements are, therefore, themselves trees of lower-level languages. Manipulating an atomic object on a level L then means to enter a lower level L' of the system.
- P. The trees on any level have a level-specific meaning. This meaning should be indicated to the user by an adequate paraphrasing of the tree on the two-dimensional display. Consequently, a specific description of paraphrasing P is required for each level.
- M. Finally, the class M of admissible tree manipulations on each level is of particular interest. The paraphrasing description should contain elements that describe how to present this set of manipulation operations to the user in order to provide for guidance in the manipulation selection process.

Fig. 3 depicts the principal schema of each system level.

Tree manipulations on each level fall into two distinct classes according to the two conceptually different kinds of activities that can take place on each system level:

Editing

If a user is faced with the development of a program for a new or for a modified problem, s/he has to input new specifications, program fragments, or transformation rules to the system. All these data are conceptually regarded as tree-structured. New trees with possibly new meanings are obtained by editing already existing ones. There is no semantic relationship between an edited tree and its original version. Editing is done by evoking a syntax-oriented editor. The operations of a tree editor can be reduced to the three basic operations "cut", "copy" and "paste". The L-specific actions of the editor concern the parsing of format-free linear input only. The editor part of each system level is therefore uniquely determined by the syntax of the language L and can be automatically generated (as in MENTOR or in the Cornell Program Synthesizer). Thus, for the editor part of each level, the class M of tree manipulation operations is given by the cut/copy/paste scheme. For this scheme, menu techniques for selecting editing operations are conceivable, apart from allowing format-free input of linearized tree notations.

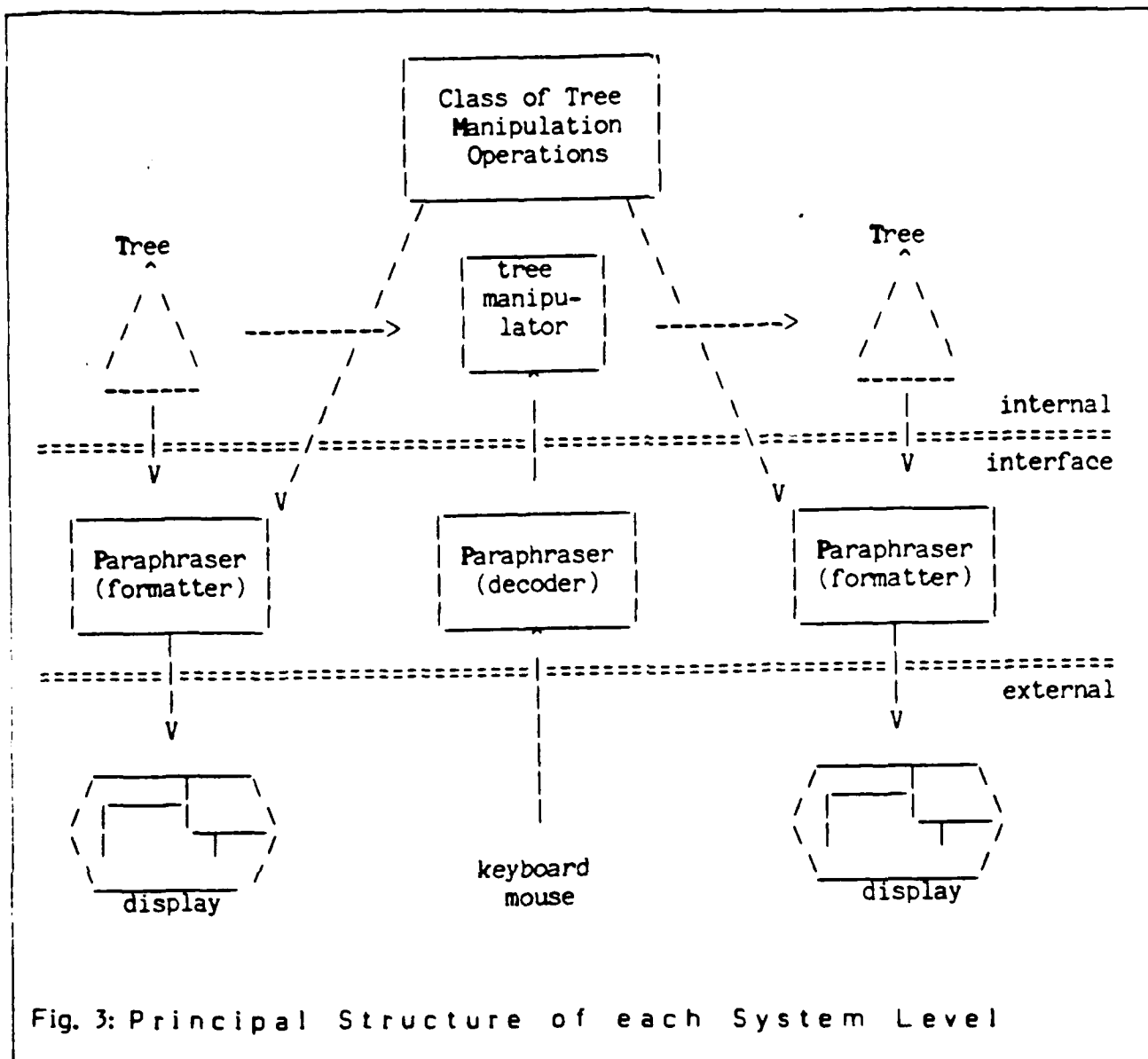


Fig. 3: Principal Structure of each System Level

Transforming

Conceptually different from editing is the activity of transforming trees into semantically equivalent ones by evoking a transformer. Preserving semantic meaning, tree transformations have to be sharply distinguished from editing, although from the user interaction point of view both activities are manipulations of trees and can be evoked and represented alike. (For example in the SMALLTALK programming environment, the left mouse button is used for selecting editing operations, whereas the middle one evokes semantically meaningful operations.)

Interpreting

It is interesting to note that the semantic interpretation of a tree can always be viewed as a tree transformation as well. The corresponding mathematical concept is that of a canonical term algebra. It is known that for any algebra an isomorphic canonical term algebra exists, where the carriers consist of trees of the given signature and where operations map an argument tree to a result tree. Thus, interpretation of a tree in a semantic term algebra means trans-

forming the tree into a minimal (congruent) representation. This way, program development as well as interpretation of "programs" on the various system levels is reduced to the one concept of tree transformation.

Interpreters of an Ada/Anna program are the specification prototyper, annotation checker, Ada compiler back end, and Ada subset to Pascal translator.

Hierarchy of System Levels

A hierarchy of system levels is anticipated; a subset follows below. We shall briefly sketch some of the characteristics of these levels. Their detailed structure and interdependencies are, however, subject of further research. We have already noted that on each such level editing will have to be distinguished from transformations. Since editing is standardized, we will in the following only mention transformation issues, denoting by M the class of characteristic transformations on a system level.

Control Level

T= CONTROLA abstract syntax trees

A= atomic commands such as transformation development actions, Ada/Anna program development actions.

M= interpretation of CONTROLA command trees

P= Menue selection of commands

Transformation Development Level

T= TRAFOLA abstract syntax trees

A= Ada/Anna program schemes as part of transformation rules, applicability conditions

M= generation of transformers from transformation rules/scripts

P= pretty printing of TRAFOLA rules, including Ada/Anna templates

Ada/Anna Transformation Level

T= Ada/Anna attributed abstract syntax trees

A= context attributes for (re-) evaluation by transformer

M= class of transformation rules as defined on the transformation development level

P= Pretty-printing of Ada/Anna programs and highlighting templates for transformation rule application

This level is itself structured into a hierarchy of sublevels that correspond to different stages in the program development process (e.g. requirement specification, design specification, applicative program, imperative program). Each sublevel has its own characteristic class of transformation rules. It may well turn out that these rules require different collections of context attributes for formulating applicability conditions. This would then require to define different but closely related paraphrasing descriptions.

Applicability Verification Level

T= applicability conditions

A= Ada/Anna semantic primitives

M= inference rules of predicate logic

6.6 System Development Components

Generator Components

Having achieved the reduction of the system's complexity to the few principles mentioned above, it is now possible to achieve corresponding reduction of the complexity of implementation. For that purpose, the development of a few basic generator components is conceived. The development of parameterized or generative system components is an indispensable concept of the PROSPECTRA project, both from a methodological and a technical point of view:

Reducing a systems complexity to a few principles, developing models as formal abstractions of these principles, and implementing highly parameterized software modules as their concrete representations is a generally accepted principle in software engineering. This is especially important in the PROSPECTRA project, since here a system is to be developed whose complexity is orders of magnitude beyond what can realistically be managed by naive ad-hoc implementation techniques.

The PROSPECTRA system is a multi-language (Ada, Anna, TRAFOLA, CONTROLA) program development environment. Editors, paraphrasers and transformers should, nevertheless, provide uniform operating principles. This requires that these modules be based on a uniform mechanism, which then, however, has to be parameterized by language descriptions. On the other hand, such parameterization increases the flexibility in systems design considerably. Changes in TRAFOLA or CONTROLA, for example, need then not lead to a redesign of all system modules. A parameterized module is called a **generator**, if upon instantiation with an actual parameter a nontrivial analysis of this parameter is performed to increase the module's performance. In this sense we will distinguish below between generators and parameterized system components.

For the PROSPECTRA project it is expected that the development of generators and their application to produce the corresponding system components will be much less expensive than developing and programming these system components one for one by hand. For the basic system (to be developed in the first two project years), it is planned to develop new components as prototype versions with less concern for generative aspects to gather experience for the development of the final system.

Apart from methodological and economic considerations, the program development environment as envisaged here is a dynamically changing one. It is expected that during the lifetime of the PROSPECTRA system new expertise in program development will be gained so that the system must be conceived from the beginning to allow its incorporation. Technically speaking, the user may input, at the transformer development level, transformation scripts representing new development strategies. Subsequently the user at the Ada/Anna development level can apply these rules. Different transformation rules are usually based on different kinds of semantic information. Consequently, both the transformer as well as the incremental evaluator of semantic information must be constructed to allow for adaptation to changes of semantic rules and transformation rules in the system. Generator components are the only way to solve this problem, if acceptable systems performance is to be maintained.

Finally it should be noted that some of the generators to be described below already exist up to adaptation and integration into the system (e.g. Parser Generator) or are presently under construction (initial version of Attribute Evaluation Generator and Transformer Generator), thereby constituting important methodological as well as technical contributions prior to project begin.

Parser Generator

The Parser Generator generates parsers for context-free languages. A (conventional batch mode) LALR(1) parser generator is available from Universität des Saarlandes. Parsers are contained as components in editors for analyzing format-free input. As part of an interactive editor, possibilities for incremental parsing are required. The existing parser generator must be adapted for these purposes.

Tree Manager

A tree manager provides the primitives for constructing, traversing and manipulating abstract syntax trees. The tree manager is parameterized by the description of the syntax of the language, together with its atomic lexical elements. The tree manager will be used as parts of editors, transformers, and attribute evaluators. IDL shall be evaluated for adaptation; coherence with DIANA must be ensured.

Attribute Evaluation Generator

Transformation strategies are expressed by transformation scripts. These include attribute grammar specifications for the computation of semantic information needed during transformation. Different transformation strategies require different kinds of semantic information. For example, attributes for transformations on the specification / applicative level will have to be defined. This corresponds to context information in a conventional compiler front-end. Similarly, attributes for data flow analysis are needed on the imperative level. At the transformer development level, the user must therefore be able to input new attribute evaluation rules as part of new transformation strategies. Corresponding attribute evaluators are then generated. An interface to the theorem prover has to be designed.

The implementation of the attribute grammar concept has to be adapted to the general system framework. One main problem area concerns the invalidation of attribution after application of transformations. A transformation may render the attribution inconsistent w.r.t structure of dependencies and values of attributes. This causes a need for updating attribute values in between consecutive transformations. For efficiency reasons this re-evaluation of attributes should be incremental. Considerable research will be required here.

Attribute storage management is crucial for the efficiency of basic level transformation. Particular care has to be taken for Ada/Anna symbol tables and attributes of type "set of laws".

Parameterized Structure-Oriented Editor

The complete syntactic description of a language is the parameter of a structure-oriented editor. Editing is an activity that defines new data (specifications, programs, transformation rules). There is no semantic relationship between the trees before and after editing. In particular, correctness of static semantics cannot be guaranteed. Therefore, editors have to call upon incremental attribute evaluation for reevaluating semantic attributes. Editor commands are evoked by appropriately interpreting user activation of input devices (keyboard, mouse). Since the user shall be allowed to act in terms of the intuitive external 2D representation of a tree, and since this representation is specified in the paraphrasing description, this decoding is the task of the paraphraser (cf. below).

Manipulation of atomic (lexical) elements of a tree means entering a lower-level editor. Thereby a user may create a chain of not-yet-completed calls to different editors. On the screen, the elements of this chain could be represented by different windows. Selecting a particular window (which is the third major kind of user activity and which is directly handled by the Virtual I/O-Driver) would then mean to resume editing on the associated system level. Whether this view is in fact appropriate for dealing with this aspect of the system's hierarchy is one of the research topics in the project.

In summary of the above, the editor as required in the project differs from existing or projected ones in that it must handle attributed trees rather than parse trees only, that it interfaces cleanly to incremental attribute evaluation, and that it supports a hierarchy of languages to separate the various system levels. Attributes are, depending on the current focus of user interest, either viewed as atomic semantic values or have themselves a tree structure that can be subject to editing operations. Furthermore, depending on the class of transformation rules, specific attributes of the tree are relevant only, while others may be invisible at this stage. This notion of attributed trees as incorporated in the tree manager is central to the planned system. Editors for such structures do not exist and hence have to be developed as part of the project.

Paraphraser Generator

At any time, the user of the system sees external ("paraphrased") representations of the internal attributed trees. As described above, this tree contains too much information that is relevant at other levels of user interaction only, at any stage. In order not to irritate the user with irrelevant information, a level-dependent external tree representation is strictly required. Since particularly important interaction levels correspond to classes of transformation rules, and since the system has to allow for adaptation to new transformation strategies, the tree representation process must be adaptable, too. A paraphraser generator is, therefore, an indispensable component of the final system to assist the dynamic evolution of the envisaged program development system.

A paraphraser generator accepts descriptions of how to format the attributed trees as well as the transformation rules of a particular system level. Since paraphrasing will require to traverse the trees in order to map subtrees to regions of the display, it seems possible to describe paraphrasing in terms of attribute grammars which employ the data types of the high-level Virtual I/O-Driver in attribute rules. This would not only allow to apply the attribute evaluator generator for constructing a further system development component, but also solve the problem of incremental reformatting of trees after transformation.

As the paraphrasing description contains all information of how regions inside a window correspond to sets of nodes of the internal trees, one would want to be able to also generate input device decoders from paraphrasing descriptions. These decoders poll input device activities and interpret these as selection of particular tree nodes. Thus, at any time, **tree configurations** consisting of trees and selected nodes represent the current focus of user concern. Paraphrasing descriptions have to contain specifications for highlighting selected nodes in trees.

Transformer Generator

The Interface to the Transformation Description Manager

The proposed program development methodology, the subject language, Ada, the annotation language, Anna, and existing standardized representations, e.g.

DIANA, need basic transformation rules with the following properties:

- input and output templates of transformation rules may have a regular structure, i.e. containing list-of-nodes. Access to list components must be made possible explicitly, i.e. using indices, implicitly by specifying a property accessed components must have, or through iteration.
- iteration operators may have to be specified, thereby differentiating between depth- and breadth-iteration.
- the domains of template parameters may be restricted by syntactic typing, i.e. specification by a grammar.
- output templates may contain "free" parameters. Their value in the case of rule application must be supplied from either the user or a program fragment library.

A set of rules entered into the system creates a rule library. This rule library can be extended by adding new rules, combining existing rules, substituting into existing rules, and changing applicability conditions of existing rules. The deletion of rules must also be supported. Changes may be caused by the system designer or result from Anna annotations, in particular from laws of algebraic data types. It must be possible to enter rule descriptors into the rule library, containing information about the author of the rule, a version identification, relations to other rules etc. The rule library has to be designed as part of the library manager containing program development steps.

Management of Transformation Scripts

Transformation scripts realize higher program transformation steps. They consist of sets of basic transformation rules together with a specification of successorship and history dependence. Each rule may specify a set of rules to be tried next, after it has been successfully applied, and a node expression describing where to try. Rule application in executing a script is history sensitive, e.g. may depend on the previously applied rules.

Language Constructs for Transformation Scripts

The language for the definition of transformation scripts must be powerful and flexible. In particular, it must contain language constructs for iteration at one node, exhaustive iteration, conditional iteration, for the specification of tree walks, for definition of areas excluded from rule application (any rule, as well rules from a given class).

Translation of Transformation Scripts

Transformation scripts are translated into automata tables which are mapped to basic tree operations. A library for automata and parts of automata has to be designed. Operations on this library and on objects in this library must include entering (i.e. generating) and deleting objects, merging two objects, optimizing objects under time or space constraints. Consistency of this library with the library on the description level has to be maintained.

6.7 Transformation Development Components

TRAFOLA Editor

TRAFOLA is the language for the definition of transformation scripts. Relations of successorship, history dependence, and control constructs describe strategies of rule application. Interactive input and modification of transformation rules is done via a structure-oriented editor. This editor is generated from the TRAFOLA syntax. The user is responsible for delivering correctness proofs for edited transformation rules.

Method Bank

The method bank will be the practical outcome of the methodological investigations about the process of specification and program development. The method bank relates program and specification development objectives to concrete transformation rules/scripts. This way, development steps are viewed as a goal-oriented process to achieve a certain objective. The concrete design and technical form of this method bank is subject to research in the project.

Initially, the method bank will just consist of a rule library (interfacing to the Library Manager), that is a set of correct transformation rules and scripts. The user modifies this rule library with the help of the method bank. In contrast to editing, these modifications will preserve correctness of the rule library. Such modifications are e.g. combinations of rules, substituting into existing rules, strengthening of applicability conditions, deletion of rules. The method bank also assists in deriving transformation rules from Anna specifications. A particular example would be the conversion of an equational axiom of a specification into a transformation rule. Transformation management means to transform TRAFOLA programs. Central parts of the method bank might, therefore, be described in TRAFOLA itself and generated by the transformer generator.

6.8 Ada/Anna Development Components

Ada/Anna Front End

A front end for a subset of Ada/Anna including the specification, applicative level and package interfaces shall be developed, that can be used as part of the Ada/Anna Editor. It will be derived from an existing attribute grammar for Ada, extended by those parts that are necessary for Anna. Expertise gained from the existing SYSTEAM Ada front end shall be used in the development of this new component. As far as possible, this front end shall be interactive allowing incremental re-evaluation of attributes. This will require significant re-search.

Ada/Anna Editor

The input of new specifications and/or programs at the applicative level of Ada/Anna is assisted by a specifically instantiated structure-oriented editor. As mentioned before, there is no guarantee that editing preserves correctness. Therefore, static semantic re-analysis of edited Ada/Anna programs is required using the Ada/Anna Front End. At this point it is not yet clear to what extent this analysis can be performed incrementally during editing. Some semantic attributes will have to be computed incrementally, in particular those which occur in applicability conditions of transformation rules.

Transformers

A hierarchy of transformation rule classes will be defined. The classes constitute levels of Ada/Anna transformation components corresponding to program development phases such as specification, applicative level, mapping to the imperative level. For these classes, transformers are generated. Different classes of transformation rules require different sets of semantic attributes in their applicability conditions. Also, the conditions themselves are of different logical complexity. (For example, low-level transformation rules specifying machine-independent optimizations such as constant folding would have applicability conditions that can be checked fully automatically. Transformation rules on the applicative level can depend on algebraic properties (e.g. commutativity) of operators. These require a substantial amount of theorem proving.)

Other problems that have to be investigated in more detail concern the interfaces between different levels, both in the conceptual and technical sense. For example the fact that different sets of semantic attributes are required for different rule classes implies that different notions of the semantics of Ada/Anna constructs exist. Consequently, different external views of Ada/Anna programs might be useful and should thus be represented by different paraphrasing. Transformations that transform programs of one level into programs on the next lower level are particularly interesting. Here, different attribute evaluators have to interface to each other.

In its present extent, the PROSPECTRA project concentrates on transformers at the specification and applicative level and to the imperative level. Transformers at the imperative level (analogous to transformations in optimizing compilers) and to the machine-oriented level can be added later when the system is in operation.

Verifiers

Theorem provers apply sequences of inference rules to derive theorems from the axioms of the theory. Inference rules are transformation rules on languages of proofs. TRAFOLA, the language for specifying transformation rules, scripts, and basic operations on sets of transformation rules will be designed to allow for formalizing knowledge embodied in transformation strategies. Hence the description of proof strategies is envisaged as one particular application of the general concept. This opens up ways to generate important components of theorem provers automatically. Since generated transformers are of the general interactive nature pointed out above, it is hoped that such theorem provers can be better guided by the problem-specific insights of the user.

The Verifier will contain theorem provers for applicability conditions (predicate logic) and for Anna Logic, with appropriate interfaces to transformers.

6.9 Control Components

Virtual I/O Driver

The Virtual I/O Driver is a PROSPECTRA high-level system interface that maps to the window manager and input device drivers provided by PCTE or a particular workstation manufacturer.

CONTROLA Editor and Controller

CONTROLA, a language for formulating control commands, forms the top level of the system hierarchy. Atomic elements of CONTROLA interface to the various system components. Command trees are input via the corresponding structure-oriented editor. Interpretation of a command leads to interaction with corresponding system components. It is conceivable that non-overlapping subtrees of command trees can be interpreted in parallel. For this purpose a concept of multi-tasking in our framework of system activities as tree manipulations would be desirable.

CONTROLA trees, together with their tree-structured atomic leaves that represent lower-level system actions, form a complete history for development activities. Complete re-interpretation of a tree means automatic replay of development processes.

Library Manager

A library manager has to provide for storing and accessing trees and their tree-structured atomic elements of all system levels. The system hierarchy will probably be mirrored in the library hierarchy. Additional structure will come from version management and from the application problem structure.

The library manager will interface to a lower level of an object oriented database provided by PCTE, CAIS or such like. This needs to provide DB-objects and relationships with attributes (preferably in a typed manner) to implement relationships of the system hierarchy, versions, and, as far as possible, information that enables the goal-oriented selection process of the method bank.

PROSPECTRA Project

7 Project Consortium

7.1 Project Structure

The partners of the project form the PROSPECTRA Project Consortium. Each partner is represented in the Consortium by its respective Team Director. The team director of the prime partner, Universität Bremen, is the Project Director.

In the following list, the partners are given in alphabetical order, the Prime Partner first.

Partners	Team Directors
Universität Bremen, Bremen, FRG	Prof. Bernd Krieg-Brückner
Universität Dortmund, Dortmund, FRG	Prof. Harald Ganzinger
Universität Passau, Passau, FRG	Prof. Manfred Broy
Universität des Saarlandes, Saarbrücken, FRG	Prof. Reinhard Wilhelm
University of Strathclyde, Glasgow, UK	Prof. Andrew D. McGettrick
SYSECA Logiciel, Saint Cloud, F	Ian G. Campbell
SYSTEM KG Dr. Winterstein, Karlsruhe, FRG	Dr. Georg Winterstein

7.2 Contribution of each Partner

7.2.1 Universität Bremen

Team

Prof. Dr. B. Krieg-Brückner, team and project director
Dr. B. Hoffmann
(S. Kahrs, B. Gersdorf, 1986)
NN's

Background Contributions and Qualifications

Bernd Krieg-Brückner has over 12 years of experience in language design, formal definition and implementation. His activity in IFIP WG 2.4 (Systems Implementation Languages) resulted in his participation in the Ada language design team as a key member. He was a major contributor to the INRIA Formal Definition of Ada. He is a German representative in the ISO standardization of Ada, a member of the Language Maintenance Committee of ISO WG Ada, and has been an active member of Ada Europe in the WG's on Language Review and Formal Semantics. He is initiator and chairman of the Ada Europe WG on Formal Methods for Specification and Development (of Ada programs).

Since 1979 he has been working on the design of Anna, a specification language extension of Ada. This work was started at Stanford University with Prof. D. C. Luckham and went on as a joint effort, with Dr. F.W. v. Henke (SRI) and Prof. O. Owe (UCSD, Univ. of Oslo). Language maintenance and formal definition of Anna shall continue on an international basis in cooperation with the PROSPECTRA Project.

Ada/Anna can be considered as a Wide Spectrum Language based on the work B. Krieg-Brückner has done in the SFB 49 at TU München. He was a key member of the project CIP for ten years. The CIP project is internationally acknowledged for doing fundamental research for the Program Development by Transformation methodology. This work is one of the bases for the PROSPECTRA project.

An offspring of this work is B. Krieg-Brückner's work on source-to-source translation, started while at UC Berkeley and continued now in a student project at U Bremen. Front and back ends (to and from Ada) for several high-level languages are expected as a result of this project by 1986, and shall be complementary to the PROSPECTRA project.

P R O S P E C T R A P r o j e c t

C o n t r i b u t i o n o f W o r k b y S u b j e c t C a t e g o r y

M.1 M e t h o d o l o g i c a l F r a m e w o r k

M.1.1 D e v e l o p m e n t b y T r a n s f o r m a t i o n

M.1.2 G o a l O r i e n t e d M e t h o d s

M.3 T r a n s f o r m a t i o n i n A d a / A n n a

M.3.1 B a s i c T r a n s f o r m a t i o n R u l e s

M.3.2 D e r i v e d T r a n s f o r m a t i o n R u l e s

M.3.3 B a s i c T r a n s f o r m a t i o n M e t h o d s

S.3 A d a / A n n a D e v e l o p m e n t C o m p o n e n t s

S.3.3 T r a n s f o r m e r s

P P r o j e c t M a n a g e m e n t

7.2.2 Universität Dortmund

Team

Prof. Dr. H. Ganzinger, team director
NN's

Background Contributions and Qualifications

H. Ganzinger has over ten years experience in compiler generation, abstract data types, formal semantics, and text processing. He is chairman of the Gesellschaft für Informatik WG on Language Implementation, and an active member of Ada Europe WGs on Formal Semantics and Formal Methods.

Since 1974 he has been working on the design and implementation of compiler generating systems. He was a key member of the project MUG (direction: Prof. Eickel) at TU München for 9 years. The MUG project is internationally acknowledged for fundamental research on all phases of compiler generation including the development of description languages for modular compiler specifications. Practical outcome of this research have been three running compiler generators, the last of which includes features for modular compiler descriptions. Here, compilers are regarded as tree transformation phases, a very relevant point of view for the basic principles of the system activity in the PROSPECTRA project.

Since 1980 he has been working on algebraic specification of abstract data types (started while on leave at UC Berkeley). He has developed notions and proof techniques for implementation selections between parameterized equational specifications based on concepts of observability. Later on he extended concepts of modularity and implementation of abstract data types to algebraic structures (having relations in addition to operations). Practical outcome of this theoretical work is a new method for modular formal language semantics and compiler specifications, where modules correspond to fundamental language concepts and basic compilation techniques.

H. Ganzinger has designed and implemented a powerful text formatting system (FOAM) on a microcomputer. Its main feature is that text formatters are generated from high-level descriptions of the text and document structure, and from specifications of formatting styles. This experience and his current work on the implementation of the virtual SMALLTALK 80 machine on a 68000-based micro computer are highly relevant to the work on user interaction and virtual graphic I/O in the PROSPECTRA project.

PROSPECTRA Project

Contribution of Work by Subject Category

S.1 System Development Components

S.1.4 Paraphraser Generator

S.1.5 Editor Generator

S.3 Ada/Anna Development Components

S.3.2 Ada/Anna Editor

S.2 Transformation Development Components

S.2.1 TRAFOLA Editor

S.4 Control Components

S.4.2 CONTROLA Editor

S.4.3 Controller

PROSPECTRA Project

7.2.3 Universität Passau

Team

Prof. Dr. M. Broy, team director
NN's

Background Contributions and Qualifications

M. Broy has nine years of experience in formal semantics, formal specification, and program development by transformation. As a key member of the CIP project (TU München), he was instrumental in its extensive, internationally acknowledged research in algebraic specification and transformation, and the theoretical background of concurrency.

His work at U Passau centers around the formal foundation of programs and program development in the areas of transformation rules and methods, algebraic specification, formal derivation of algorithms, and semantics of concurrent communicating systems.

Contribution of Work by Subject Category

M.2 Formal Basis

M.2.1 Semantic Foundation of the Methodology

M.2.2 Algebraic Specification

M.2.3 Concurrency

PROSPECTRA Project

7.2.4 Universität des Saarlandes

Team

Prof. Dr. R. Wilhelm, team director
Dr. U. Möncke
B. Weisgerber
S. Pistorius
R. Heckmann

Background Contributions and Qualifications

R. Wilhelm has been working on the design and implementation of compiler generating systems for fifteen years. He was a leading member of the project MUG at TU München for six years. The MUG project is internationally acknowledged for fundamental research on all phases of compiler generation.

R. Wilhelm has done internationally acknowledged fundamental research in attributed tree transformations for more than ten years. The Tree Transformation Group at Saarbrücken, directed by R. Wilhelm, will bring its important expertise in tree transformation techniques into the PROSPECTRA project. The group has jointly worked on transformation of programs represented as attributed trees for about five years. A language, OPTRAN, has been designed for the description of such transformations, generators have been implemented for efficient tree pattern matching, attribute evaluation and re-evaluation.

Contribution of Work by Subject Category

S.1 System Development Components

S.1.1 Parser Generator

S.1.3 Attribute Evaluation Generator

S.1.6 Transformer Generator

7.2.5 University of Strathclyde, Glasgow

Team

Professor A.D. McGettrick, team director
NN's

Background Contributions and Qualifications

A.D. McGettrick has held three Science and Engineering Council (UK) research grants on aspects of formal methods related to program verification and specification.

As a member of Ada Europe he has been active mainly in the Formal Methods Working Group (secretary) but also in the Formal Semantics and Telecommunication Working Groups. He also joined the group headed by Professor Stephan Goldsack for the European Commission on specification associated with Ada.

His book on program verification using Ada was written around 1980/1. Many of the ideas on verification and specification have been further developed since then through UK research grants and through such mechanism as the Ada Europe Working Group on Formal Methods. This will form the basis for the proposed R&D activities.

Contribution of Work by Subject Category

S.3 Ada/Anna Development Components

S.3.4 Verifier

7.2.6 SYSECA Logiciel

T e a m

Ian G. Campbell, team director
Christian Fiegel
Dr. Michel Lai
NN's

B a c k g r o u n d C o n t r i b u t i o n s a n d Q u a l i f i c a t i o n s

Ian Campbell has over 19 years of experience in systems software, operating systems design, and development tools. He is at present the SYSECA project manager for the Emeraude project to produce an industrially available basis for advanced, integrated program support environments entirely compatible with the PCTE defined portable common tool interface.

Christian Fiegel has been responsible for the design of the distribution mechanisms over the LAN for the Emeraude project and the PCTE project in collaboration with ICL. Prior to that he developed an object management system for a language based, integrated environment.

Particular background contributions from SYSECA include the results of different software engineering environment projects such as:

- Emeraude: French national advanced software engineering environment base
- PCTE: Basis for a portable common tool environment defined in the software technology PCTE project of the ESPRIT programme
- Concerto: French telecommunications research laboratory CNET's integrated software development environment.

C o n t r i b u t i o n o f W o r k b y S u b j e c t C a t e g o r y

S.2 T r a n s f o r m a t i o n D e v e l o p m e n t C o m p o n e n t s

S.2.2 M e t h o d B a n k

S.4 C o n t r o l C o m p o n e n t s

S.4.4 L i b r a r y M a n a g e r

E E v a l u a t i o n , R e v i e w , E x e r c i s e s

7.2.7 SYSTEAM KG, Karlsruhe

Team

Dr. G. Winterstein, team director
Dr. E. Zimmermann
Dr. P. Dencker
NN's

Background Contributions and Qualifications

G. Winterstein's original research area was formal logics; this is a very suitable background for the PROSPECTRA project in addition to his practical work on Ada implementation. He is a member of Ada Europe WG's Formal Semantics, Formal Methods and Implementation (as convenor), also a member of the EC study group on Ada specification issues (Goldsack).

G. Winterstein was the leader of the Ada implementation team at U Karlsruhe from 1979 to 1982 when he founded his own company SYSTEAM. The Karlsruhe Ada implementation was, apart from the NYU operational definition, the first complete Ada implementation to become fully operational (for Ada 80). The Ada compiler is now maintained and upgraded by SYSTEAM and formal validation for ANSI Ada has been achieved in Nov. 84; back ends are developed by GMD Karlsruhe (Prof. Goos) and by SYSTEAM. The Karlsruhe implementation was originally derived from the INRIA Formal Definition of Ada. The INRIA Abstract Syntax definition was developed into the de facto standard DIANA by the Karlsruhe team in cooperation with Tartan Labs.

The present front end is derived from an Attribute Grammar for Ada which will be a basis for developing transformation rules by Universität Bremen in cooperation with SYSTEAM.

The experience in compiler generators (a parser generator, attribute grammar generator, and code generator generator were developed at U Karlsruhe) and practical generation for realistic languages (Pascal, PEARL, LIS, Ada) and Ada and DIANA has migrated to SYSTEAM to a large extent. E. Zimmermann and P. Dencker are two of those who came from U Karlsruhe; their experience in compiler generation and attribute grammars for Ada is particularly welcome for the project.

SYSTEAM will provide its Ada compiler for system development. The Ada subset to Pascal translator developed by SYSTEAM (AdaP system) will be an important bootstrap tool.

Contribution of Work by Subject Category

S.1 System Development Components

S.1.2 Tree Manager

S.3 Ada/Anna Development Components

S.3.1 Ada/Anna Front End

S.4 Control Components

S.4.1 Virtual I/O Driver

The PROSPECTRA Project

(with an emphasis on Verification)

Andrew D. McGettrick

University of Strathclyde

May, 1986

The PROSPECTRA project forms a part of the ESPRIT programme of the Commission of the European Communities. It is concerned with the development of a methodology and support system for the production of correct Ada programs. The project is around 70 man years in size, lasting 5 years. The partners and main contributors are:

B. Krieg-Bruckner, Universitat Bremen
H. Ganzinger, Universitat Dortmund
M. Broy, Universitat Passau
R. Wilhelm, U. Moncke, B. Weisgerber, Universitat des Saarlandes
A. McGettrick, University of Strathclyde
I. Campbell, SYSECA Logiciel
G. Winterstein and L. Treff, SYSTEAM KG

The project has been under way for just over 1 year. Many of the fundamental aspects of the project are being researched and it will not always be possible to provide complete answers about the ultimate direction of progress. However, we shall outline the main features of the project and then pay special attention to the verification issues associated with them. In advance, it should be said that the 'verification' involved does not have the traditional significance; but certainly algebraic manipulation and theorem proving will be part of this.

1. Background

The motivation for the project stems from a deep concern about the methods and techniques used to produce the majority of today's programs, many of which are to be used for sensitive applications. Programming is often associated with an undisciplined approach, with testing methods that are inadequate and with a sense of 're-inventing the wheel'. In addition truly large systems produce complexity which is all-but-unmanageable.

A strict methodology for program development is desirable, together with software support of a sophisticated kind. The proposed methodology combines and integrates program construction and verification so that the resulting programs are known to be correct in the sense that they conform to the initial specification. It does not cover the earlier phases of analysing requirements nor the formulation of formal specification; these are seen to be outwith the scope of the project. However once a formal specification has been obtained the methodology is rigorous from then onwards.

The basis for the PROSPECTRA project is the idea that, starting from an initial formal specification, it is possible to proceed from there to a final program by applying a sequence of correctness-preserving transformations. The idea is not new. The origins of the project can be seen in the work of the CIP project in Munich [Bauer 85] where the transformational approach has been under investigation for several years. However there are substantial differences between the PROSPECTRA ideas and the CIP ideas. These will emerge as we proceed. However, at this stage we note that PROSPECTRA is directly concerned with the Ada language, and indeed the role of transformations and transformation scripts is seen as being more fundamental.

The Ada/Anna combination provides a convenient framework within which specification can be described and programs can be written. Thus it is a wide-spectrum language. The correctness of transformations needs to be established and this can only be realistically done if formal definitions abound. Part of the PROSPECTRA activity is to provide a thorough basis on which to develop all the transformations, and so on.

2. The PROSPECTRA Methodology

Details of the PROSPECTRA methodology and the accompanying support system can be found in [Krieg-Bruckner 86b] and there is no need to repeat that detail here. However, we shall provide an outline, to establish some terms that can be used in the ensuing discussion.

The life cycle of a development activity is seen as consisting of three main phases:

pre-development phase

- when there is an informal problem description and an informal analysis of requirements

development phases

- consisting of two main parts

SPECIFICATION

when a formal specification of requirements takes place, so providing the contract with the user

IMPLEMENTATION

when the design specification of components is provided using a top-down decomposition, and this is followed by a bottom-up process of composing pieces of software to provide a working product

post-development phase

- during which evolution takes place in response to changes in requirements.

Within this framework the requirements specification tends to be characterised by being non-constructive, by being a loose specification and by exhibiting only the necessary requirements. Conversely design specifications require to be constructive, and to be readily amenable to implementation. The major challenge is to show how to move from one form of specification to another.

Within PROSPECTRA, correctness-preserving transformations are used to accomplish this, and then the construction and verification activities are combined into one single activity. The only verification that needs to be performed relates to the applicability rules associated with the various transformations.

To have to develop programs by having to describe every single transformation in order would be tedious in the extreme. An important ingredient of the PROSPECTRA methodology is to look at the development of a kind of calculus of transformations, whereby groups of transformations can be described and combined into coherent units.

3. The Support System

Various components will be provided to facilitate the production of aids to assist development. There will be components

to allow editing of programs, e.g. by inputting new specifications, program fragments or new transformation rules; a parameterised structure-oriented editor is used for this purpose.

to perform paraphrasing, i.e. omitting information that is irrelevant for the purpose at hand (paraphraser generator)

to generate transformers for rules and scripts.

These might be described as the system development components.

Other components are provided to assist with the Ada/Anna front-end, the transformational development and control. A feature of these is that they will permit development histories to be recorded and replayed if so required.

4. A Simple Example

A simple example that illustrates the PROSPECTRA methodology is taken from [Krieg-Bruckner 86a]. This example makes use of the / and mod operations. These specifications can be phrased as

```
function "/" (A,B:NATURAL) return NATURAL;
-- | where B > 0,
-- |     return Q:NATURAL=>
-- |         exists R: NATURAL => R < B and
-- |             A = B * Q + R;

function "mod" (A,B : NATURAL) return NATURAL;
-- | where B > 0,
-- |     return R: NATURAL => R < B and
-- |         exists Q:NATURAL =>
-- |             A = B * Q + R;
```

However, here the two functions are specified by means of characteristic predicates. Yet the two are intimately related and a style of specification that recognises this results in


```

function "/" (A,B: NATURAL) return NATURAL;
    --|  where B > 0;

function "mod" (A,B : NATURAL) return NATURAL;
    --|  where B > 0;

--|  axiom for all A,B : NATURAL =>
--|      A mod B < B,
--|      A = B * (A / B) + A mod B;

```

The challenge is now to move from this specification through a sequence of correctness preserving transformations to an Ada program.

For the next step it becomes necessary to identify some axioms that can form the basis of transformation. Consider the conditional equation

$$A < B \rightarrow A / B = 0,$$

$$A \geq B \rightarrow A / B = (A - B) / B + 1$$

and

$$A < B \rightarrow A \bmod B = A$$

$$A \geq B \rightarrow A \bmod B = (A - B) \bmod B$$

It is then necessary to show that, from these axioms, the original algebraic specification continues to hold. Effectively a proof by induction, on the magnitude of A, can be given. For consider

$$A = B * (A / B) + A \bmod B$$

Then from these new axioms we can deduce successively that

$$A = B * ((A - B) / B + 1) + A \bmod B$$

$$A = B * ((A - B) / B) + B + A \bmod B$$

$$A - B = B * ((A - B) / B) + (A - B) \bmod B$$

$$A_1 = B * (A_1 / B) + A_1 \bmod B$$

and A_1 is of smaller magnitude than B . This argument forms the basis of a proof by induction.

From these axioms we can produce the following implementation in recursive form

```
function "mod" (A , B : NATURAL) return NATURAL is
begin
    if A < B then
        return A;
    else
        return (A - B) mod B;
    end if;
end "mod";
```

A similar development will produce a recursive version of $/$.

At this point a standard transformation can be invoked, for the removal of tail recursion:

```
function F(X : S) return R is
begin
    if B(X) then
        return T(X);
    else
        return F(H(X));
    end if;
end;
```

This program schema can be replaced by the iterative equivalent

```
function F(X : S) return R is
    VX : S := X;
begin
    while not B(VX) loop
        VX := H(VX);
    end loop;
    return T (VX);
end F;
```

so producing an iterative version of "mod" (and similarly of /).

5. Uses of a Verifier within PROSPECTRA

Within the PROSPECTRA project three major uses can be identified for the verifier:

1. in animating or asking questions of an initial formal specification; for this specification will form the basis of the entire development and it is essential that it should exhibit the kinds of properties that are expected.
2. in going from an initial formal specification to an initial abstract implementation there is a need to check the correctness of this step
3. in checking the applicability conditions associated with transformation rules
4. to check the correctness of the correctness - preserving transformation rules.

There is no verifier in the traditional sense - there is no verification condition generator, for example.

Each of these uses merits considerable investigation. Let us look at them in turn:

- (i) in the first a rewrite system, for example, can be used to animate the specification; alternatively the Knuth-Bendix algorithm might be applied, as in the REVE [Kirchner 82] system; or a theorem prover might be employed
- (ii) in the second standard catalogues might be used or a more generative approach might be employed; alternatively other techniques can be used
- (iii) the work to be employed here relates to the degree of sophistication of the permitted transformations, and is governed by this
- (iv) there is a relationship between correctness-preserving transformation and the formal definition of the underlying programming language.

6. Conclusion

There is still a great deal of basic research to be carried out within the PROSPECTRA framework. Indeed this talk may have aired more questions than answers.

It might be observed that, if our methodology and support system turn out to be powerful enough, the task of producing an automatic verifier of the more traditional kind will be greatly reduced. For a verification condition generator can be viewed as a process whereby transformations are carried out on predicates, the transformations being determined by the program. Additionally algebraic simplifiers are a means whereby various axioms are applied, and axioms themselves are essentially transformations.

It remains to be seen whether the methodology we advocate does indeed appear to be productive. At the moment very few people would develop their programs in the manner suggested and clearly a large educational problem exists. However, at the moment the onus is on us to provide an environment in which users can comfortably develop their correct Ada programs.

7. References

[Bauer, F.L., et al 85] The Munich Project CIP, Vol. 1 : The Wide
Spectrum Language CIP-L. Lecture Notes in Computer Science,
Springer 1985.

[Kirchner, C., and Kirchner, H. 82] New Applications of the REVE
System, Centre de Recherche en Informatique de Nancy, France,
1982

[Krieg-Bruckner, B. 86a] Integration of Program Construction and
Verification : the PROSPECTRA Methodology and System,
Universitat Bremen, 1986.

[Krieg-Bruckner, B., et al 86b] Program Development by Specification
and Transformation in Ada/Anna, in Ada : Managing the
Transition, pp 249-258, Ada Companion Series, Cambridge
University Press.

PROGRAM DEVELOPMENT BY

SPECIFICATION AND

TRANSFORMATION

- B. Kreig-Brueckner, Bremen**
- H. Ganzinger, Dortmund**
- M. Broy, Passau**
- R. Wilhelm, Saarlandes**
- A. McGettrick, Strathelyde**
- I. Campbell, SYSECA LOGICIEL**
- G. Winterstein and L. Treff, SYSTEAM**

MOTIVATION

**BASIC CONCERN ABOUT METHODS
AND TECHNIQUES USED IN PROGRAMMING
TODAY**

**UNDICIPLINED APPROACH
INADEQUATE RELIABILITY
RE-INVENTING OF WHEEL
COMPLEXITY ALL BUT UNHARNESSED**

OBJECTIVES

**TO DEVELOP A STRICT METHODOLOGY
AND APPROPRIATE SOFTWARE SUPPORT
LEADING TO CORRECT (ADA) PROGRAMS**

INITIAL FORMAL SPECIFICATION

**APPLICATION OF CORRECTNESS
PRESERVING TRANSFORMATIONS
LEADING TO PROGRAMS**

**PROGRAM CONSTRUCTION AND PROGRAM
VERIFICATION PULLED TOGETHER**

SOFTWARE LIFE CYCLE

PREDEVELOPMENT PHASE

**INFORMAL PROBLEM DESCRIPTION
AND ANALYSIS OF REQUIREMENTS**

DEVELOPMENT PHASE

SPECIFICATION

**FORMAL SPECIFICATION OF
REQUIREMENTS TAKES PLACE
LEADING TO "CONTRACT"**

IMPLEMENTATION

**THE DESIGN SPECIFICATION OF
COMPONENTS IS PROVIDED USING
A TOP-DOWN DECOMPOSITION, AND
BOTTOM-UP PROCESS OF COMPOSING
PIECES OF SOFTWARE**

POST-DEVELOPMENT PHASE

**EVOLUTION, LEADING TO CHANGES
IN REQUIREMENTS**

REQUIREMENTS SPECIFICATION

**NON-CONSTRUCTIVE
LOOSE SPECIFICATION
EXHIBITS ONLY NECESSARY
REQUIREMENTS**

DESIGN SPECIFICATION

**CONSTRUCTIVE
READILY AMENABLE TO
IMPLEMENTATION**

**HOW ARE REQUIREMENTS SPECIFI-
CATIONS CHANGED TO DESIGN
SPECIFICATIONS?**

MAIN FEATURES

**SPECIFICATION -- FORMAL SPECIFICATIONS
ARE FOUNDATIONS ON WHICH TO BUILD**

**PROGRAMMING LANGUAGE SPECTRUM
-- ADA/ANNA**

**SOFTWARE COMPONENTS -- METHODOLOGY
USE OF ADA SUPPORTS THE CONCEPT OF
PACKAGE DEVELOPMENT**

TOOL ENVIRONMENT --

**CENTRAL CONCEPT IS APPLICATION OF
TRANSFORMATIONS TO TREES**

**TRAFOLA--LANGUAGE OF TRANSFORMATION
DESCRIPTIONS - SCRIPTS**

CONTROLA--COMMAND LANGUAGE

```

function "/" (A,B: NATURAL) return NATURAL
  -- |  where B > 0,
  -- |      return Q: NATURAL  =>
  -- |          exists R: NATURAL  => R < B  AND
  -- |              A = B*Q + R;

```

```

function "mod" (A,B: NATURAL) return NATURAL;
  -- |  where B > 0,
  -- |      return R: NATURAL=>  R < B AND
  -- |          exists Q: NATURAL  =>
  -- |              A = B * Q + R;

```

UNIQUENESS QUESTION?

function "/" (A,B: NATURAL) return NATURAL;

-- | where B > 0;

function "mod" (A,B: NATURAL) return NATURAL;

-- | where B > 0;

-- | axiom for all A,B: NATURAL =>

-- | A mod B < B,

-- | A = B * (A/B) + A mod B;

FOR THE NEXT STEP, ONE WAY FORWARD
IS TO IDENTIFY AXIOMS THAT FORM THE
BASIS OF TRANSFORMATIONS

$$A < B \longrightarrow A/B \approx 0$$

$$A \geq B \longrightarrow A/B \approx (A - B)/B + 1$$

$$A < B \longrightarrow A \bmod B = A$$

$$A \geq B \longrightarrow A \bmod B = (A - B) \bmod B$$

$$A = B * (A/B) + A \mod B$$

$$A = B * ((A - B)/B + 1) + A \mod B$$

$$A = B * ((A - B)/B) + B + A \mod B$$

$$A - B = B * ((A - B)/B) + A - B \mod B$$

$$A1 = B * (A1/B) + A1 \mod B$$

THIS FORMS THE BASIS FOR A PROOF BY
INDUCTION ON THE MAGNITUDE OF A.

```
function "mod" (A,B: NATURAL) return
NATURAL is
begin
    if A < B then
        return A;
    else
        return (A - B) mod B;
    end if;
end"mod";
```

A SIMILAR DEVELOPMENT WILL PRODUCE A
RECURSIVE VERSION OF /. HOWEVER NOT
QUITE FAIL RECURSION.

Given any $A, B \in \mathbb{N}; B > 0 \quad \exists Q, R \in \mathbb{N}$ such that

$$A = B * Q + R \quad \wedge \quad 0 \leq R < B$$

Proof by induction on N where, say, $N = \max(0, A-B) + 1$

For the base case $N = 1$ take - - -

Induction step: assume result true for all $X, Y \in \mathbb{N}, Y > 0$

with $X - Y < N$. Take any A, B such that

$$A - B < N + 1$$

Let $X = A - B$ and $Y = B$. Then $X - Y < N$ and by induction hypothesis

$$X = Y * Q' + R' \quad \text{and} \quad 0 \leq R' < Y$$

$$\text{i.e.} \quad A = B * (Q' + 1) + R' \quad \text{and} \quad 0 \leq R' < Y$$

$$\text{choose} \quad Q = Q' + 1 \quad \text{and} \quad 0 \leq R = R'$$

```

function F (X:S)
  return R is
begin
  if B (X) then
    return T(X);
  else
    return F (H(X));
  end if;
end;

```

```

function F (X:S)
  return R is
    VX : S := X;
begin
  while not B(VX) loop
    VX := H(VX);
  end loop;
  return T (VX);
end;

```

```

function "mod" (A,B: NATURAL) return
  NATURAL is
    VA: NATURAL := A;
begin
  while VA >= B loop
    VA := VA - B;
  end loop;
  return VA;
end;

```

**NOTE THE INTRODUCTION OF ASSIGNMENT AND
THE IMPLICATIONS FOR ADA.**

Procedure **EUCLID** (A, B: integer; Q,R: out integer);

QDASH, RDASH: integer;

begin

if A < B then

Q: = 0; R: = A;

else

EUCLID (A - B, B, QDASH, RDASH);

Q: = QDASH + 1; RDASH: = R;

end if;

end;

SUPPORT SYSTEM

**THIS IS RELATED TO THE DIFFERENT
LANGUAGES THAT EXIST**

- CONTROLA** For uniform system commands
- TRAFOLA** For describing transformation rules,
Scripts and Methods
- GENLA** For description of systems, tree
description, attribute domains, edit
rules, etc.
- PA^M DA** For program development
- VERIFILA** For applicability conditions, theorems
about properties of transformations,
corrections, etc.

HISTORIES CAN BE REMEMBERED AND REPLAYED

ROLE OF VERIFIER WITHIN PROSPECTRA

- 1. to animate or ask questions of an initial formal specification**
 - term rewriting**
 - Knuth-Bendix**
 - deducing properties, eg. uniqueness, proving theorems**
- 2. in going from an initial formal specification to an initial abstract implementation there is a need to ensure corectness**
- 3. in checking applicablity conditions**
 - syntax only directed**
 - type checked**
 - domain knowledge**

4. to check correctness of the correctness-processing transformations themselves

based on the formal definition
of a subset of Ada/Anna.

5. to 'discover' programs

theorem prover used to discover
constructive proofs which are then
easily changed into programs

(mathematical induction →
recursive subprograms)

CONCLUSIONS

1. Much remains to be done
2. Need to prove the methodology realistic (subsidiary project will attempt to apply this in an industrial setting) . . . and to supply a comfortable environment
3. Automatic verifiers of the traditional kind not needed if the strict methodology adhered to. But clearly desirable:

verification condition generator can be implemented by a sequence of transformations to predicates

algebraic simplifiers can be implemented by a sequence of applications of arithmetic axioms (transformations)

theorem proven

ON THE USE OF SEMANTIC SPECIFICATION
FOR THE VERIFICATION AND VALIDATION OF REAL TIME SOFTWARE

Author and Affiliation : Patrick de BONDELI,
CR2A and AEROSPATIALE/Space Division,
FRANCE

Mailing Address : 14, boulevard Jean Mermoz
(air mail exclusively) 92200 NEUILLY-SUR-SEINE, FRANCE

Phone : 331 47 22 06 14 (Home)
331 34 75 07 83 (Office)
331 47 80 23 31 (Office, before 3-15-1986)
331 47 68 97 97 (Office, after 3-15-1986)

-:-:-:-

ABSTRACT :

Our purpose is to illustrate, through a simple, but realistic and reusable, example of Ada package, the use of semantic specifications to implement, verify, and validate real time software.

The example we consider is an abstract data type (adt) intended to support a specific kind of inter-task synchronization-communication mechanism, "BROADCASTING", which is not predefined in the Ada language.

The specifications of this adt consist in three elements :

- The Ada package specification.
- Annotations in the package specification which provide pre and post-conditions for each operation.
- A Predicate-Transition PETRI Net (Pr-T Net) which specifies the synchronization-communication rules enforced by the adt.

This "specification-oriented Pr-T Net" is then developed into an Ada implementation-oriented Pr-T Net" using Pr-T net semantic models of Ada tasking which come from a previous work we presented in 1983.

Verification and validation then consist in :

- Proving that the "Ada implementation-oriented Pr-T Net" is correct with respect to the "Specification-oriented Pr-T Net".
- Deriving from the specifications and executing a test program in order to double-check the proof.
(Fortunately, this later operation revealed no error when it was actually performed).

KEYWORDS :

Ada real time program verification - Formal semantics -
Predicate-Transition nets - Specifications - Abstract data types.

1 - INTRODUCTION

Our purpose is to illustrate, through a simple, but realistic and useful, example, the use of semantic specifications to implement, verify, and validate real time software.

Ada packages provide a good mean of structuring software by separating the module (package) specification from the implementation details. But, when you specify a package in Ada, you give only the syntax and static semantics (type of input and output data, ...) of the operations made available by the package and not their dynamic semantics (what these operations are supposed to do).

On the other hand, verification and validation of a package are only possible if you can verify and validate the implementation against what your package and the operations it provides are supposed to do and compare the theoretical behaviour defined by the semantic specification to the actual behaviour of the implementation.

When only sequential operations are involved, the behaviour of these operations is well defined by specifying for each operation the "preconditions", that is the conditions that must hold before the operation is started, and the "postconditions", that is the conditions that hold as a result of the operation.

The ANNA annotation system (ref. ANNA) does that and also provides useful annotations during the development of the implementation.

In real time software, many concurrent operations may take place at a given time and it is necessary to specify the synchronizations and interactions between them.

Different formalisms may be used to specify these synchronizations and interactions.

The most commonly used formalisms today and the most thoroughly investigated by researchers are probably the Temporal Logic (ref. MP82, BKP84) and the PETRI Nets (ref. PET77, NET80, BRA82).

We have chosen to use the Predicate-Transition nets (or Pr-T nets), a high level class of PETRI nets (ref. GL79, GL80), for several reasons :

- . They can be represented graphically ;
- . We have, in a previous work (ref. BOND83), given a semantic definition of the Ada tasking constructs in Pr-T nets. Other papers giving elements on Ada tasking semantics using PETRI nets have also been published elsewhere (ref. MZGT85, SC85). We are therefore able to stay with the same kind of semantic models (the Pr-T nets) from specification to implementation (in Ada). Since verification and validation essentially consist in comparing the implemented behaviour to the specified behaviour, it is a great help to stay with the same kind of semantic models from specification to implementation.

- . We have good reasons to hope, from on-going research work (ref. GL83, PL83, KL84, VM84), that we will be able in a few years to undertake formal mathematical proofs in many practical cases on this formalism. However, that is not yet the state of the art today and we will keep in this presentation a deductive (and less satisfactory) kind of proof to support verification and validation.

The remaining of our presentation will be as follows :

- .2 / Rationale and requirements for our example : Package BROADCASTING.
- .3 / A specification for Package BROADCASTING
- .4 / An Ada implementation of Package BROADCASTING.
- .5 / Verification and validation of Package BROADCASTING :
 - 5.1. Proving the correctness of the implementation against the specification
 - 5.2. Deriving a test set from the semantic specification
- .6 / Conclusions
- .Appendix : Predicate-Transition Nets (Pr-T Nets)

2 - RATIONALE AND REQUIREMENTS FOR PACKAGE BROADCASTING

In Ada, the only inter-task synchronization mechanism is the "Rendez-vous" which provides synchronization points ("entries") between two tasks :

- . The "caller" may call an "entry" of the task it wants to synchronize with and waits until this entry call is "accepted".
- . The "accepter" arriving on an "accept" statement for one of its entries E has the following behaviour :
 - if no call was issued on E, it waits until a call is issued then it "accepts" the first (oldest) call, which may involve taking parameters from it, performing a sequence of statements and returning parameters.

Only when this "accept" is performed, the two tasks (caller and acceptor) can resume asynchronous processing.

In real time applications, particularly in GC applications, another kind of synchronization is quite often useful :
A task has to broadcast a message on a "message-carrier" to an unknown number of "receiving" tasks.

- . The broadcasting task does not want to wait, no matter if some "receiving" tasks are not ready to take the message.

- . The receiving tasks, when they invoke the "receive" primitive, wait until a message is present on the message-carrier and then take all this same message (which still keeps being present on the message-carrier after that).
- . An additional primitive, "Reset", is provided to allow the broadcasting task to "suppress" the message on the message-carrier (or "unload" the message-carrier) and force the subsequent receivers to wait for a new message.

A very common example of use of such a style of synchronization in GC would be a navigation subsystem periodically broadcasting its last fix to other subsystems.

We will take this BROADCASTING package, offering the synchronization type MESSAGE-CARRIER with the three operations BROADCAST, RESET, RECEIVE as our working example.

3 - SPECIFICATION OF PACKAGE BROADCASTING

This specification is given in two parts :

- . The next page displays the Ada specification including as comments the pre-conditions and post-conditions of each operation.
- . The following page displays the Pr-T net giving the synchronization specification.
Readers who are not familiar with Pr-T nets should carefully read the appendix and preferably also the basic references on PETRI nets (PET77, NET80, BRA82, GL79, GL80) before proceeding with reading the remaining of this paper.

Ada Specification of Package BROADCASTING

Generic

```
type MESSAGE is private; -- type of message to broadcast
Package BROADCASTING is
  type MESSAGE_CARRIER is limited private;
  -- The "MESSAGE_CARRIERS" are the vehicles for broadcasting messages.
  -- A MESSAGE_CARRIER may be empty or may carry one single message.
  -- It is initially empty.
  procedure BROADCAST (MESS : MESSAGE; CARRIER : in out
    MESSAGE_CARRIER);
  -- PRECONDITION : CARRIER is empty or carries a message.
  -- POSTCONDITION : CARRIER carries the message MESS.
  procedure RESET (CARRIER : in out MESSAGE_CARRIER);
  -- PRECONDITION : CARRIER is empty or carries a message.
  -- POSTCONDITION : CARRIER is empty.
  procedure RECEIVE (MESS : out MESSAGE; CARRIER : in out
    MESSAGE_CARRIER);
  -- PRECONDITION : CARRIER carries a message.
  -- POSTCONDITION : The message carried by CARRIER is assigned to MESS
  -- without being removed from CARRIER.
  -- If not PRECONDITION
  -- then the caller waits until PRECONDITION;
  -- end if.
```

Private

```
-- implementation of type MESSAGE_CARRIER
end BROADCASTING;
```

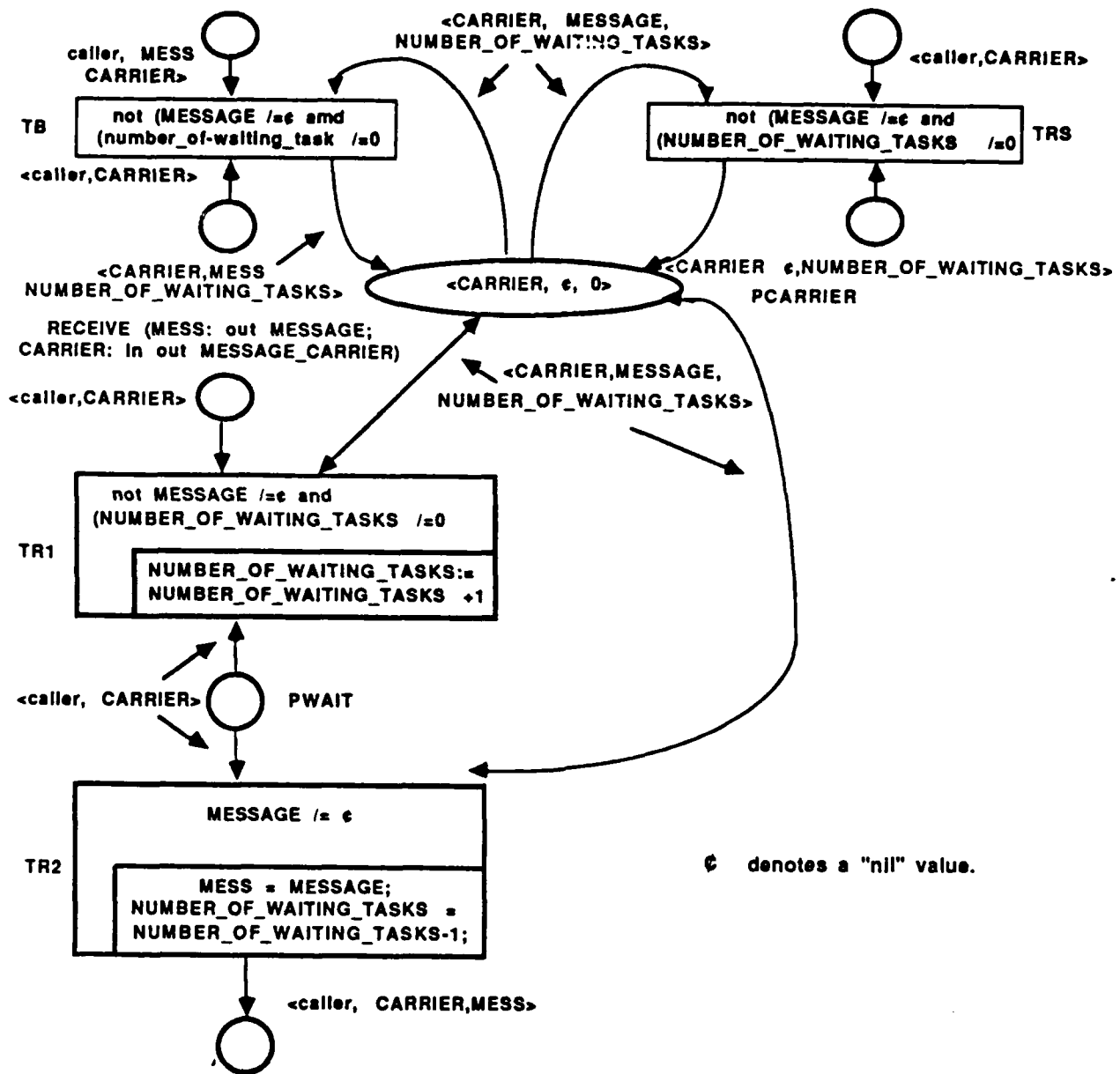
Note :

If the type T of objects to broadcast is limited private (for example, T is a task type), then one should use an access type on T as the type MESSAGE :

```
type ACCESS_T is access T;
Package T_BROADCASTING is new BROADCASTING (ACCESS_T);
```

BROADCAST (MESS_MESS;
CARRIER_In out MESSAGE_CARRIER)

RESET (CARRIER: In out
MESSAGE_CARRIER)



Abstract data type BROADCASTING

Synchronisation Specification

The properties defined by this Pr-T net are the following:

- a) The four operations TB, TRS, TR1, and TR2 are indivisible (1 transition each) and exclusive on a given carrier.
- b) The sequence (TR1, TR2) is indivisible if a MESSAGE is present (MESSAGE $\neq \emptyset$) when TR1 is fired.

The present property is equivalent to the following:

If TR2 is firable, then none of the transitions TB, TRS, TR1 is firable.

- c) Procedure BROADCAST loads, with no unbounded wait (without having to wait anything else than another transition completing its firing), the message MESS onto the CARRIER.

This property is established as follows:

- if TB is fired, then the place PCARRIER gets MESS in its message field (trivial)
- If TB has a token in its input place, then TB is firable after a finite delay (i.e without having to wait possible delivering of tokens onto the input places of TRS and TR1):
 - Initially, TB is immediately firable if a token is delivered in its input place (MESSAGE = $\emptyset$, NUMBER_OF_WAITING_TASKS = 0)
 - It is impossible that the condition MESSAGE $\neq \emptyset$ and (NUMBER_OF_WAITING_TASKS $\neq 0$) indefinitely holds, since TR2 is then firable. This is due to the fact that the following invariant holds (trivial):
number of tokens on PWAIT = NUMBER_OF_WAITING_TASKS
- d) Procedures RESET unloads, with no unbounded wait, the message which is present on the CARRIER passed as a parameter (if a message was present).

The "no unbounded wait" condition is due to the same reason as for procedure BROADCAST.

- e) Procedure RECEIVE:

If a message is present, then the content of this message is assigned to the parameter MESS with no unbounded wait; else the caller waits on PWAIT until a message is broadcast.

This property is a direct consequence of property b), because it is impossible that the condition MESSAGE $\neq \emptyset$ and (NUMBER_OF_WAITING_TASKS $\neq 0$) indefinitely holds, but it is possible that MESSAGE $\neq \emptyset$ indefinitely holds (no firing of RESET).

The properties defined by this Pr-T net are the following :

- a) The four operations TB, TRS, TR1, TR2 are indivisible (1 transition each)
- b) The sequence (TR1, TR2) is indivisible if a MESSAGE is present (MESSAGE $\neq \emptyset$) when TR1 is fired.

The present property is equivalent to the following :

If TR2 is firable, then none of the transitions TB, TRS, TR1 is firable.

- c) Procedure BROADCAST loads, with no unbounded wait (without having to wait anything else than another transition completing its firing), the message MESS onto the CARRIER.

This property is established as follows :

- If TB is fired, then the place PCARRIER gets MESS in its message field (trivial)
- If TB has a token in its input place, then TB is firable after a finite delay (i.e. without having to wait possible delivering of tokens onto the input places of TRS and TR1) :
 - . Initially, TB is immediately firable if a token is delivered in its input place (MESSAGE = $\emptyset$, NUMBER_OF_WAITING_TASKS = 0)
 - . It is impossible that the condition MESSAGE $\neq \emptyset$ and (NUMBER_OF_WAITING_TASKS $\neq 0$) indefinitely holds, since TR2 is then firable. This is due to the fact that the following invariant holds (trivial) :
number of tokens on PWAIT = NUMBER_OF_WAITING_TASKS
- d) Procedure RESET unloads, with no unbounded wait, the message which is present on the CARRIER passed as a parameter (if a message was present).

The "no unbounded wait" condition is due to the same reason as for procedure BROADCAST.

- e) Procedure RECEIVE :
If a message is present, then the content of this message is assigned to the parameter MESS with no unbounded wait ;
else the caller waits on PWAIT until a message is broadcast.

This property is a direct consequence of property b), because it is impossible that the condition MESSAGE $\neq \emptyset$ and (NUMBER_OF_WAITING_TASKS $\neq 0$) indefinitely holds, but it is possible that MESSAGE $\neq \emptyset$ indefinitely holds (no firing of RESET).

4 - IMPLEMENTATION OF PACKAGE BROADCASTING

The Pr-T net on next page gives the principle of this implementation. Place PCARRIER on the specification Pr-T net is developed (implemented) as a server task type MESSAGE_CARRIER (central part of the Pr-T net).

Setting the initial token onto place PCARRIER on the specification Pr-T net is implemented by sending a <CARRIER_name> token when a CARRIER object is created onto the MESSAGE_CARRIER body input place ("MESSAGE_CARRIER object creation") and by initializing the local data (MESSAGE_PRESENT := FALSE; NUMBER_OF_WAITING_TASKS := 0).

Taking off the <CARRIER_name> token when a CARRIER object must end as a consequence of its parent unit termination is implemented by the couple of places "PTERMINATE" and "PEND" :

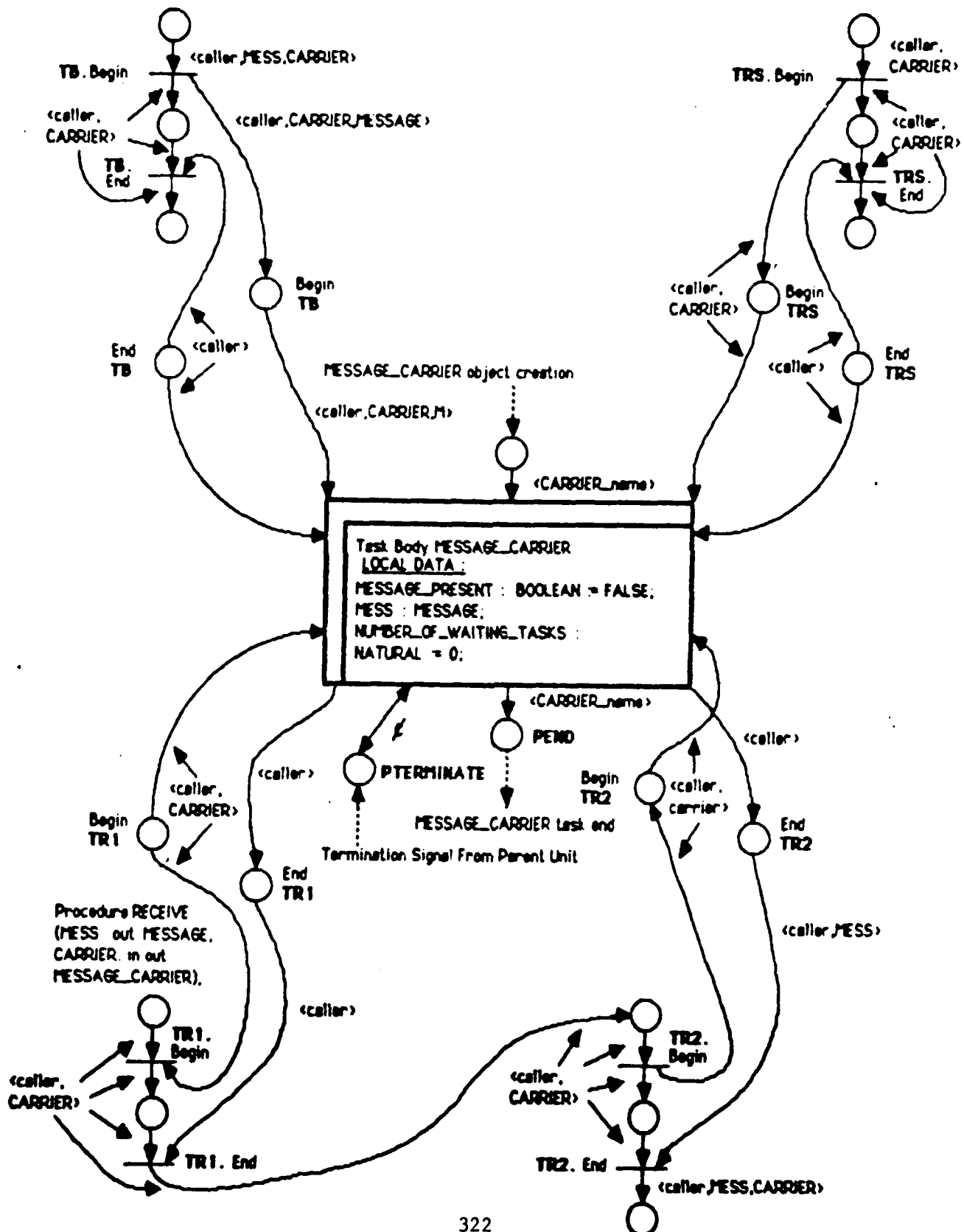
When the parent unit is ready to terminate, it sends a ϕ token onto place "PTERMINATE" ; as a consequence the <CARRIER-name> token is attracted onto place "PEND" from where it is eventually taken off by the terminating parent unit.

Access to place PCARRIER by transitions TB, TRS, TR1, TR2 is implemented as :

- access to places Begin TB and End TB by transitions TB.begin and TB.end,
- access to places Begin TRS and End TRS by transitions TRS.begin and TRS.end,
- access to places Begin TR1 and End TR1 by transitions TR1.begin and TR1.end,
- access to places Begin TR2 and End TR2 by transitions TR2.begin and TR2.end.

Procedure BROADCAST (MESS: MESSAGE;
CARRIER: in out MESSAGE_CARRIER);

Procedure RESET
(CARRIER: in out MESSAGE_CARRIER);



In terms of PETRI net theory, we can say that we have a morphism between the Specification Pr-T net and the net on previous page representing the implementation principle with the following relations :

Specification		Implementation Principle
Transition TB	→	Transitions TB.begin, TB.end
Transition TRS	→	Transitions TRS.begin, TRS.end
Transition TR1	→	Transitions TR1.begin, TR1.end
Transition TR2	→	Transitions TR2.begin, TR2.end
Place PCARRIER	→	Set of places (Begin TB, End TB, Begin TRS, End TRS, Begin TR1, End TR1, Begin TR2, End TR2)

The four services for which the MESSAGE_CARRIER type tasks are responsible have the following specification :

- TB service :
 Pre-condition : not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0));
 Post-condition : MESS (internal data) := MESS (parameter);
 MESSAGE_PRESENT := TRUE;
- TRS service :
 Pre-condition : not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0));
 Post-condition : MESSAGE_PRESENT := FALSE;
- TR1 service :
 Pre-condition : not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0));
 NUMBER_OF_WAITING_TASKS = X;
 Post-condition : NUMBER_OF_WAITING_TASKS := X + 1;
- TR2 service :
 Pre-condition : MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0);
 NUMBER_OF_WAITING_TASKS = X /= 0;
 Post-condition : MESS (parameter) := MESS (internal data);
 NUMBER_OF_WAITING_TASKS := X - 1;

The complete implementation is given by the next Pr-T net. We can see on this net the detail of task body MESSAGE_CARRIER consisting of a "for-ever" loop containing a selective wait with 3 open alternatives (TB, TRS, TR1 services) or 1 open alternative (TR2 service) depending upon the value of predicate MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0).

On each waiting point, the block or program unit which created tasks of MESSAGE_CARRIER type can force these tasks to terminate (one must therefore have as many instantiations of this part of the net (task body MESSAGE_CARRIER) as blocks or program units creating static tasks of type MESSAGE_CARRIER or declaring an ACCESS type on the type MESSAGE_CARRIER).

This task body is directly implementable in Ada as a task body containing a `SELECT` statement with 4 `ACCEPT` alternatives (the 4 services) and a `TERMINATE` alternative (see in ref. BOND83 the nominal Pr-T net model of the Ada select statement).

Following the Pr-T net on next page, we therefore give :

- the private part of Package `BROADCASTING` giving the Ada external specification of task type `MESSAGE_CARRIER`
- the Package body `BROADCASTING` containing the body of the three procedures `BROADCAST`, `RESET`, `RECEIVE` and the task body `MESSAGE_CARRIER`.

The Ada task type MESSAGE_CARRIER specification (private part of Package BROADCASTING) is the following :

```
task type MESSAGE_CARRIER is
  entry BEGIN_RECEIVE;
    — called by procedure RECEIVE to start waiting a message.
  entry END_RECEIVE (M : out MESSAGE);
    — called by procedure RECEIVE after the call to BEGIN_RECEIVE
    — in order to wait and receive the message into M.
  entry BROADCAST (M : MESSAGE);
    — called by procedure BROADCAST to broadcast the message M and
    — wake up the waiting tasks.
  entry RESET;
    — called by procedure RESET in order to unload the MESSAGE_CARRIER
    — which consequently is empty.
end MESSAGE_CARRIER;
```


Package body BROADCASTING is the following :

```
Package body BROADCASTING is
  task body MESSAGE_CARRIER is
    MESSAGE_PRESENT : BOOLEAN := FALSE;
    MESS : MESSAGE;
    NUMBER_OF_WAITING_TASKS : NATURAL := 0;
  begin
    loop
      select
        when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) =>
          accept BROADCAST (M : MESSAGE) do
            MESS := M;
          end BROADCAST;
          MESSAGE_PRESENT := TRUE;
        or
          when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) => accept RESET;
          MESSAGE_PRESENT := FALSE;
        or
          when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) => accept BEGIN_RECEIVE;
          NUMBER_OF_WAITING_TASKS := NUMBER_OF_WAITING_TASKS + 1;
        or
          when MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0) =>
            accept END_RECEIVE (M : out MESSAGE) do
              M := MESS;
            end END_RECEIVE;
            NUMBER_OF_WAITING_TASKS := NUMBER_OF_WAITING_TASKS - 1;
        or terminate;
      end select;
    end loop;
  end MESSAGE_CARRIER;
  procedure BROADCAST (MESS : MESSAGE; CARRIER : in out
    MESSAGE_CARRIER) is
  begin
    CARRIER.BROADCAST (MESS);
  end BROADCAST;
  procedure RESET (CARRIER : in out MESSAGE_CARRIER) is
  begin
    CARRIER.RESET;
  end RESET;
  procedure RECEIVE (MESS : out MESSAGE; CARRIER : in out
    MESSAGE_CARRIER) is
  begin
    CARRIER.BEGIN_RECEIVE;
    CARRIER.END_RECEIVE (MESS);
  end RECEIVE;
end BROADCASTING;
```

5 - VERIFICATION AND VALIDATION OF PACKAGE BROADCASTING

5.1 Proving the Correctness of the Implementation against the Specification

Obtaining semantic models of Ada tasking features by Pr-T nets has been the subject of a previous work (ref. BOND83).

These models were used to map the Pr-T net representing our implementation into the Ada package.

We will assume that this operation was correctly performed and that the Ada package is a perfect image of the implementation Pr-T net.

We will therefore restrict ourselves to prove the correctness of the implementation Pr-T net against the specification Pr-T net. To do that, we must verify that our implementation Pr-T net preserves the a, b, c, d, e properties which were identified on the specification Pr-T net.

- Notations for Transition Firing Sequences:

T1, T2, T3: firing of T1, then T2, then T3 sequentially.

T1/T2/T3 : firing of T1, T2, T3 concurrently.

(T1) excl (T2) excl (T3) : T1, T2, T3 are mutually exclusive.

- Property a : the four operations TB, TRS, TR1, TR2 are indivisible and exclusive on a given carrier

Specification	Implementation
TB	TB1, TC2, (TB2//TC3)
TRS	TRS1, TC4, (TRS2//TC%)
TR1	TR11, TC6, (TR12//TC7)
TR2	TR21, TC9, (TR22//TC10)

If we consider the sequences above, we can see that (TC2, TC3) excl (TC4, TC5) excl (TC6, TC7) excl (TC9, TC10) for a given MESSAGE_CARRIER (1 single token is available). It follows that the above sequences are indeed indivisible and exclusive on a given carrier after they have fired their first transition (TB1 or TRS1 or TR11 or TR21) which is a pure synchronization.

- Property b : if TR2 is firable, then none of the transitions TB, TRS, TR1 is firable.

TC1 must have been fired before TC2 (for TB) or TC4 (for TRS) or TC6 (for TR1) is firable and TC8 must have been fired before TC9 (for TR2) is firable. For a given MESSAGE_CARRIER, TC1 and TC8 are in effective conflict (same input place PC1 and one single token at most on it for given MESSAGE_CARRIER) and their firing condition (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS !=0) holding or not) is opposite.

5 - VERIFICATION AND VALIDATION OF PACKAGE BROADCASTING

5.1. Proving the Correctness of the Implementation against the Specification

Obtaining semantic models of Ada tasking features by Pr-T nets has been the subject of a previous work (ref. BOND83).

These models were used to map the Pr-T net representing our implementation into the Ada package.

We will assume that this operation was correctly performed and that the Ada package is a perfect image of the implementation Pr-T net.

We will therefore restrict ourselves to prove the correctness of the implementation Pr-T net against the specification Pr-T net.

To do that, we must verify that our implementation Pr-T net preserves the a, b, c, d, e properties which were identified on the specification Pr-T net.

- Notations for Transition Firing Sequences :

T1, T2, T3 : firing of T1, then T2, then T3 sequentially.

T1//T2//T3 : firing of T1, T2, T3 concurrently.

(T1) excl (T2) excl (T3) : T1, T2, T3 are mutually exclusive.

- Property a : the four operations TB, TRS, TR1, TR2 are indivisible

Specification		Implementation
TB	—————>	TB1, TC2, (TB2//TC3)
TRS	—————>	TRS1, TC4, (TRS2//TC5)
TR1	—————>	TR11, TC6, (TR12//TC7)
TR2	—————>	TR21, TC9, (TR22//TC10)

If we consider the sequences above, we can see that (TC2, TC3) excl (TC4, TC5) excl (TC6, TC7) excl (TC9, TC10) for a given MESSAGE_CARRIER (1 single token is available).

It follows that the above sequences are indeed indivisible after they have fired their first transition (TB1 or TRS1 or TR11 or TR21) which is a pure synchronization.

- Property b : if TR2 is firable, then none of the transitions TB, TRS, TR1 is firable

TC1 must have been fired before TC2 (for TB) or TC4 (for TRS) or TC6 (for TR1) is firable and TC8 must have been fired before TC9 (for TR2) is firable.

For a given MESSAGE_CARRIER, TC1 and TC8 are in effective conflict (same input place PC1 and one single token at most on it for a given MESSAGE_CARRIER) and their firing condition (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0) holding or not) is opposite.

Therefore, for a given MESSAGE_CARRIER, the presence of the token on PC2 and PC3 is mutually exclusive and TC2 (for TB), TC4 (for TRS), TC6 (for TR1) are not firable (no token on PC2) if TC9 (for TR2) is firable (the token is on PC3).

- Property c : procedure BROADCAST loads, with no unbounded wait, the message MESS onto the CARRIER

Execution of procedure BROADCAST corresponds to the firing sequence :

TB $\longrightarrow$ TB1, TC2, (TC3//TB2)

This sequence enables one to send the message MESS onto the CARRIER because :

- . TC2 assigns the message MESS (parameter of BROADCAST) to the message MESS (internal to the CARRIER)
- . TC3 assigns, internally to the CARRIER, MESSAGE_PRESENT := TRUE

We still have to show that there is no unbounded wait.

The only possibility of unbounded wait after the firing of TB1 (which is firable as soon as procedure BROADCAST is called) in the sequence above is on TC2.

As, by hypothesis, TC12 or TC13 are never firable (no token put onto PTERMINATE) if the CARRIER is still visible, an unbounded wait on TC2 can correspond only to a case where the token of the CARRIER is waiting on the place PC3 (PC3 is the only place where a wait is possible besides PC2 and the presence of the token on PC2 when TB1 is fired makes TC2 firable).

We are going to show that it is not possible for the token of a CARRIER to stay indefinitely waiting on PC3 :

- . It is easy to see that each incrementing of NUMBER_OF_WAITING_TASKS (possible only by firing TC7) corresponds to a sequence of transitions : (TC1//TR11), TC6, ((TC7, TC11)//(TR12, TR21)). Firing such a sequence puts one token on PR3 and PR6. We finally have :
sum of tokens going onto PR6 = NUMBER_OF_WAITING_TASKS for a given CARRIER.

Therefore, if MESSAGE_PRESENT then becomes true for this CARRIER, TC8 becomes firable and the sum of tokens on PR6, or about to reach it, for the given CARRIER is equal to NUMBER_OF_WAITING_TASKS.

- . We have then firing cycles :
(TC8, TC9, ((TC10 + TC11)//TR22))
without any waiting condition on PC3 until
NUMBER_OF_WAITING_TASKS becomes zero again.
This is due to the fact that, in such a firing cycle :
- NUMBER_OF_WAITING_TASKS is decremented once,
- one token is taken from PR6,
- MESSAGE_PRESENT is never assigned (and remains true) ;
therefore TC8 is always firable (and TC1 is not) when the CARRIER token reaches PC1.

When `NUMBER_OF_WAITING_TASKS` becomes zero again, then it is TC1, and no longer TC8, which is firable when the `CARRIER` token reaches PC1 and it is therefore no longer possible to go onto PC3.

Finally, we have thus demonstrated the impossibility to reach an unbounded waiting condition on PC3.

- Property d : procedure `RESET` unloads, with no unbounded wait, the message which was present on the `CARRIER` (if a message was present)

Execution of procedure `RESET` corresponds to the firing sequence :

`TRS -> TRS1, TC4, (TRS2//TC5).`

This sequence enables one to unload the message which was present on the `CARRIER` since TC5 assigns `MESSAGE_PRESENT := FALSE`.

The fact that there is no unbounded wait can be shown the same way it was for procedure `BROADCAST`.

- Property e : procedure `RECEIVE` : if a message was present, then the value of this message is assigned, with no unbounded wait, to the parameter `MESS`, else the caller waits until a message is broadcast

If callers of `RECEIVE` were already waiting when the message is broadcast, we have already seen that they are woken up without any further wait (cf. the demonstration of property c). Therefore, we suppose that there is no caller of `RECEIVE` waiting when the message is broadcast and loaded onto the `CARRIER` (which implies that `NUMBER_OF_WAITING_TASKS = 0` for this `CARRIER`).

Two cases are then possible when procedure `RECEIVE` (transition TR11) begins to execute :

- . either the `CARRIER` token was on PC2 and the following sequence :
`TC6, ((TC7, TC11, TC8)//(TR12, TR21)), TC9, ((TC10, TC11)//TR22)`
is immediately firable and TC9 assigns the internal value `MESS` to the parameter `MESS` with no wait ;
- . or the `CARRIER` token was not on PC2, but then it could only be on PC4, PC5, PC6, PC8 or PC1 since `NUMBER_OF_WAITING_TASKS` was 0 (TC8 not firable) by hypothesis ;
the `CARRIER` token was then about to come back onto PC2 since from all the places listed above it could do nothing but come back onto PC2 with no wait, TC8 being not firable.
This case is therefore similar to the first one.

If the message was not present when procedure RECEIVE (transition TR11) begins to execute, then : TC8 is not firable (MESSAGE PRESENT = FALSE) and the CARRIER token can only be on PC2 or about to come back onto PC2 (see above).

After TR11, one can fire :

TC6, ((TC7, TC11, TC1)/(TR12, TR21)) which has the following consequences :

- . the RECEIVE caller waits on PR3,
- . NUMBER_OF_WAITING_TASKS is incremented by 1 (TC7).

5.2. Deriving a Test Set from the Semantic Specification

From the specification (Ada specification plus the Pr-T net defining the synchronization specification), we have defined the five properties a, b, c, d, e which package BROADCASTING must preserve.

Then it is quite straight forward to derive a program which successively checks each of these properties with the help of a good debugger (the program initiates the test actions on an instantiation of package BROADCASTING and all the observations are made through the debugger).

The program on next pages is an example of such a test-program for package BROADCASTING.

All the lines of comment in executable parts of this program define observations that must be performed through the debugger.

If no convenient debugger is available, then the test program is much more complicated because it must perform itself all the observations which are normally done through the debugger.

Note that such a test set is based solely on the specification : during the test phase, we do not bother with the implementation details, and we only aim at checking that the package behaves in compliance with its specification.

```

with BROADCASTING;
procedure BROADCASTING_TEST is
  subtype LINE_80 is STRING (1..80);
  package LINE_80_BROADCASTING is new BROADCASTING (LINE_80);
  use LINE_80_BROADCASTING;
  MY_MESSAGE : constant LINE_80 := "I am the message" & (17..80 =>
    ' ');
  M_C : MESSAGE_CARRIER;
  task type BROADCASTER; -- broadcasts MY_MESSAGE on M_C
  task type RECEIVER; -- receives a message on M_C
  task type RESETTER; -- resets M_C
  task body BROADCASTER is
  begin
    BROADCAST (MY_MESSAGE, M_C);
  end BROADCASTER;
  task body RECEIVER is
    LOCAL_MESSAGE : LINE_80 := (1..80 => '*');
  begin
    RECEIVE (LOCAL_MESSAGE, M_C);
    null; -- breaking here enables one to evaluate LOCAL_MESSAGE
  end RECEIVER;
  task body RESETTER is
  begin
    RESET (M_C);
  end RESETTER;
begin
  A_TEST : -- tests the property a
  declare
    BROADCAST_1, BROADCAST_2 : BROADCASTER;
    RECEIVE_1, RECEIVE_2 : RECEIVER;
    RESET_1, RESET_2 : RESETTER;
  begin
    -- check by a step by step execution using the debugger that none
    -- of the "TB", "TRS", "TR1", "TR2" operations of the tasks above
    -- are interleaved and execute the following, while keeping
    -- checking that, to be sure to terminate this block :
    delay 1.0;
    C_TEST : -- tests the property c
    begin
      RESET (M_C);
      BROADCAST (MY_MESSAGE, M_C);
      -- check here that MY_MESSAGE is present on M_C.
    end C_TEST;
  end A_TEST;
  B_TEST : -- checks the property b
  declare
    BROADCAST_1, BROADCAST_2 : BROADCASTER;
    RECEIVE_1, RECEIVE_2 : RECEIVER;
  begin
    -- check that the sequence (TR1, TR2) in RECEIVE_1 and RECEIVE_2
    -- is not interleaved with anything else.
    null;
  end B_TEST;

```

```

D_TEST : — checks the property d
begin
  RESET (M_C);
  — check that the message is no longer present on M_C.
  RESET (M_C);
  — check that M_C state remains invariant.
end D_TEST;
E_TEST : — checks the property e
declare
  RECEIVE_1 : RECEIVER;
begin
  delay 1.0;
  — check that RECEIVE_1 waits on PWAIT for a message to be
  — broadcast.
  BROADCAST (MY_MESSAGE, M_C);
  — check that RECEIVE_1.LOCAL_MESSAGE receives MY_MESSAGE and that
  — RECEIVE_1 terminates.
end E_TEST;
end BROADCASTING_TEST;
pragma MAIN;

```

6 - CONCLUSIONS

The main advantages of our verification and validation technique appear to be the following :

- . The same modelling technique (Pr_T nets) is kept from specification to implementation which makes it easier to prove that the implementation is correct with respect to the specification.
- . A test set can be derived straightforwardly from the specification without any need to consider the implementation details.
- . The implementation being proved correct before testing is performed, only minor bugs (the rare typos which cannot be detected by the Ada compiler, omission of a line of code, ...), due to the fact that the implementation process remains manual, should remain when testing begins.
The cost of testing should then dramatically drop down.
As a matter of fact, no bug was found when we performed the tests defined in section 5.2. for package BROADCASTING on our Ada environment.

The main shortcomings are the following :

- . The style of proof we presented is not totally satisfactory. We already said in section 1 that some on-going research works (ref. GL83, PL83, KL84, VM84) make us hope that we will be able in a few years to undertake more formal mathematical proofs on Pr-T nets in many practical cases.
- . We did not say anything in this paper about verification of specifications themselves. However, it is possible to verify such properties as coherence and completeness on specifications.

REFERENCES :

- LRM : Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815A, January 1983, DoD

- ANNA : ANNA
A Language for Annotating Ada Programs.
Preliminary Reference Manual, June 1984
D.C. LUCKHAM, F.W. von HENKE, B. KRIEG-BRUECKNER, O. OWE,
Stanford University

- MP82 : Z. MANNA and A. PNUELI
Verification of Concurrent Programs : The Temporal Framework,
in :
The Correctness Problem in Computer Science,
International Lecture Series in Computer Science,
Academic Press LONDON, 1982

- BKP84 : H. BARRINGER, R. KUIPER, A. PNUELI
Now You May Compose Temporal Logic Specifications,
Proceedings of the 16th ACM Symposium on the Theory of
Computing, WASHINGTON, 1984

- PET77 : J.L. PETERSON : PETRI Nets
ACM Computing Surveys Vol. 9 No 3, September 1977

- NET80 : Net Theory and Applications, Proceedings of the Advanced
Course on General Net Theory of Processes and Systems (HAMBURG
1979).
Lecture Notes in Computer Science n° 84, SPRINGER-VERLAG 1980

- BRA82 : G.W. BRAMS (Collective name)
Réseaux de PETRI : Théorie et Pratique
Editor : MASSON in PARIS, 1982

- GL79 : H.J. GENRICH, K. LAUTENBACH :
The Analysis of Distributed Systems by means of
Predicate/Transition Nets,
in Semantics of Concurrent Computation,
Lecture Notes in Computer Science n° 70, SPRINGER-VERLAG 1979

- GL80 : H.J. GENRICH, K. LAUTENBACH, P.S. THIAGARAJAN :
Elements of General Net Theory, in ref. NET80

- BOND83 : P. de BONDELI :
Models for the Control of Concurrency in Ada based on Pr-T
nets
Proceedings of the Adatec-Ada/Europe Joint Conference on Ada -
Edited by the Commission of the European Communities,
BRUSSELS, March 1983

- GL83 : H.J. GENRICH, K. LAUTENBACH :
S-invariance in Predicate/Transition Nets, in Application and
Theory of PETRI Nets, Informatik Fachberichte 66,
SPRINGER-VERLAG 1983

- PL83 : A. PAGNONI, K. LAUTENBACH :
Liveness and Duality in Marked-Graph-Like Pr/T Nets, in the
Proceedings of the 4th European Workshop on Application and
Theory of PETRI Nets, Edited by CNRS/LAAS-TOULOUSE, 1983

- KL84 : R. KUJANSUU, M. LINDQVIST :
Efficient Algorithms for Computing S-invariants for
Predicate/Transition Nets, in the Proceedings of the 5th
European Workshop on Application and Theory of PETRI Nets,
Edited by the AARHUS University, DENMARK in 1984

- VM84 : J. VAUTHERIN, G. MEMMI :
Computation of flows for Unary Predicate/Transition Nets, in
the Proceedings of the 5th European Workshop on Application
and Theory of PETRI Nets, Edited by the AARHUS University,
DENMARK in 1984

- MZGT85 : D. MANDRIOLI, R. ZICARI, C. GHEZZI, F. TISATO :
Modeling the Ada Task System by PETRI Nets, in Comput. Lang.
vol. 10 n° 1, 1985

- SC85 : S.M. SHATZ, W.K. CHENG :
Static Analysis of Ada Programs using the PETRI Net Model, in
IEEE Proceedings of ISCAS, 1985

A P P E N D I X

PREDICATE-TRANSITION NETS

1. INTRODUCTION

Predicate-Transition Nets were first introduced in (Ref Q.79). They are an evolution of Place-Transition Nets (the classical PETRI Nets).

They allow a more concise modelling of systems.

A place in a Predicate-Transition Net can model several places of an equivalent Place-Transition Net and can have tokens which are "coloured" by tuples of data which can be constants or variables.

Similarly, a transition in a Predicate-Transition Net can model several transitions of an equivalent Place-Transition Net.

Arcs are valued by tuples of constants and variables defining the set of tokens produced or consumed by a transition on a place.

On each transition a logical expression specifies the relations involving the different tokens which must hold to enable the transition.

This expression is omitted if it is uniformly true.

2. PREDICATE-TRANSITION NETS VERSUS PLACE-TRANSITION NETS TO MODEL ADA PROGRAMS

Many features of ADA indicate that Predicate-Transition Nets are a much more adequate tool than Place-Transition Nets to model tasking constructs of ADA, such as:

- Task Types : Many different task objects can be defined on the same Task Type and have concurrent executions. The Predicate-Transition Nets allow to have a single net for the task body and to designate each task object by the "colour" of its token. Similarly, the entries of all task objects in the task type

will share the same set of places. An entry for a specific task object will be distinguished on these places by the colour of its token.

- Selective waits : The whole phase of selecting the branch to be executed can be gathered on a single branch of net, no matter how many different select branches are present.

Even after the branch to be executed has been selected, the different branches possible can often keep being gathered in a single branch of net and being distinguished only by the colour of the tokens if the statements in these branches generate control structures having a common net model.

3. A FORMAL DEFINITION OF PREDICATE-TRANSITION NETS

A good introduction to Predicate-Transition Nets can be found in (Ref Q.80). The following definition is similar to that given in (Ref. Q.79) or (Ref Q.80).

Definition: A predicate-transition-net (PrT-net) consists of the following constituents:

1. A directed net (S, T, F) where
 - S is the set of predicates ('first-order' places) ,
 - T is the set of ('first-order') transitions ,
 - $F: \subseteq (S \times T) \cup (T \times S)$ is the set of arcs $\longrightarrow$.
2. A structured set $U = (U; op_1 \dots, op_m; p_1, \dots, p_n)$ with operators op_i and predicates p_j .

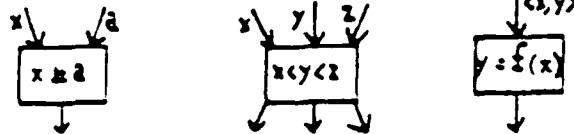
3. A labelling of arcs assigning to all elements of T a formal sum of n -tuples of variables where n is the 'arity' of the predicate connected to the arc. The zero-tuple indicating a no-argument predicate (an ordinary place) is denoted by the special symbol ϵ .

Examples :



4. An inscription on transitions assigning to some elements of T a logical formula built from equality, operators and predicates given with U ; variables occurring free in a transition have to occur at an adjacent arc.

Examples :



5. A marking of predicates of S with n -tuples of individual (items).

Examples :



6. A natural number K which is a universal bound for the number of copies of the same item which may occur at a single place (K may be called place capacity).

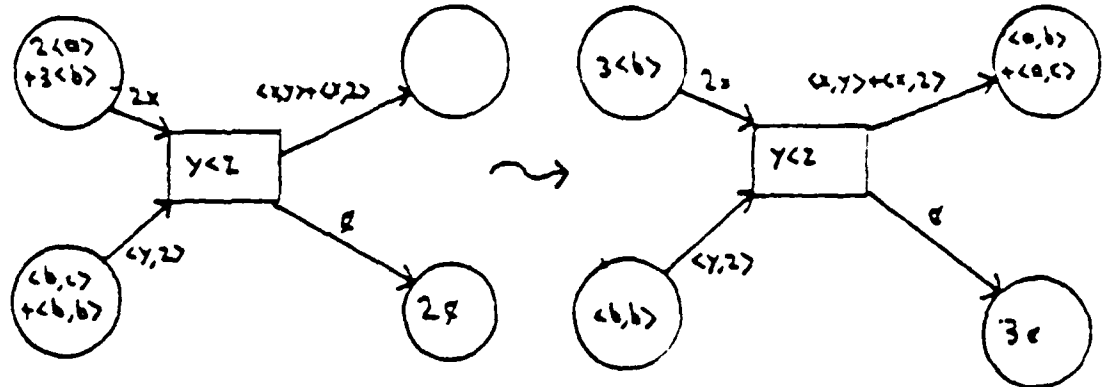
7. The transition rule " $\rightsquigarrow$ " which expresses the common interpretation of predicate-transition-nets:

Each element of T represents a class of possible changes of markings (ordinary transitions). Such an indivisible change consists of removing ($\bigcirc \rightarrow \square$) and adding ($\square \rightarrow \bigcirc$) copies of items from/to places according to the schemes expressed by the arc labels. It may occur whenever, for an assignment of individuals to the variables which satisfies the formula inscribed to the transition, all input places carry enough copies of proper items and for no output place the capacity K is exceeded by adding the respective copies of items.

Example: For a structure $((a,b,c)); < := \text{alphabetical ordering})$ and $K = 3$, two of the nine instances of the following transition are enabled under the marking shown on the left side:

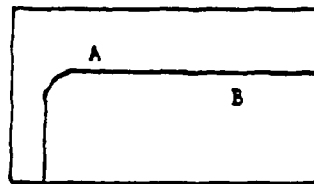
$(x,y,z) \leftarrow (a,b,c)$ and $(x,y,z) \leftarrow (b,b,c)$.

Due to conflict, however, at most one will occur. For the assignment $(x,y,z) \leftarrow (a,b,c)$ the resulting marking is shown on the right side.



4. SPECIFIC NOTATIONS AND PROPERTIES OF THE NETS WE USE

- Notations for transitions:



A: logical expression enabling the transition.

B: defines the operation modelled by firing the transition.



No operation is attached to this transition.



No operation is attached to this transition and the logical expression "A" is uniformly true.

- Notations for arcs:

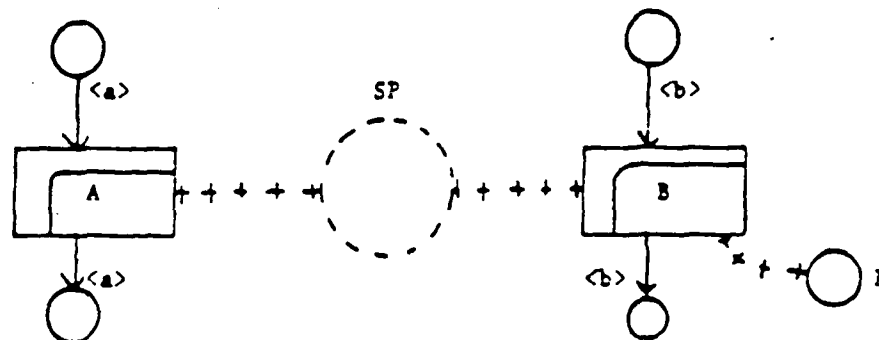


- Initial marking: It is represented by the tokens explicitly figured on the places.

- Indivisibility of Transitions:

Transitions are not all indivisible and may either:

- be indivisible, or:
 - model a "purely sequential" sequence of operations, or:
 - model a set of operations which are detailed on another net.
- When some transitions model sets of operations which are detailed on other nets, the following notations may be found:



- SP is a set of places which is further detailed on another net.
 - There are, on another more detailed net, arcs joining: A to the places of SP, B to the places of SP, B to the place P.
- Place Capacity: In our models, the places are supposed to have a capacity large enough to be never reached by tokens present on the places. Therefore we do not consider the place capacity.

**ON THE USE OF SEMANTIC SPECIFICATION
FOR THE VERIFICATION AND VALIDATION
OF REAL TIME SOFTWARE**

Patrick de Bondeli

CR2A

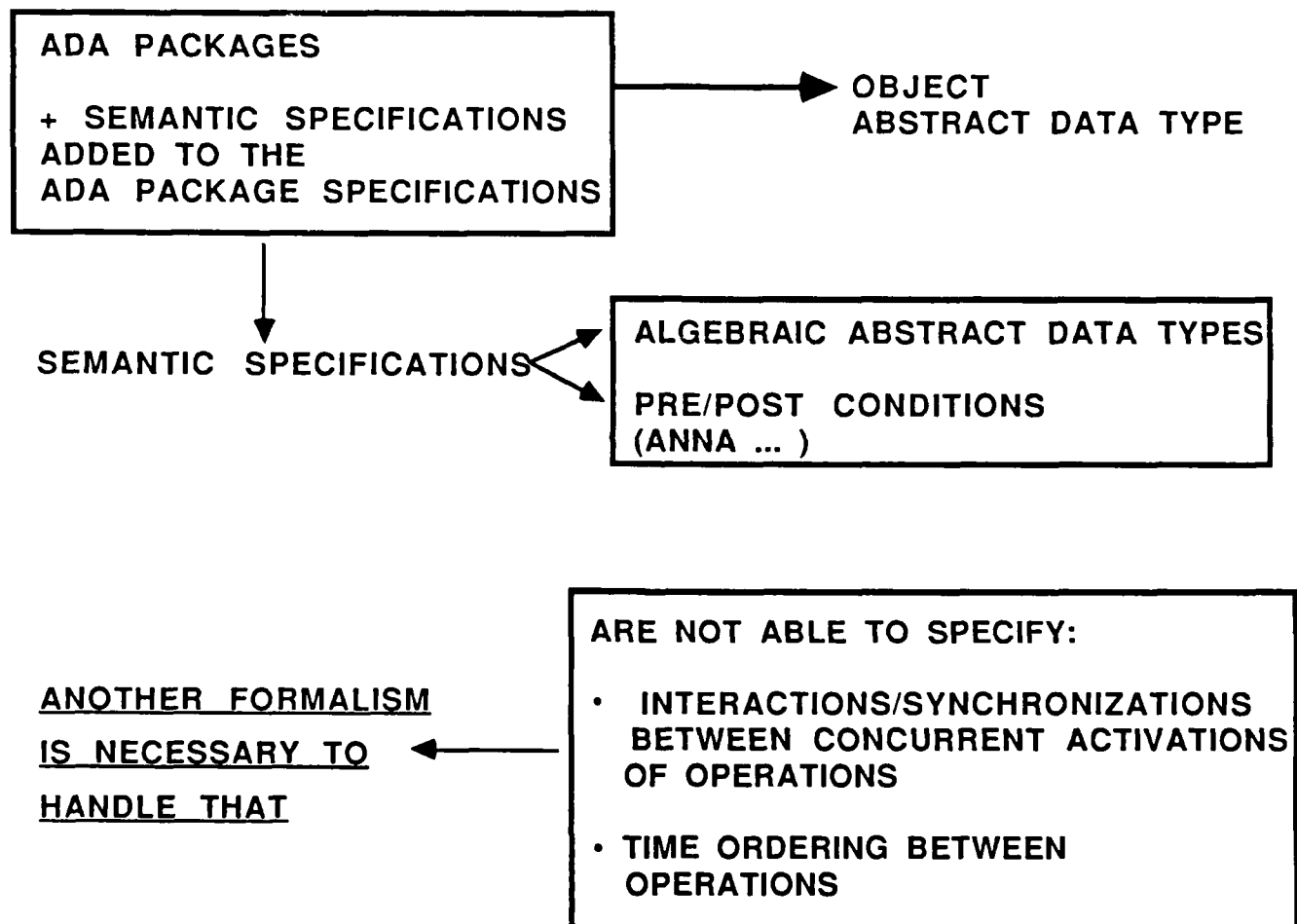
AEROSPATIALE/Space Division

FRANCE

PURPOSE OF THE PRESENTATION

**ILLUSTRATE THROUGH A SIMPLE
BUT REALISTIC EXAMPLE THE USE
OF SEMANTIC SPECIFICATIONS TO
IMPLEMENT, VERIFY AND VALIDATE
REAL TIME SOFTWARE.**

BACKGROUND



DIFFERENT FORMALISMS MAY BE USED TO SPECIFY INTERACTIONS/SYNCHRONIZATIONS BETWEEN CONCURRENT OPERATIONS AND TIME ORDERING; MOST NOTABLY:

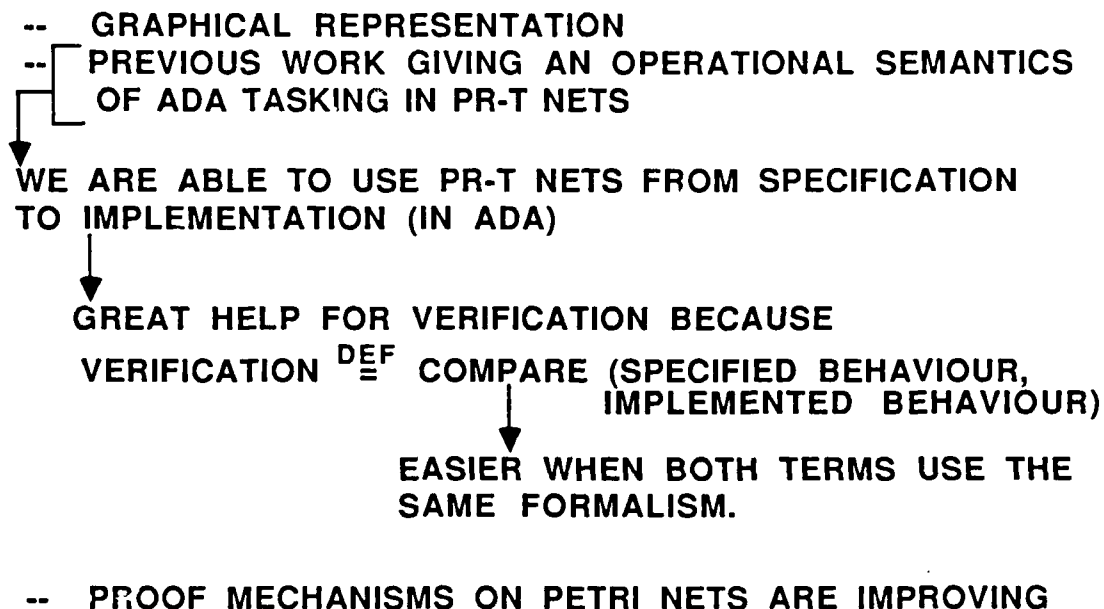
TEMPORAL LOGICS

PETRI NETS

OUR CHOICE:

PREDICATE - TRANSITION NETS (PR-T NETS) A HIGH LEVEL CLASS OF PETRI NETS.

MOTIVATIONS FOR THIS CHOICE:



CONTENTS FOR THE REMAINDER OF THE PRESENTATION

- RATIONALE AND REQUIREMENTS FOR OUR EXAMPLE:
PACKAGE BROADCASTING**
- A SPECIFICATION FOR PACKAGE BROADCASTING**
- AN ADA IMPLEMENTATION OF PACKAGE BROADCASTING**
- VERIFICATION AND VALIDATION OF PACKAGE BROADCASTING:**
 - PROVING THE CORRECTNESS OF THE
IMPLEMENTATION AGAINST THE SPECIFICATION**
 - DERIVING A TEST SET FROM THE SEMANTIC
SPECIFICATION**
- CONCLUSIONS**

RATIONALE AND REQUIREMENTS FOR PACKAGE BROADCASTING

CALLER

```

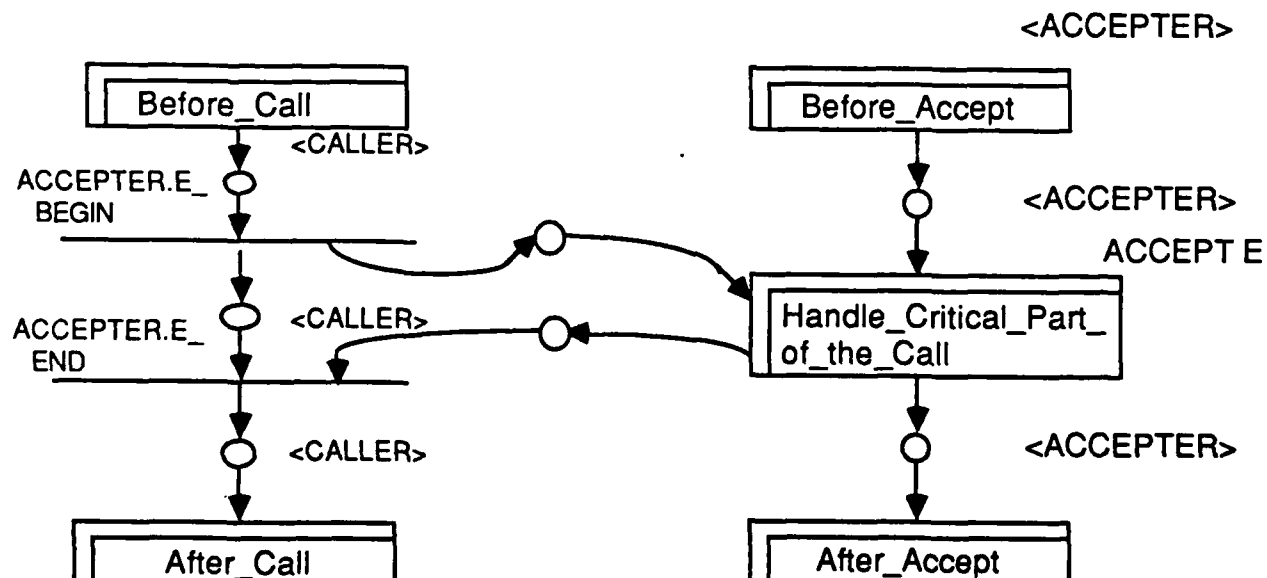
      .
      .
      .
-- Before_Call;
ACCEPTER.E;
-- After_Call;
      .
      .
      .
      .
  
```

ACCEPTER

```

Task ACCEPTER is
  entry E;
End ACCEPTER;
Task body ACCEPTER is
Begin
  -- Before_Accept;
  accept E do
    -- Handle_critical_
    -- part_of_the_call;
  end E;
End ACCEPTER;
  
```

<ACCEPTER>



**In Real Time Applications, another kind
of Synchronization is also useful:**

BROADCASTING

**A task has to broadcast a message on a "message_carrier"
to an unknown number of "receiving" tasks**

- **The "broadcasting" task does not want to wait,
no matter if some "receiving" tasks are not ready
to take the message.**
- **The "receiving" tasks, when they invoke the "receive"
primitive, wait until a message is present on the
message_carrier and then take all this same
message (which still keeps being present on the
message carrier after that).**
- **An additional primitive "Reset" allows the
broadcasting task to suppress the message on
the message carrier (thereby forcing subsequent
"receivers" to wait for a new message)**

SPECIFICATION OF PACKAGE BROADCASTING

ADA Specification

Generic

type MESSAGE **is private** ; -- type of message to broadcast

Package BROADCASTING is

type MESSAGE_CARRIER **is limited private**;

-- MESSAGE_CARRIERS are the vehicle for broadcasting messages.

-- A MESSAGE_CARRIER may be empty or may carry one message

-- It is initially empty.

procedure BROADCAST(MESS : MESSAGE; CARRIER : **in out**
MESSAGE_CARRIER);

-- **PRECONDITION**: CARRIER is empty or carries one message

-- **POSTCONDITION**: CARRIER carries the message MESS.

procedure RESET(CARRIER : **in out** MESSAGE_CARRIER);

-- **PRECONDITION**: CARRIER is empty or carries one message.

-- **POSTCONDITION** : CARRIER is empty.

procedure RECEIVE (MESS: **out** MESSAGE; CARRIER: **in out**
MESSAGE_CARRIER);

-- **PRECONDITION**: CARRIER carries one message.

-- **POSTCONDITION**: The message carried by CARRIER is assigned
-- to MESS but it is not removed from CARRIER.

-- If not **PRECONDITION**

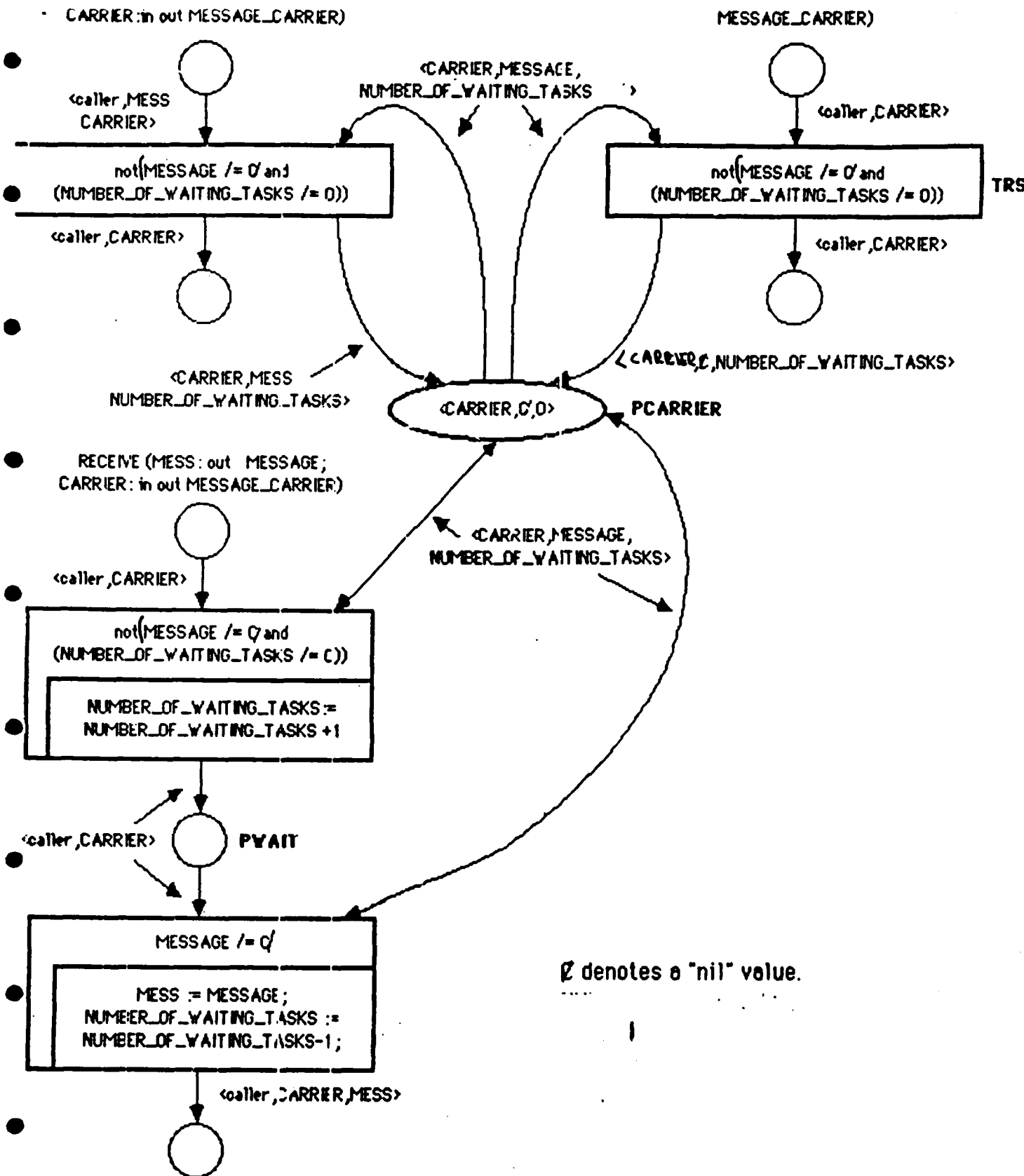
-- then the caller waits until **PRECONDITION**;

-- end if

PRIVATE

-- Implementation of type MESSAGE_CARRIER.

End BROADCASTING.



$\emptyset$ denotes a "nil" value.

Abstract data type BROADCASTING **Synchronisation Specification**

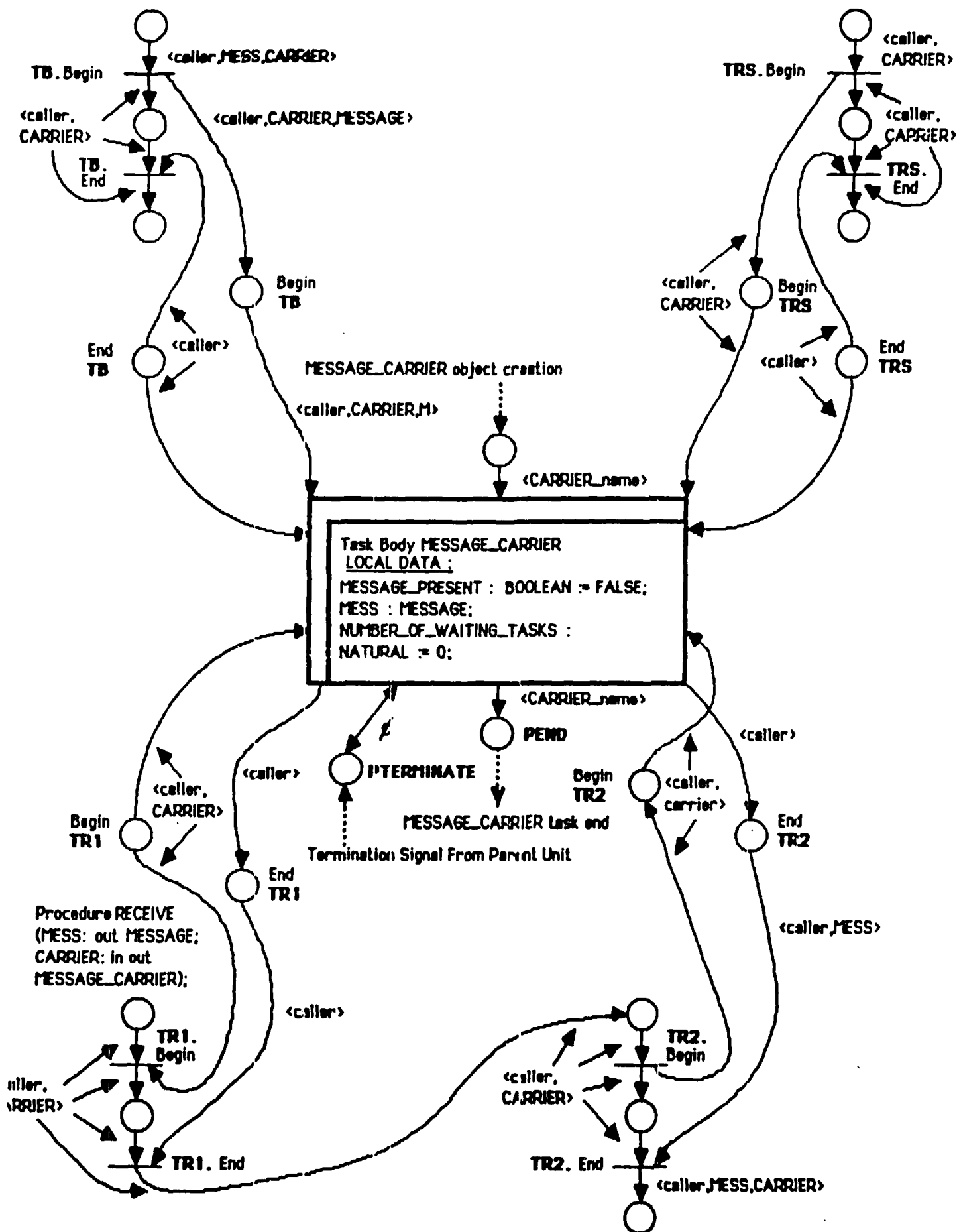
The properties defined by this Pr-T net are the following:

- a) TB, TRS, TR1, TR2 are indivisible (1 transition each) and exclusive (because each one takes the token from place PCARRIER).
- b) (TR1, TR2) is indivisible if $MESSAGE \neq \epsilon$ when TR1 is fixed.
i.e.: if TR2 is fixable, none of hte TB, TRS, TR1 is fixable.
- c) Procedure BROADCAST loads, with no unloaded wait, the message MESS onto the CARRIER:
 - if TB is fixed, then PCARRIER gets MESS in its message field
 - if TB has a token in its input place, then it is fixable after a finite delay.
 - Initially TB is immediately fixable if a token is delivered in its input place
 - It is impossible that the condition:
 $(MESSAGE \neq \epsilon)$ and $(number\_of\_waiting\_tasks \neq 0)$
indefinitely holds since TR2 is then fixable because always
 $number\ of\ taken\ on\ PWAITS = number\_of\_waiting\_tasks$
- d) Procedure RESET unloads, with no unbounded wait, the message which was present on the CARRIER
- e) Procedure RECEIVE:
If a message is present, then the content of this message is assigned to MESS with no unbounded wait (consequence of b), else the caller waits on AWAIT until a message is broadcast.

IMPLEMENTATION OF PACKAGE BROADCASTING

PRINCIPLE:

"Unfold" place PCARRIER into a server task having four entries ;(one for each of the 4 transitions connected to it) and "unfold" each of these transitions (TB, TRS, TR1, TR2) into a call to one of these entries. This principle is illustrated by the following slide.



Principle Of Implementation For The Abstract Data Type BROADCASTING

The 4 services for which the MESSAGE_CARRIER type tasks are responsible have the following specifications:

```
-- TB:
    PRECONDITION:  not (MESSAGE_PRESENT and
                       (NUMBER_OF_WAITING_TASKS/=0));
    POSTCONDITION: MESS (internal data):=MESS(parameter);
                  MESSAGE_PRESENT:=TRUE;

-- TRS:
    PRECONDITION:  not (MESSAGE_PRESENT and
                       (NUMBER_OF_WAITING_TASKS/=0));
    POSTCONDITION: MESSAGE_PRESENT:=FALSE;

-- TR1:
    PRECONDITION:  not(MESSAGE_PRESENT and
                       (NUMBER_OF_WAITING_TASKS/=0));
                  NUMBER_OF_WAITING_TASKS=X;
    POSTCONDITION: NUMBER_WAITING_TASKS:=X+1;

-- TR2:
    PRECONDITION:  MESSAGE_PRESENT and
                       (NUMBER_OF_WAITING_TASKS/=0);
                  NUMBER_OF_WAITING_TASKS=X/=0;
    POSTCONDITION: MESS(parameter):=MESS(internal data);
                  NUMBER_OF_WAITING_TASKS:=X-1;
```

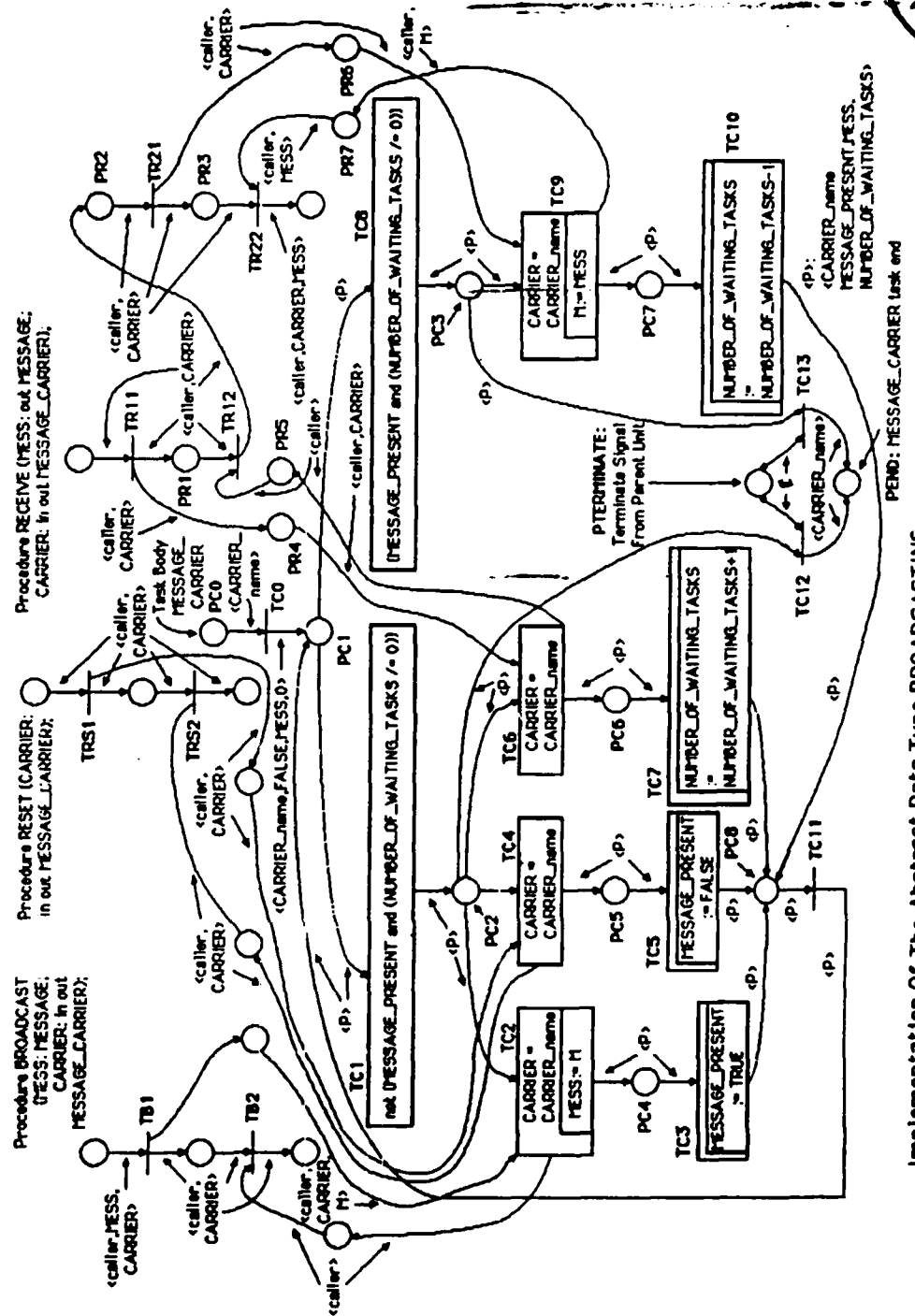
DETAIL OF IMPLEMENTATION

-- IT IS GIVEN BY THE PR-T NET ON THE FOLLOWING SLIDE.

-- THIS PR-T NET IS DIRECTLY IMPLEMENTABLE IN ADA:

- The MESSAGE_CARRIER task type specification gives the private part of Package BROADCASTING
- The PR-T net gives Package body BROADCASTING.

→ NEXT SLIDES



Implementation Of The Abstract Data Type BROADCASTING

16

The Ada task type MESSAGE_CARRIER specification (private part of Package BROADCASTING) is the following :

```
task type MESSAGE_CARRIER is
  entry BEGIN_RECEIVE;
    — called by procedure RECEIVE to start waiting a message.
  entry END_RECEIVE (M : out MESSAGE);
    — called by procedure RECEIVE after the call to BEGIN_RECEIVE
    — in order to wait and receive the message into M.
  entry BROADCAST (M : MESSAGE);
    — called by procedure BROADCAST to broadcast the message M and
    — wake up the waiting tasks.
  entry RESET;
    — called by procedure RESET in order to unload the MESSAGE_CARRIER
    — which consequently is empty.
end MESSAGE_CARRIER;
```

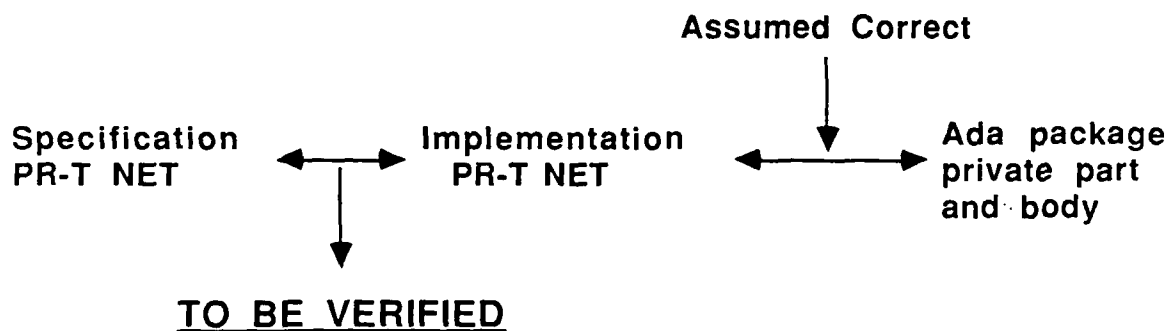
Package body BROADCASTING is the following :

```
Package body BROADCASTING is
  task body MESSAGE_CARRIER is
    MESSAGE_PRESENT : BOOLEAN := FALSE;
    MESS : MESSAGE;
    NUMBER_OF_WAITING_TASKS : NATURAL := 0;
  begin
    loop
      select
        when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) =>
          accept BROADCAST (M : MESSAGE) do
            MESS := M;
          end BROADCAST;
          MESSAGE_PRESENT := TRUE;
        or
          when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) => accept RESET;
          MESSAGE_PRESENT := FALSE;
        or
          when not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /=
          0)) => accept BEGIN_RECEIVE;
          NUMBER_OF_WAITING_TASKS := NUMBER_OF_WAITING_TASKS + 1;
        or
          when MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS /= 0) =>
            accept END_RECEIVE (M : out MESSAGE) do
              M := MESS;
            end END_RECEIVE;
            NUMBER_OF_WAITING_TASKS := NUMBER_OF_WAITING_TASKS-1;
        or terminate;
      end select;
    end loop;
  end MESSAGE_CARRIER;
  procedure BROADCAST (MESS : MESSAGE; CARRIER : in out
    MESSAGE_CARRIER) is
  begin
    CARRIER.BROADCAST (MESS);
  end BROADCAST;
  procedure RESET (CARRIER : in out MESSAGE_CARRIER) is
  begin
    CARRIER.RESET;
  end RESET;
  procedure RECEIVE (MESS : out MESSAGE; CARRIER : in out
    MESSAGE_CARRIER) is
  begin
    CARRIER.BEGIN_RECEIVE;
    CARRIER.END_RECEIVE (MESS);
  end RECEIVE;
end BROADCASTING;
```


VERIFICATION OF PACKAGE BROADCASTING

What is to be verified

Correct mapping of the Implementation PR-T net to the ADA package can be checked using the (BOND83) PR-T net grammar for ADA tasking.



Property a: The Four Operations TB, TRS, TR1, TR2 Are Indivisible

TB $\longrightarrow$ TB1, TC2, (TB2//TC3)

TRS $\longrightarrow$ TR21, TC4, (TRS2//TC5)

TR1 $\longrightarrow$ TR11, TC6, (TR12//TC7)

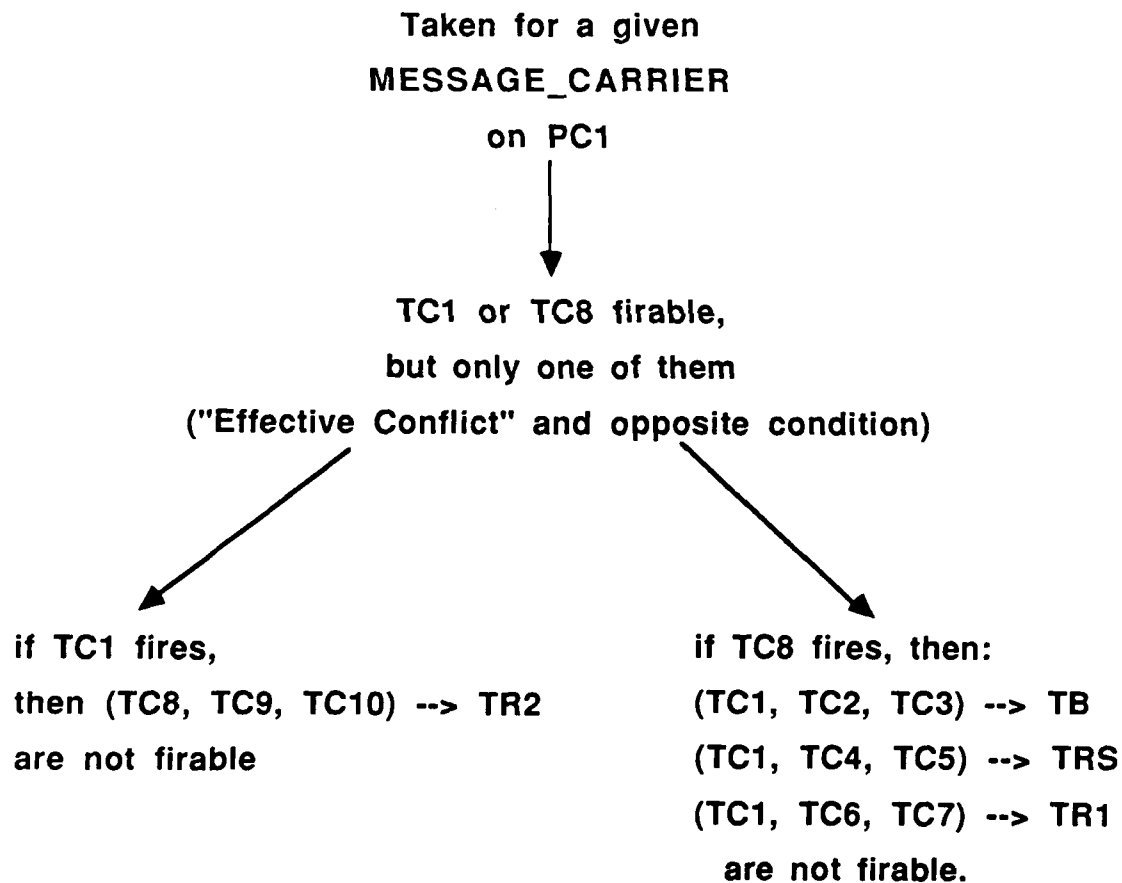
TR2 $\longrightarrow$ TR21, TC9, (TR22/TC10)

**Indivisibility after firing the 1st transition
due to the fact that:**

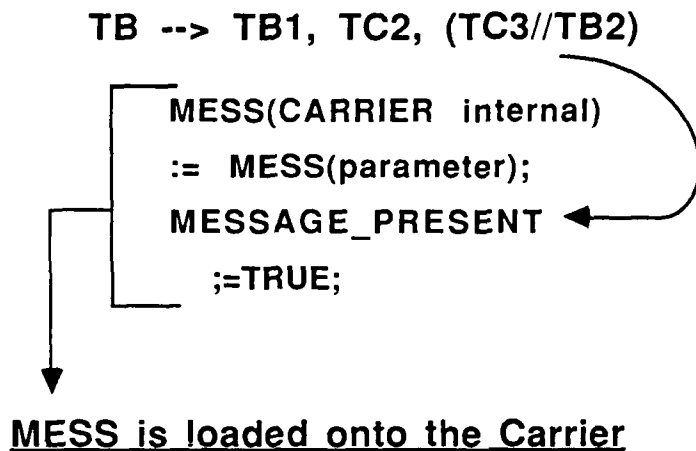
(TC2, TC3) excl (TC4, TC5) excl (TC6, TC7) excl (TC9, TC10)

**Because one token only is available for these on a given
MESSAGE_CARRIER.**

**Property b: If TR2 is Firable, then None of the
Transitions TB, TRS, TR1 is Firable**



**Property C: Procedure BROADCAST Loads, with
no Unbounded Wait, the Message MESS
onto the Carrier**



No Unbounded Wait:

- The only possible wait in (TB1, TC2, (TC3//TB2)) is:
Wait a token on PC2 to fire TC2.

- TC12 or TC13 cannot fire while the Carrier is still visible
- The only waiting places on the MESSAGE_CARRIER state machine are PC2 and PC3.

→ We must prove that TC9 is eventually firable
when a Carrier token is on PC3
(no unbounded wait on PC3).

Proof that TC9 is eventually firable when a Carrier token is on PC3:

The sketch of this proof is the following:

-- Number of tokens that eventually accumulate on PR6
= NUMBER_OF_WAITING_TASKS

-- While not (MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS/=0))
the Carrier token cannot reach PC3 (TC8 not firable)

-- If MESSAGE_PRESENT and (NUMBER_OF_WAITING_TASKS/=0)
then PC3 is eventually reached by the Carrier token
and we eventually have:
Number of (TC8, TC9, TC10, TC11) consecutive sequences
= Number of tokens on PR6
= NUMBER_OF_WAITING_TASKS;

-- As TC10 decrements NUMBER_OF_WAITING_TASKS,
this variable is 0 at the completion of these
NUMBER_OF_WAITING_TASKS sequences;
we then have:
TC8 not firable --> PC3 not reachable
TC1 firable

→ TC9 is eventually firable when a carrier token is on PC3

**Property d: Procedure RESET Unloads, with no
Unloaded Wait, the Message which was
Possibly Present on the Carrier**

-- TRS --> TRS1, TC4, (TC5//TRS2)



MESSAGE_PRESENT:=FALSE;



Procedure RESET unloads the the message

-- No unloaded wait:

The proof is similar to that for Procedure BROADCAST

Property 3: Procedure RECEIVE: If a Message was Present, then the Value of this Message is Assigned, with no Unloaded Wait, to the Parameter MESS, else the Caller Waits until a Message is Broadcast

- If Callers of RECEIVE were waiting when the message is broadcast
==> the callers are woken up (cf. proof for property c).
- If no caller of RECEIVE is waiting when the message is broadcast
==> NUMBER_OF_WAITING_TASKS=0

2 cases are then possible when RECEIVE starts (TR11 is firing):

- The Carrier token was on PC2

-- If MESSAGE_PRESENT

=> TC6, ((TC7, TC11, TC8)//(TR12, TR21)), TC9, (TC10, TC11)//TR22)

NUMBER_OF_WAITING_TASKS

= 1

The message is passed
to the RECEIVE caller

If not MESSAGE_PRESENT

TC6, ((TC7, TC11, TC1)//TR12, TR21

NUMBER_OF_WAITING_TASKS =1

The RECEIVE Caller waits on PR3

- The Carrier token was not on PC2
=> it will eventually return on PC2 because:

NUMBER_OF_WAITING_TASKS=0 => TC8 NOT FIRABLE

DERIVING A TEST SET FROM THE SPECIFICATION

- THE TEST SET SHOULD CHECK THAT EACH OF THE
PROPERTIES A,B,C,D,E DEFINED ON THE SPECIFICATION
HOLDS

- SIMPLEST WAY TO BUILD SUCH A TEST SET:
 - BUILD A TEST PROCEDURE THAT:
 - Instantiates a package on generic package BROADCASTING
 - Declares an object M_C: MESSAGE_CARRIER;
 - Creates several tasks that concurrently use M_C
in a manner which aims at testing the desired
properties on M_C.

 - TRANSLATE THE PROPERTIES A,B,C,D,E INTO CONDITIONS
THAT MUST HOLD ON M_C INTERNAL STATE AND ON THE
STATE OF THE TASKS WHICH USE M_C.

DEFINE DEBUGGER ACTONS TO OBSERVE THESE
CONDITIONS AND INSERT COMMENTS IN THE TEST
PROCEDURE EXPLAINING THESE ACTIONS AT THE
PLACE THEY MUST BE PERFORMED.

 - EXECUTE THE TEST PROCEDURE UNDER CONTROL OF
THE DEBUGGER AND PERFORM THE OBSERVATIONS
THROUGH THE DEBUGGER.

NOTES:

- WE ACTUALLY BUILT SUCH A TEST PROCEDURE
"BROADCASTING_TEST" AND THE TESTED
PACKAGE REVEALED NO BUG.**
- A "DEBUGGER" SHOULD, AS IN "BROADCASTING_TEST",
BE USED TO PRESERVE THE BEHAVIOUR OF ALREADY
VERIFIED, AND THEREFORE A PRIORI CORRECT,
PIECES OF SOFTWARE RATHER THAN TO
"DEBUG".**

CONCLUSIONS

MAIN ADVANTAGES OF OUR V AND V TECHNIQUE:

- THE SAME MODELLING TECHNIQUE (PR-T NETS) IS KEPT FROM SPECIFICATION TO IMPLEMENTATION
----> IT IS EASIER TO PROVE THAT THE IMPLEMENTATION IS CORRECT WITH RESPECT TO THE SPECIFICATION
- A TEST SET CAN BE DERIVED STRAIGHTFORWARDLY FROM THE SPECIFICATION.
- AS A CONSEQUENCE OF PRIOR VERIFICATION, THE COST OF TESTING SHOULD DRAMATICALLY DROP DOWN (ONLY MINOR BUGS SHOULD REMAIN)

PROGRESS TO REACH AN OPERATIONAL STATUS IN 1990:

- REPLACE THE DEDUCTIVE STYLE OF PROOF THAT WAS
PRESENTED BY A MORE MECHANIZABLE ONE, USING THE
LINEAR ALGEBRA REPRESENTATION OF NETS.
(RESEARCH OFFERS REASONABLE HOPE)

- IMPLEMENT A GENERAL PETRI NET ENVIRONMENT
OPEN TO THE IMPLEMENTATION OF NEW NET CATEGORIES
AND NEW PROOF ALGORITHMS (A BIG DEAL!)

- VERIFICATION OF THE IMPLEMENTATION AGAINST THE
SPECIFICATION IS FINE, BUT VERIFICATION OF THE
PROPERTIES OF THE SPECIFICATION SHOULD BE CONSIDERED TOO
(NOT A BIG PROBLEM: SHOULD BE A BY-PRODUCT OF THE
REMAINING)

- TRY AND MECHANIZE THE IMPLEMENTATION PROCESS
(A LONG TERM EFFORT, PROBABLY NOT FOR 1990!)

An Empirical Study of Testing Concurrent Ada Programs

Kuo-Chung Tai^a and Richard H. Carver^a

Department of Computer Science
Box 8206
North Carolina State University
Raleigh, NC 27695-8206

Evelyn E. Obaid

Department of Mathematics & Computer Science
San Jose State University
San Jose, CA 95192

* Ada is a registered trademark of the US Government - Ada Joint Program Office

2 This research was supported in part by the Integrated Manufacturing Systems Engineering Institute at North Carolina State University

Abstract

Repeated executions of a concurrent Ada program P with input X may exercise different sequences of rendezvous and thus produce different results. Therefore, the correctness of P with input X cannot be determined by one or more executions of P with input X . Recently, we have studied an approach, called deterministic execution, to testing concurrent Ada programs. This approach uses test cases of the form (X, S) , where X is an input and S is a rendezvous sequence. Such a test case is called an IN_R test case.

For a given IN_R test case (X, S) for P , the deterministic execution approach determines whether or not S can be produced by an execution of P with input X and, if so, actually reproduces S . Our strategy is to transform P into another concurrent Ada program P' such that any execution of P' with (X, S) as input reproduces S if and only if S can be produced by an execution of P with input X .

In this paper, we describe how to apply the deterministic execution approach to test several concurrent Ada programs. Through the discussion of testing these programs, we address the problems encountered and illustrate how to solve these problems. The results of our empirical study indicate that the deterministic execution approach is very effective for testing concurrent Ada programs.

1. Introduction

The high cost of software testing has stimulated substantial research efforts toward the development of testing techniques and tools. However, the area of testing concurrent programs has received little attention [Bri73,78]. The conventional approach to testing a program is to select a set of test inputs and execute this program with each test input exactly once. This approach, however, is inappropriate for testing concurrent programs.

Let P be a concurrent Ada program. An execution of P with some input involves a sequence of rendezvous, called a rendezvous sequence, which determines the result of this execution. Repeated executions of P with the same input, however, may exercise different rendezvous sequences and thus produce different results. This can happen due to two reasons. One is that the relative progress of concurrent tasks in P is unpredictable. The other is that a select statement in P allows a nondeterministic selection over several possible alternatives.

Because of the "unpredictable rendezvous sequence" property, testing concurrent Ada programs has three critical issues which do not exist in testing sequential Ada programs. The first issue is that the correctness of P with input X cannot be determined by a single execution of P with input X . Repeated executions of P with input X may increase the chance of detecting errors in P . However, even if P with input X has been executed successfully many times, a future execution of P with input X may still produce an unexpected result.

The second issue is the reproduction of the rendezvous sequence (or the result) of an execution of P with input X . Assume that an execution of P with input X has produced a rendezvous sequence S with an unexpected result. (S is said to be feasible, i.e., it is allowed by the implementation of P with input X). In order to understand how the unexpected result was produced, we would like to reproduce S and collect certain information. However, there is no guarantee that S can be reproduced by repeated executions of P with input X . The problem of reproducing a feasible rendezvous sequence is referred to as the reproducible testing problem for Ada. (In a reference to the reproduction of a feasible rendezvous sequence, the word "feasible" is often omitted, since it is impossible to reproduce an infeasible rendezvous sequence.)

The third issue is the determination of whether or not a given rendezvous sequence is feasible. Assume that an execution of P with input X has produced an invalid rendezvous sequence S' . (A rendezvous sequence of P with input X is said to be invalid if it is not allowed by the specification of P with input X). After making corrections to P , we would like to know whether or not S' is still feasible, i.e., whether or not S' can still be produced by the modified P with input X . However, even if many executions

of the modified P with input X have not produced S' , we cannot say that S' is infeasible for the modified P with input X . The problem of determining whether or not a rendezvous sequence is feasible is referred to as the feasibility problem.

To determine the correctness of a concurrent program, two different approaches can be applied. One approach, called "repeated execution", is to repeatedly execute a concurrent program with the same input. The other approach, called "deterministic execution", is to reproduce sequences of synchronizations of a concurrent program. The deterministic execution approach is usually applied to assembly programs and involves the use of low-level interfaces with the underlying operating system. Therefore, this approach has been considered tedious and ad hoc. In [Tai85], it was suggested that the deterministic execution approach could be applied to concurrent programs written in a high-level language and could be done so without having interfaces with the underlying operating system.

To test a concurrent Ada program P , the deterministic execution approach uses test cases of the form (X,S) , where X is an input and S is a rendezvous sequence. Such a test case is called an IN_R test case. In order to test a concurrent Ada program with IN_R test cases, we need to solve both the reproducible testing and the feasibility problems for Ada. To solve the reproducible testing problem for Ada, our strategy is to transform a concurrent Ada program P into another concurrent Ada program P' such that the reproduction of a rendezvous sequence S of P with input X requires exactly one execution of P' with (X,S) as input. In other words, if S is feasible for P with input X , then an execution of P' with input (X,S) definitely reproduces S . Notice that this strategy also solves the feasibility problem for Ada provided that P' is constructed so that if S is infeasible, then an execution of P' with input (X,S) can never reproduce S .

Recently we have developed three reproducible testing methods, called R_PERMIT, BAR_PERMIT, and BARG_PERMIT respectively, for three different classes of concurrent Ada programs [Tai86a,b,c]. In this paper, we apply these reproducible testing methods to test several concurrent Ada programs. Through the discussion of testing these programs, we address the problems encountered in reproducible testing and illustrate how to solve these problems. Also, we show the effectiveness of the deterministic execution approach for error detection.

This paper is organized as follows. In section 2, the notion of a rendezvous sequence is formalized and a type of error in concurrent Ada programs, called a rendezvous error, is defined. Section 3 shows how to apply the R_PERMIT method to test concurrent Ada programs. Section 4 presents applications of the BAR_PERMIT method. Section 5 demonstrates the BARG_PERMIT method, which is a complete solution to the reproducible testing problem for Ada. Finally, section 6 concludes this paper. Throughout this paper, it is assumed that the result of an execution of a

concurrent Ada program depends solely on the input and the rendezvous sequence of this execution. Also, it is assumed that a concurrent Ada program executes on a single processor and does not use interrupts or the time clock.

2. Rendezvous Sequences and Errors of Concurrent Ada Programs

In section 2.1, we provide a formal definition of a rendezvous sequence such that the result of an execution of a concurrent Ada program depends solely on the input and the R-sequence of this execution. Based on the notion of R-sequences, a type of error in a concurrent Ada program, called a rendezvous error, is defined in section 2.2.

2.1 Rendezvous Sequences

An execution of a concurrent Ada program involves a sequence of starts and finishes of rendezvous. A rendezvous sequence (R-sequence) is defined as

((C₁,E₁), (C₂,E₂), ...) (1)

where, C_i (E_i), i>0, denotes the caller (the entry name) of the ith rendezvous to be started. (An entry name includes the name of the task owning this entry). An R-sequence may possibly be followed by a termination indicator denoted by "\$". An R-sequence without an ending "\$" implies that the R-sequence is infinite or it results in a deadlock.

Consider a concurrent Ada program P with input X. Due to the "unpredictable rendezvous sequence" property, both the specification and implementation of P may allow different R-sequences during repeated executions of P with input X. An R-sequence is said to be valid (feasible) for P with input X if it is allowed by P's specification (implementation); otherwise it is said to be invalid (infeasible).

Consider the concurrent Ada program in Fig. 1, called BOUNDED_BUFFER, for solving the bounded buffer problem with the buffer size being two. BOUNDED_BUFFER contains three tasks and does not require any input. Task BUFFER_CONTROL provides two entries DEPOSIT and WITHDRAW and synchronizes the execution of DEPOSIT and WITHDRAW operations on the buffer such that the items in the buffer are withdrawn in the order of deposit. Task PRODUCER (CONSUMER) contains a sequence of three calls to entry DEPOSIT (WITHDRAW). Let P (C) denote PRODUCER (CONSUMER) and D (W) denote DEPOSIT (WITHDRAW). According to the specification of BOUNDED_BUFFER, ((P,D), (P,D), (C,W), (C,W), (P,D), (C,W), \$), denoted by S', is a valid R-sequence, while ((P,D), (P,D), (P,D), (C,W), (C,W), (C,W), \$), denoted by S'', is invalid. According to the implementation of BOUNDED_BUFFER, S' is feasible and S'' is infeasible.

Assume that an execution of BOUNDED_BUFFER has produced the first two rendezvous in S'. Notice that the entry call statement in

PRODUCER producing the fifth rendezvous can be uniquely determined from PRODUCER and the previous rendezvous. Thus, DEPOSIT (the entry name of the fifth rendezvous) is also uniquely determined. In other words, for a feasible R-sequence based on the definition in (1), (C_i, E_i) can be replaced with C_i . Thus, a feasible R-sequence can be represented as

$(C_1, C_2, \dots)$ (2)

where C_i , $i > 0$, denotes the caller of the i th rendezvous to be started. For the sake of simplicity, in later discussions, an R-sequence, whether it is feasible or not, is represented using the definition in (2).

2.2 Rendezvous Errors

Let $VAL(P, X)$ be the set of valid R-sequences of P with input X and $FEA(P, X)$ be the set of feasible R-sequences of P with input X . P is said to have a rendezvous error if for some input X , $VAL(P, X)$ and $FEA(P, X)$ are not the same. Thus, a rendezvous error implies the existence of a valid, but infeasible R-sequence or a feasible, but invalid R-sequence.

A valid, but infeasible R-sequence of P with input X is an R-sequence which is allowed by P 's specification, but can never be produced by any execution of P with input X . The existence of such an R-sequence does not necessarily cause an execution of P with input X to produce an unexpected result. But such an R-sequence indicates a discrepancy between the specification and the implementation of P .

A feasible, but invalid R-sequence of P with input X is an R-sequence which is not allowed by P 's specification, but can be produced by some execution of P with input X . If such an R-sequence is produced by an execution of P with input X , the result of this execution is most likely incorrect. Notice that if the implementation of P allows an R-sequence resulting in a deadlock, then this R-sequence is feasible, but invalid. Thus, a deadlock error is considered to be a rendezvous error.

Errors in P can be classified into two categories: rendezvous errors and computation errors. P is said to have a computation error if an execution of P with input X produces a valid R-sequence, but an incorrect result. The deterministic execution approach can be applied to detect both types of errors. Based on the specification of P , we can select both valid and invalid R-sequences of P with input X . Let S be such an R-sequence. If S is valid (invalid), then it should be feasible (infeasible); otherwise, P has a rendezvous error. The feasibility of S for P with input X can be determined by exactly one execution of P' with (X, S) as input, where P' is P transformed according to a reproducible testing method. If this execution produces S as the resulting R-sequence, then S is feasible; otherwise, if this execution produces a different R-sequence, then S is infeasible. If S is both valid and feasible, we need to determine the correctness of the result, since P may have a computation error.

3. Applications of the R_PERMIT Method

The reproducible testing problem for Ada is to reproduce feasible R-sequences. Consider again program BOUNDED_BUFFER in Fig. 1. As discussed in section 2.1, an R-sequence of BOUNDED_BUFFER can be described as

(C1,C2,...)

where C_i, i>0, is the name of the calling task (PRODUCER or CONSUMER) of the ith rendezvous to be started.

The basic idea of the R_PERMIT method is to transform BOUNDED_BUFFER in order to control the execution sequence of entry calls such that an entry call in task PRODUCER (CONSUMER) is allowed to be issued only after the previous rendezvous has started and when PRODUCER (CONSUMER) is the caller for the next rendezvous. A clearer explanation of the R_PERMIT method is to consider that BOUNDED_BUFFER owns a permit for rendezvous, called R_PERMIT. Before issuing an entry call for a rendezvous, tasks PRODUCER and CONSUMER must make a request for R_PERMIT and wait until it receives this permit. After starting a rendezvous, the PRODUCER or CONSUMER task executing the rendezvous must release R_PERMIT immediately.

Let CONTROL be a task in package R_CONTROL which controls the reproduction of a given R-sequence. Task CONTROL contains an entry family called REQUEST_R_PERMIT and an entry called RELEASE_R_PERMIT. Fig. 2 shows the program BOUNDED_BUFFER transformed for reproducible testing. Statements in the program which were inserted or modified for reproducible testing are indicated by comments beginning with "--#". In tasks PRODUCER and CONSUMER each entry call statement is replaced with

```
CONTROL.REQUEST_R_PERMIT(j);  
entry call statement;
```

where j is a unique number assigned to task PRODUCER (j=1) and CONSUMER (j=2). In task BUFFER_CONTROL, each accept statement

```
accept E(...) do  
  ...;  
end E;
```

is transformed into

```
accept E(...) do  
  CONTROL.RELEASE_R_PERMIT;  
  ...;  
end E;
```

Except for the first rendezvous, task CONTROL accepts a call to entry REQUEST_R_PERMIT(j) only after the RELEASE_R_PERMIT call for the previous rendezvous has been accepted and when task j is the calling task for the next rendezvous. After the acceptance of a call to entry REQUEST_R_PERMIT(j), task CONTROL is waiting to accept a call to entry RELEASE_R_PERMIT, which must come from the task executing the rendezvous initiated by task j. An implementation of task CONTROL was given in [Tai86a] and is omitted here.

To illustrate the effectiveness of the deterministic execution approach for detecting errors, we need to consider incorrect versions of program BOUNDED_BUFFER. Also, we will consider IN_R test cases for error detection. Since BOUNDED_BUFFER does not require input, we only consider the selection of valid R-sequences as well as invalid R-sequences.

Error (3.1): Assume that in task BUFFER_CONTROL, the variable COUNT is incorrectly initialized to one. Due to this error, the valid R-sequence (P,P,C,C,P,C,\$) becomes infeasible. The reproducible testing of the incorrect BOUNDED_BUFFER with (P,P,C,C,P,C,\$) produces a deadlock. This error also cause the invalid R-sequence (P,C,C,P,P,C,\$) to become feasible. As a result, the reproducible testing of the incorrect BOUNDED_BUFFER with (P,C,C,P,P,C,\$) does not produce an expected deadlock.

Error (3.2): Assume that in task BUFFER_CONTROL, the condition "COUNT<SIZE" is incorrectly written as "COUNT<=SIZE". Due to this error, the invalid R-sequence (P,P,P,C,C,C,\$) becomes feasible. Thus, the reproducible testing of the incorrect BOUNDED_BUFFER with (P,P,P,C,C,C,\$) does not produce an expected deadlock.

4. Applications of the BAR_PERMIT method

The R_PERMIT method can be applied to a concurrent Ada program only if certain features OF Ada are not used in this program. In this section we apply the BAR_PERMIT method which allows some of the features not allowed by R_PERMIT. Section 4.1 demonstrates how to test concurrent Ada programs which access the COUNT attributes of entries. Section 4.2 discusses the testing of concurrent Ada programs which contain conditional/timed entry calls and selective wait statements with delay alternatives or else parts.

4.1 COUNT Attributes of Entries

Fig. 3 shows an Ada package called RESOURCE which solves the concurrent readers and writers problem. The strategy used in package RESOURCE is called "many readers or one writer with the writers having a higher priority, but all waiting readers are given access after a writer has finished" [Geh84,p.65]. Notice that in the body of task RW, the COUNT attributes of entries START_READ and START_WRITE are accessed. Therefore, the reproduction of a feasible R-sequence of package RESOURCE may require zero, one, or more arrivals at entries START_READ and START_WRITE. However, the R_PERMIT method suffers from the restriction that exactly one entry call arrival is allowed between two consecutive rendezvous.

To reproduce a feasible R-sequence of package RESOURCE, we need to control accesses to the COUNT attributes of entries and to control the sequence of arrivals and rendezvous of entry calls. In order to do so, we consider each entries COUNT attribute as a

pseudo-entry and each access of a COUNT attribute as a pseudo rendezvous. A sequence of arrivals and rendezvous (including both real- and pseudo-rendezvous) of entry calls is called an arrival-rendezvous sequence (AR-sequence). To reproduce a feasible R-sequence S of package RESOURCE, we need to reproduce a feasible AR-sequence with S as its R-sequence.

Fig. 4 shows a graphical description of a valid AR-sequence, called RW_SEQ, of package RESOURCE with two writer tasks (W1 and W2) and three reader tasks (R1, R2, and R3). In RW_SEQ, each reader (writer) calls procedure READ (WRITE) exactly once and thus issues two entry calls. (For a reader (writer), the first call is to entry START_READ (START_WRITE) and the second call to entry END_READ (END_WRITE).) Arrivals of entry calls are represented by the names of readers and writers. These names are listed in columns based on the names of entries called by these entry calls. The value in each pair of parentheses immediately following the name of a reader or writer indicates the number of the rendezvous accepting the entry call. For example, the third arrival in RW_SEQ is denoted by "R3 (4)", indicating that the caller is R3 and this call will be accepted at the fourth rendezvous. Each rendezvous is represented by a line consisting of a sequence of "*"s. The line representing the i th rendezvous, $0 < i \leq 12$, is indicated by number i at the left end. The right end of each line representing a rendezvous indicates the corresponding statement executing the rendezvous. (Such a statement is called a rendezvous statement.)

The AR-sequence RW_SEQ is called a batch AR-sequence (BAR-sequence) in [Tai86b,c]. An AR-sequence is said to be a BAR-sequence provided that:

- (1) For each rendezvous at a select statement, it is the only possible rendezvous and
- (2) if the i th rendezvous is executed by task T, then all entry calls between the $(i-1)$ th and the i th rendezvous are to task T.

Our strategy for reproducing the BAR-sequence RW_SEQ is to transform package RESOURCE to make it possible to control the sequence of arrivals and rendezvous of entry calls during the program execution.

The basic idea of the transformation is to precede each entry call or rendezvous statement with one or more entry calls to a task called CONTROL in package BAR_CONTROL. (The specification and implementation of package BAR_CONTROL were given in [Tai86b,c].) As a result of this transformation, each entry call (rendezvous) statement is allowed to be executed only when this statement is to issue (execute) the next arrival (rendezvous) in a given AR-sequence. Similar to the R_PERMIT method described in section 3, we consider that package RESOURCE owns a permit for arrival, called A_PERMIT, and a permit for rendezvous, called R_PERMIT. Each entry call statement is preceded with a request for A_PERMIT. Each rendezvous statement is preceded with a code segment which requests R_PERMIT and waits for and releases

A_PERMIT for each required arrival.

Fig. 5 shows package RESOURCE transformed for reproducible testing. Notice that the declarations of procedures READ and WRITE in package RESOURCE are modified to include one additional parameter which passes the task number of a reader or writer. Each of the procedures READ and WRITE has two entry call statements. Task RW has three rendezvous statements. The first is the only select statement. The second is the assignment statement accessing the COUNT attribute of entry START_READ. (This assignment statement is in the last alternative of the select statement.) The third is the accept statement inside the loop immediately following the second rendezvous statement. These three statements are preceded by the code WAIT_FOR_RENDEZVOUS which is indicated by the comment "--###".

In procedures READ and WRITE, each entry call statement is preceded with a request for A_PERMIT as shown below:

```
CONTROL.REQUEST_A_PERMIT(ID);
```

```
entry call statement;
```

where ID is the task identification assigned to each task.

The execution of a rendezvous statement is controlled by delaying its execution until the required entry calls have arrived. This is accomplished by prefacing each rendezvous statement in task RW with the code in Figure 6, called WAIT_FOR_RENDEZVOUS. The call to CONTROL.REQUEST_R_PERMIT(ID) returns when a rendezvous statement in task RW is to produce the next rendezvous. This call returns with a value called TOTAL_ARRIVALS which is the total number of entry call arrivals required (excluding the failed conditional/timed entry calls) before the occurrence of the next rendezvous. (Since a BAR-sequence is used, all these entry calls are to task ID.) Each call to CONTROL.WAIT_ARRIVAL returns after an entry call to task RW has been allowed to be issued. The purpose of the internal while loop is to wait until this entry call has arrived. (The interval between the issuing and the arrival of an entry call is unpredictable.) The call to CONTROL.RELEASE_A_PERMIT notifies task CONTROL of the arrival of this entry call to task RW.

The completion of code WAIT_FOR_RENDEZVOUS occurs after all the required entry calls to task j have arrived. Then the rendezvous statement in task RW immediately following the completed WAIT_FOR_RENDEZVOUS begins execution. After a rendezvous starts, task RW should notify task CONTROL of the release of R_PERMIT. In task RW, each accept statement of the form

```
accept E(...);
```

is transformed into

```
accept E(...);
```

```
CONTROL.RELEASE_R_PERMIT;
```

Now we discuss the situation that an access to an entry's COUNT attribute occurs outside an assignment statement or the "when" conditions of a select statement. Consider the following

statement in task RW in Fig. 4.

```
For L in 1..START_READ'COUNT loop ...; end;
```

It is impossible to notify task CONTROL of the finish of accessing START_READ'COUNT after the evaluation of START_READ'COUNT and before the beginning of the for loop. However, this problem can be solved by transforming the above for loop statement into

```
TEMP := START_READ'COUNT;
```

```
CONTROL.RELEASE_R_PERMIT;
```

```
for I in 1..TEMP loop ...; end;
```

as shown in Fig. 5(b). The same transformation can be applied to any non-assignment statement which references a COUNT attribute.

To test package RESOURCE, we used a procedure which contains two writers and three readers and the BAR-sequence given in Fig. 4. Below we consider two possible errors in package RESOURCE.

Error (4.1): Assume that in task RW, in the when condition of the third alternative of the select statement, "and" is incorrectly written as "or". During an execution of the incorrect RESOURCE with RW_SEQ, when the second rendezvous is to be selected, both the third and fourth alternatives of the select statement are live (i.e., they are open and have entry calls to accept). (The fourth alternative is expected to be the only live one.) If the third alternative is chosen, then this execution results in a deadlock, because the next rendezvous in RW_SEQ can never be reproduced. If the fourth alternative is selected, then this error is undetected. (Note that the BAR_PERMIT method cannot guarantee to correctly determine the feasibility of a BAR-sequence.)

Error (4.2): Assume that in task RW, in the when condition of the first alternative of the select statement, the phrase "and START_WRITE'COUNT=0" is incorrectly missing. During an execution of the incorrect RESOURCE with RW_SEQ, when the sixth rendezvous is to be selected, both the first and third alternatives are live. If the first alternative is chosen, then this execution results in a deadlock, because the entry call from W1 to END_WRITE can never be issued.

4.2 Conditional/Timed Entry Calls and Selective Wait Statements with Delay Alternatives or Else Parts

The procedure in Fig. 7 solves another version of the bounded buffer problem. This procedure, called BOUNDED_BUFFER_NEW, differs from Fig. 1 in that task PRODUCER makes conditional entry calls to task BUFFER_CONTROL and a delay alternative is added to the selective wait statement in task BUFFER_CONTROL. The exit statements in tasks PRODUCER, CONSUMER, and BUFFER_CONTROL are used to provide a normal termination without the use of a terminate alternative in the select statement in BUFFER_CONTROL.

To reproduce an R-sequence of BOUNDED_BUFFER_NEW, we need to control the selection of the delay alternative in the select statement of task BUFFER_CONTROL and we need to control the

success or failure of the conditional entry call statement in task PRODUCER. In order to control the selections of delay alternatives we consider a delay alternative as a pseudo-entry and the execution of a delay alternative as a pseudo rendezvous (Similarly for the else part of a select statement).

In order to control the success or failure of a conditional entry call, we have to indicate in the AR-sequence whether the call succeeds or not (Similarly for a timed entry call). Fig. 8 shows a feasible BAR-sequence, called BB_SEQ, of procedure BOUNDED_BUFFER_NEW. The structure of BB_SEQ is similar to that in Fig. 4. In BB_SEQ, task PRODUCER (P) issues three conditional entry calls of which only two succeed. The second conditional entry call by task PRODUCER fails, as indicated by the letter "f" inside the pair of parentheses immediately after the second "P". Task CONSUMER (C) issues two entry calls. The fifth rendezvous in BB_SEQ is an execution of the delay alternative in the select statement of task BUFFER_CONTROL. To reproduce BB_SEQ, program BOUNDED_BUFFER_NEW has been transformed according to the BAR_PERMIT method as shown in Fig. 9.

The conditional entry call statement in task PRODUCER

```
select
  DEPOSIT(C);
else
  null;
end select;
```

was transformed into

```
CONTROL.SELECT_REQUEST_A_PERMIT(ID)(STATUS);
if STATUS then
  DEPOSIT(C);
else null;
end if;
```

The call to SELECT_REQUEST_A_PERMIT(ID) returns STATUS, which indicates whether or not the call should succeed according to the given BAR-sequence. A timed entry call statement is transformed in the same way.

The delay alternative in task BUFFER_CONTROL

```
delay 3.0;
exit;
```

was transformed into

```
delay 3.0;
CONTROL.RELEASE_R_PERMIT;
exit;
```

An else part is transformed similarly.

Possible errors in BOUNDED_BUFFER_NEW as described in (3.1) and (3.2) can be detected by using BAR-sequences corresponding to the R-sequences for detecting errors in (3.1) and (3.2).

5. Applications of The BARG_PERMIT method

In Fig. 10, task BUFFER_CONTROL of program BOUNDED_BUFFER_NEW has been modified to include a fourth alternative in the select statement. (The modified BOUNDED_BUFFER_NEW is called BOUNDED_BUFFER_IN.) The guarding condition in this alternative is true only if both the PRODUCER and CONSUMER tasks have issued an entry call and both tasks are waiting at the select statement. Notice that if this condition is true, then one or both of the conditions for accepting a DEPOSIT entry call or a WITHDRAW entry call must also be true. An R-sequence of BOUNDED_BUFFER_IN in which it is required to have two or more possible rendezvous is called an "inherently nondeterministic R-sequence" [Tai86c]. (A similar problem may arise if a select statement contains two or more accept statements for the same entry or if a select statement contains two or more delay alternatives with the same delay value.) In order to reproduce an R-sequence of BOUNDED_BUFFER_IN, we must extend the definition of a rendezvous to include the index number of the alternative of the select statement selected for a real- or pseudo-rendezvous.

As shown below, all the alternatives of a select statement are assigned unique index numbers (starting with one) according to their order of appearance:

```
select
    ...; -- the 1st alternative
or ...; -- the 2nd alternative
...
or ...; -- the nth alternative
end select;
```

Fig. 11 shows a feasible BAR-sequence called INDEX_SEQ of procedure BOUNDED_BUFFER_IN. The structure of INDEX_SEQ is similar to that in Fig. 8 except that the right most end of each line representing a rendezvous indicates the index number of the alternative selected. Notice that the third alternative was selected for the second rendezvous in INDEX_SEQ, although a rendezvous was possible for both the first and the third alternatives.

Fig. 12 shows the modified task BUFFER_CONTROL transformed for reproducing INDEX_SEQ using the BARG_PERMIT method. This transformation is the same as the BAR_PERMIT method except for the following modifications.

- (1) In task BUFFER_CONTROL, the following declarations have been inserted.

```
MAX_SELECT_INDEX : constant INTEGER := 4;
-- the maximum number of accept and delay alternatives in
-- the select statement.
GUARD : array(0..MAX_SELECT_INDEX) of BOOLEAN;
```
- (2) The first statement in code WAIT_FOR_RENDEZVOUS CONTROL.REQUEST_R_PERMIT(TOTAL_ARRIVALS); is replaced with


```

CONTROL.REQUEST_R_PERMIT(TOTAL_ARRIVAL,R_INDEX);
GUARD := (others => FALSE);
GUARD(R_INDEX) := TRUE;

```

The additional parameter R_INDEX in entry REQUEST_R_PERMIT of task CONTROL returns the index of the alternative to be selected for the next rendezvous.

- (3) For the select statement, if the kth, $k > 0$, alternative has a when condition, i.e., it is of the form


```

when ... => selective_wait_alternative

```

 then it is transformed into


```

when ... and GUARD(k) => selective_wait_alternative

```

 If the kth alternative does not have a when condition, i.e., it is of the form


```

selective_wait_alternative

```

 then it is transformed into


```

when GUARD(k) => selective_wait_alternative

```
- (4) Package BAR_CONTROL is modified based on the new definition of a rendezvous and the new definition of entry REQUEST_R_PERMIT. Also, the name of package BAR_CONTROL is changed to BARG_CONTROL.

The above modifications to the BAR_PERMIT method guarantee that when a select statement is executed, if an accept or delay alternative is to be selected for a rendezvous, there is exactly one open alternative.

6. Conclusion

In this paper, we have shown how to apply three reproducible testing methods to test several concurrent Ada programs. Also, we have demonstrated the effectiveness of the deterministic execution approach for error detection. More details of these reproducible testing methods and other issues in the deterministic execution approach were discussed in [Tai86b,c].

Our reproducible testing methods can be easily automated. Testing tools based on these methods are portable and can be constructed by using the front-end of an Ada compiler. Automated tools for testing concurrent Ada programs, similar to those for analyzing and debugging concurrent Ada programs [e.g., Ger84, Hel85a, Tay83] should be part of an Ada programming environment [Tay85].

One major problem in the deterministic execution approach is the selection of rendezvous sequences which are effective for error detection. The test generation techniques already developed for sequential programs [e.g., Adr82, How82, Tai80] can be applied to generate rendezvous sequences. However, generation of specification-based rendezvous sequences needs to consider various specification models and languages, including finite state machines, Petri nets, path expressions, temporal logic, event-based model, flow expressions, constrained expressions, etc. [e.g., Avr86, Che83, Das85, Hel85b, Luc85, Mil85, Ram83]. Research is underway to develop techniques for selecting rendezvous sequences.

The deterministic execution approach can be applied to test any concurrent program. However, the reproducible testing problem requires different solutions for different synchronization constructs and different concurrent languages. We have already developed solutions to the reproducible testing problem for several synchronization constructs, including semaphores and monitors [Car86]. Finally, it is important to point out that the reproducible testing problem also exists for sequential languages containing "guarded commands" or other nondeterministic constructs (e.g., [Par83]).

References

- [Adr82] Adrion, W. R., M. A. Branstad, and J. C. Cherniavsky, "Validation, verification, and testing of computer software," ACM Computing Survey, Vol. 14, No. 2, June 1982, 159-192.
- [Avr86] Avrunin, G.S., etc., "Constrained expressions: adding analysis capabilities to design methods for concurrent software systems." IEEE Tran. Soft. Eng., Vol. SE-12, No. 2, Feb. 1986, 278-292.
- [Bri73] Brinch Hansen, P., "Testing a multiprogramming system," Software-Practice and Experience, Vol. 3, 1973, 145-150.
- [Bri78] Brinch Hansen, P., "Reproducible testing of monitors," Software - Practice and Experience, Vol. 8, 1978, 721-729.
- [Car86] Carver, R., and Tai, K. C., "Reproducible testing of concurrent programs based on shared variables", to appear in Proc. 6th International Conference on Distributed Computing Systems, May 1986.
- [Che83] Chen, B.S., and Yeh, R. T., "Formal specification and verification of distributed systems," IEEE Trans. Soft. Eng., Vol. SE-9, No. 6, Nov. 1983, 710-722.
- [Das85] Dasarathy, B., "Timing constraints of real-time systems: constructs for expressing them, Methods for validating them," IEEE Trans. Soft. Eng., Vol SE-11, No. 1, Jan. 1985, 80-86.
- [Geh84] Gehani, N., Ada Concurrent Programming, Prentice-Hall, 1984.
- [Ger84] German, S. M., "Monitoring for deadlock and blocking in Ada tasking," IEEE Trans. Soft. Eng., Vol. SE-10, No. 6, Nov. 1984, 764-777.
- [Hel85a] Helmbold, D., and Luckham, D., "Debugging Ada tasking programs," IEEE Software, March 1985, 47-57.
- [Hel85b] Helmbold, D., "TSL: task specification language," Proc. Ada Inter. Conf. (ACM Ada LETTERS, Vol. V, Issue 2, Sept./Oct. 1985), 255-274.
- [How82] Howden, W. E., "Validation of scientific programs," ACM Computing Surveys, Vol. 14, No. 2, June 1982, 193-227.
- [Luc85] Luckham, D., and von Henke, F. W., "An overview of Anna, a specification language for Ada," IEEE Software, Vol. 2, No. 2, March 1985, 9-22.
- [Mil85] Milne, "CIRCAL and the representation of communication, concurrency, and time," ACM Trans. Programming Languages and Systems, Vol. 7, No. 2, April 1985, 270-298.
- [Par83] Parnas, D. L., "A generalized control structure and its formal definition," Comm. ACM, Vol. 26, No. 8, Aug. 1983, 572-581.
- [Ram83] Ramamritham, K., and Keller, R. M., "Specification of synchronizing processes," IEEE Trans. Soft. Eng., Vol. SE-9, No. 6, Nov. 1983, 722-733.
- [Tai80] Tai, K. C., "Program testing complexity and test criteria," IEEE Trans. on Software Engineering, Vol. SE-6, Nov. 1980, 531-537.
- [Tai85] Tai, K. C., "On testing concurrent programs," Proc. COMPSAC 85, Oct. 1985, 310-317.
- [Tai86a] Tai, K. C., "An approach to testing concurrent Ada programs," to appear in Proc. 1986 Washington Ada Symposium, March 1986, 253-264.

- [Tai86b] Tai, K. C., and Obaid, E. E., "Reproducible testing of Ada tasking programs", accepted for publication in Proc. IEEE 2nd International Conference on Ada Applications and Environment, April 1986, 69-80.
- [Tai86c] Tai, K.C., Obaid, E.E., and Carver, R.H., "Testing of concurrent Ada programs", Technical Report TR-86-06, Department of Computer Science, North Carolina State University, March 1986.
- [Tay83] Taylor, R.N., "A general-purpose algorithm for analyzing concurrent programs," Commu. ACM, Vol. 26, No. 5, May 1983, 362-375.
- [Tay85] Taylor, R. N., and Standish, T. A., "Steps to an advanced Ada programming environment," IEEE Trans. Soft. Eng., Vol. SE-11, No. 3, March 1985, 302-310.

```

with TEXT_IO; use TEXT_IO;
procedure BOUNDED_BUFFER is
  task PRODUCER;
  task CONSUMER;
  task BUFFER_CONTROL is
    entry DEPOSIT(C : in CHARACTER);
    entry WITHDRAW(C : out CHARACTER);
  end BUFFER_CONTROL;
  task body PRODUCER is
    begin
      DEPOSIT('A');
      DEPOSIT('B');
      DEPOSIT('C');
    end PRODUCER;
  task body CONSUMER is
    H : CHARACTER;
    begin
      WITHDRAW(H);
      PUT(H);
      WITHDRAW(H);
      PUT(H);
      WITHDRAW(H);
      PUT(H);
    end CONSUMER;
  task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
    begin
      loop
        select
          when COUNT < SIZE =>
            accept DEPOSIT(C : in CHARACTER) do
              BUFFER(IN_INDEX) := C;
            end;
            IN_INDEX := IN_INDEX mod SIZE + 1;
            COUNT := COUNT + 1;
          or when COUNT > 0 =>
            accept WITHDRAW(C : out CHARACTER) do
              C := BUFFER(OUT_INDEX);
            end;
            OUT_INDEX := OUT_INDEX mod SIZE + 1;
            COUNT := COUNT - 1;
          or terminate;
        end select;
      end loop;
    end BUFFER_CONTROL;
begin
  null;
end BOUNDED_BUFFER;

```

Figure 1

```

with TEXT_IO, R_CONTROL;                                --#
use TEXT_IO, R_CONTROL;                                  --#
procedure BOUNDED_BUFFER is

    task PRODUCER;
    task CONSUMER;
    task BUFFER_CONTROL is
        entry DEPOSIT(C : in CHARACTER);
        entry WITHDRAW(C : out CHARACTER);
    end BUFFER_CONTROL;

    task body PRODUCER is
        ID : constant INTEGER := 1;  --# task number of PRODUCER
    begin
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        DEPOSIT('A');
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        DEPOSIT('B');
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        DEPOSIT('C');
    end PRODUCER;

    task body CONSUMER is
        H : CHARACTER;
        ID : constant INTEGER := 2;  --# task number of CONSUMER
    begin
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        WITHDRAW(H);
        PUT(H);
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        WITHDRAW(H);
        PUT(H);
        CONTROL.REQUEST_R_PERMIT(ID);                --#
        WITHDRAW(H);
        PUT(H);
    end CONSUMER;

```

Figure 2(a)

```

task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
begin
    loop
        select
            when COUNT < SIZE =>
                accept DEPOSIT(C : in CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;    --#
                    BUFFER(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod SIZE + 1;
                COUNT := COUNT + 1;
            or when COUNT > 0 =>
                accept WITHDRAW(C : out CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;    --#
                    C := BUFFER(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod SIZE + 1;
                COUNT := COUNT - 1;
            or terminate;
        end select;
    end loop;
end BUFFER_CONTROL;

begin
    null;
end BOUNDED_BUFFER;

```

Figure 2(b)

```

package RESOURCE is
  procedure READ(X:out REAL);
  procedure WRITE(X:in REAL);
end RESOURCE;
package body RESOURCE is
  SHARED_DATA : REAL := 0.0;
  procedure READ(X:out REAL);
  begin
    RW.START_READ;
    X := SHARED_DATA;
    RW.END_READ;
  end READ;
  procedure WRITE(X:REAL);
  begin
    RW.START_WRITE;
    SHARED_DATA := X;
    RW.END_WRITE;
  end WRITE;
  task RW is
    entry START_READ;
    entry END_READ;
    entry START_WRITE;
    entry END_WRITE;
  end RW;
  task body RW is
    NO_READERS : NATURAL := 0;
    WRITER_PRESENT : BOOLEAN := FALSE;
  begin
    loop
      select
        when not WRITER_PRESENT and
          START_WRITE'COUNT=0 =>
          accept START_READ;
          NO_READERS := NO_READERS + 1;
        or
          accept END_READ;
          NO_READERS := NO_READERS - 1;
        or
          when not WRITER_PRESENT and
            NO_READERS=0 =>
            accept START_WRITE;
            WRITER_PRESENT := TRUE;
        or
          accept END_WRITE;
          WRITER_PRESENT := FALSE;
          for L in 1..START_READ'COUNT loop
            accept START_READ;
            NO_READERS := NO_READERS + 1;
          end loop;
        end select;
      end loop;
    end RW;
  end RESOURCE;

```

Figure 3

	START_READ	END_READ	START_WRITE	END_WRITE	
			W2 (1)		
			W1 (6)		
(1)	*****	*****	*****	*****	select
				W2 (2)	
		R3 (4)			
(2)	*****	*****	*****	*****	select
(3)	*****	*****	*****	*****	COUNT
(4)	*****	*****	*****	*****	accept
		R3 (5)			
(5)	*****	*****	*****	*****	select
		R1 (9)			
(6)	*****	*****	*****	*****	select
				W1 (7)	
(7)	*****	*****	*****	*****	select
(8)	*****	*****	*****	*****	COUNT
(9)	*****	*****	*****	*****	accept
		R2 (10)			
(10)	*****	*****	*****	*****	select
		R2 (11)			
		R1 (12)			
(11)	*****	*****	*****	*****	select
(12)	*****	*****	*****	*****	select

Figure 4

```

with BAR_CONTROL; use BAR_CONTROL;           --#
package RESOURCE is
    ...
    procedure READ(X:out REAL; ID:INTEGER);    --#
    procedure WRITE(X:REAL; ID:INTEGER);       --#
end RESOURCE;

package body RESOURCE is
    SHARED_DATA : REAL := 0.0;

    procedure READ(X:out REAL; ID:INTEGER) is   --#
    begin
        CONTROL.REQUEST_A_PERMIT(ID);          --#
        RW.START_READ;
        X := SHARED_DATA;
        CONTROL.REQUEST_A_PERMIT(ID);          --#
        RW.END_READ;
    end READ;

    procedure WRITE(X:REAL; ID:INTEGER) is     --#
    begin
        CONTROL.REQUEST_A_PERMIT(ID);          --#
        RW.START_WRITE;
        SHARED_DATA := X;
        CONTROL.REQUEST_A_PERMIT(ID);          --#
        RW.END_WRITE;
    end WRITE;

    task RW is
        entry START_READ;
        entry END_READ;
        entry START_WRITE;
        entry END_WRITE;
    end RW;

```

Figure 5(a)

```

task body RW is
  NO_READERS : NATURAL := 0;
  WRITER_PRESENT : BOOLEAN := FALSE;
  ID : constant INTEGER := ...;          --#
  TOTAL_ARRIVALS : INTEGER;              --#
  TEMP : INTEGER;                        --#
begin
  loop
    WAIT_FOR_RENDEZVOUS;                  --###
    select
      when not WRITER_PRESENT and
        START_WRITE'COUNT=0 =>
        accept START_READ;
        CONTROL.RELEASE_R_PERMIT;        --#
        NO_READERS := NO_READERS + 1;
      or
        accept END_READ;
        CONTROL.RELEASE_R_PERMIT;        --#
        NO_READERS := NO_READERS - 1;
      or
        when not WRITER_PRESENT and
          NO_READERS=0 =>
          accept START_WRITE;
          CONTROL.RELEASE_R_PERMIT;      --#
          WRITER_PRESENT := TRUE;
      or
        accept END_WRITE;
        CONTROL.RELEASE_R_PERMIT;        --#
        WRITER_PRESENT := FALSE;
        WAIT_FOR_RENDEZVOUS;            --###
        TEMP := START_READ'COUNT;      --#
        CONTROL.RELEASE_R_PERMIT;        --#
        for L in 1..TEMP loop
          WAIT_FOR_RENDEZVOUS;          --###
          accept START_READ;
          CONTROL.RELEASE_R_PERMIT;      --#
          NO_READERS := NO_READERS + 1;
        end loop;
      end select;
    end loop;
  end RW;
end RESOURCE;

```

Figure 5(b)

```

CONTROL.REQUEST_R_PERMIT(ID)(TOTAL_ARRIVALS);
CURRENT_ARRIVALS := sum of COUNT attributes of all entries of
                    task ID;
while CURRENT_ARRIVALS < TOTAL_ARRIVALS loop
    CONTROL.WAIT_ARRIVAL;
    PREVIOUS_ARRIVALS := CURRENT_ARRIVALS;
    CURRENT_ARRIVALS := sum of COUNT attributes of all entries of
                        task ID;
    while CURRENT_ARRIVALS = PREVIOUS_ARRIVALS loop
        delay ...; -- to reduce the amount of busy-waiting
        CURRENT_ARRIVALS := sum of COUNT attributes of all entries
                        of task ID;
    end loop;
    CONTROL.RELEASE_A_PERMIT;
end loop;

```

Figure 6. Code WAIT_FOR_RENDEZVOUS for Task ID

```

with TEXT_IO; use TEXT_IO;
procedure BOUNDED_BUFFER_NEW is

    task PRODUCER;
    task CONSUMER;
    task BUFFER_CONTROL is
        entry DEPOSIT(C : in CHARACTER);
        entry WITHDRAW(C : out CHARACTER);
    end BUFFER_CONTROL;

    task body PRODUCER is
        C : CHARACTER;
    begin
        loop
            GET(C);
            select
                DEPOSIT(C);
            else
                null;
            end select;
            exit when C='%' ; -- % is the last symbol
        end loop;
    end PRODUCER;

    task body CONSUMER is
        C : CHARACTER;
    begin
        loop
            WITHDRAW(C);
            PUT(C);
            exit when C='%' ;
        end loop;
    end CONSUMER;

```

Figure 7(a)

```

task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
begin
    loop
        select
            when COUNT < SIZE =>
                accept DEPOSIT(C : in CHARACTER) do
                    BUFFER(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod SIZE + 1;
                COUNT := COUNT + 1;
            or when COUNT > 0 =>
                accept WITHDRAW(C : out CHARACTER) do
                    C := BUFFER(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod SIZE + 1;
                COUNT := COUNT - 1;
            or delay 3.0;
                exit;
            end select;
        end loop;
    end BUFFER_CONTROL;

begin
    null;
end BOUNDED_BUFFER_NEW;

```

Figure 7(b)

	DEPOSIT		WITHDRAW	
	-----		-----	
	P (1)			
			C (2)	
(1)	***** accept			
	P (4)			
(2)	***** accept			
			C (4)	
	P (3)			
(3)	***** accept			
(4)	***** accept			
(5)	***** delay			

Figure 8

```

with TEXT_IO, BAR_CONTROL;           --#
use  TEXT_IO, BAR_CONTROL;           --#
procedure BOUNDED_BUFFER_NEW is

    task PRODUCER;
    task CONSUMER;
    task BUFFER_CONTROL is
        entry DEPOSIT(C : in CHARACTER);
        entry WITHDRAW(C : out CHARACTER);
    end BUFFER_CONTROL;

    task body PRODUCER is
        ID : constant INTEGER := 1;  --# task number OF PRODUCER
        C : CHARACTER;
        STATUS : BOOLEAN;             --#
    begin
        loop
            GET(C);
            CONTROL.SELECT_REQUEST_A_PERMIT(ID)(STATUS);  --#
            if STATUS then              --#
                DEPOSIT(C);
            else
                null;
            end if;
            exit when C='%'; -- % is the last symbol
        end loop;
    end PRODUCER;

    task body CONSUMER is
        C : CHARACTER;
        ID : constant INTEGER := 2;  --# task number of CONSUMER
    begin
        loop
            CONTROL.REQUEST_A_PERMIT(ID);  --#
            WITHDRAW(C);
            PUT(C);
            exit when C='%';
        end loop;
    end CONSUMER;

```

Figure 9(a)

```

task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
begin
    loop
        WAIT_FOR_RENDEZVOUS;          --###
        select
            when COUNT < SIZE =>
                accept DEPOSIT(C : in CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;    --#
                    BUFFER(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod SIZE + 1;
                COUNT := COUNT + 1;
            or when COUNT > 0 =>
                accept WITHDRAW(C : out CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;    --#
                    C := BUFFER(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod SIZE + 1;
                COUNT := COUNT - 1;
            or delay 3.0;
                CONTROL.RELEASE_R_PERMIT;
                exit;
        end select;
    end loop;
end BUFFER_CONTROL;

begin
    null;
end BOUNDED_BUFFER_NEW;

```

Figure 9(b)

```

task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
begin
    loop
        select
            when COUNT < SIZE =>
                accept DEPOSIT(C : in CHARACTER) do
                    BUFFER(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod SIZE + 1;
                COUNT := COUNT + 1;
            or when COUNT > 0 =>
                accept WITHDRAW(C : out CHARACTER) do
                    C := BUFFER(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod SIZE + 1;
                COUNT := COUNT - 1;
            or when (DEPOSIT'COUNT = 1) and (WITHDRAW'COUNT = 1) =>
                accept DEPOSIT(C : in CHARACTER);
            or delay 3.0;
                exit;
            end select;
        end loop;
    end BUFFER_CONTROL;

```

Figure 10

	DEPOSIT	WITHDRAW	
	-----	-----	
	P (1)		
(1)	*****		accept --#1
		C (3)	
	P (2)		
(2)	*****		accept --#3
(3)	*****		accept --#2
(4)	*****		delay --#4

Figure 11


```

task body BUFFER_CONTROL is
    SIZE : constant INTEGER := 2;
    BUFFER : array(1..SIZE) of CHARACTER;
    COUNT : INTEGER range 0..SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1..SIZE := 1;
    MAX_SELECT_INDEX : constant INTEGER := 4;
    GUARD : array(0..MAX_SELECT_INDEX) of BOOLEAN;
begin
    loop
        WAIT_FOR_RENDEZVOUS;
        select
            when (COUNT < SIZE) and (GUARD(1)) =>
                accept DEPOSIT(C : in CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;
                    BUFFER(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod SIZE + 1;
                COUNT := COUNT + 1;
            or when (COUNT > 0) and (GUARD(2)) =>
                accept WITHDRAW(C : out CHARACTER) do
                    CONTROL.RELEASE_R_PERMIT;
                    C := BUFFER(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod SIZE + 1;
                COUNT := COUNT - 1;
            or when ((DEPOSIT'COUNT=1) and (WITHDRAW'COUNT=1))
                and (GUARD(3)) =>
                accept DEPOSIT(C : in CHARACTER);
            or when GUARD(4) =>
                delay 3.0;
                CONTROL.RELEASE_R_PERMIT;
                exit;
        end select;
    end loop;
end BUFFER_CONTROL;

```

Figure 12

AN EMPIRICAL STUDY OF TESTING
CONCURRENT Ada* PROGRAMS

K. C. TAI and RICHARD H. CARVER

DEPARTMENT OF COMPUTER SCIENCE
NORTH CAROLINA STATE UNIVERSITY
RALEIGH, NORTH CAROLINA, USA

EVELYN OBAID

DEPARTMENT OF MATHEMATICS
& COMPUTER SCIENCE
SAN JOSE STATE UNIVERSITY
SAN JOSE, CALIFORNIA, USA

* Ada IS A REGISTERED TRADEMARK OF THE
US GOVERNMENT - AJPO

A PROBLEM WITH THE CORRECTNESS OF
CONCURRENT ADA PROGRAMS

AN EXECUTION OF A CONCURRENT Ada PROGRAM WITH
AN INPUT PRODUCES A SEQUENCE OF RENDEZVOUS
CALLED A RENDEZVOUS SEQUENCE (R-SEQUENCE).

LET P BE A CONCURRENT Ada PROGRAM. REPEATED
EXECUTIONS OF P WITH INPUT X MAY PRODUCE
DIFFERENT R-SEQUENCES.

FOR P WITH INPUT X, THE SPECIFICATION OF P
DEFINES A SET OF VALID R-SEQUENCES OF P WITH
INPUT X, DENOTED BY $VALID(P,X)$.

FOR P WITH INPUT X, THE IMPLEMENTATION OF P
DEFINES A SET OF FEASIBLE R-SEQUENCES OF P WITH
INPUT X, DENOTED BY $FEASIBLE(P,X)$.

P IS SAID TO HAVE A RENDEZVOUS ERROR IF FOR
SOME INPUT X, $VALID(P,X) \neq FEASIBLE(P,X)$.

HOW TO DETERMINE WHETHER OR NOT P CONTAINS
RENDEZVOUS ERRORS?

- VERIFICATION
- TESTING

THE PAPER IN THE WORKSHOP PROCEEDINGS SHOWS
THE RESULT OF APPLYING OUR TESTING
TECHNIQUES. DETAILS OF THESE TECHNIQUES
ARE GIVEN IN [TAI86a,b,c].

THE CONVENTIONAL APPROACH TO TESTING
A PROGRAM

- SELECT INPUTS OF THIS PROGRAM
- EXECUTE THIS PROGRAM WITH EACH TEST
INPUT EXACTLY ONCE AND CHECK THE
RESULT

THE ABOVE APPROACH IS NOT APPROPRIATE FOR
TESTING CONCURRENT PROGRAMS.

PROBLEMS WITH TESTING CONCURRENT Ada PROGRAMS

PROBLEM 1. CORRECTNESS PROBLEM

THE CORRECTNESS OF P WITH INPUT X CANNOT
BE DETERMINED BY A SINGLE EXECUTION OF
P WITH INPUT X.

HOW TO DETERMINE THE CORRECTNESS OF P
WITH INPUT X BY TESTING?

THE REPEATED EXECUTION APPROACH

- REPEATEDLY EXECUTE P WITH INPUT X
- * RANDOM PRODUCTION OF FEASIBLE
R-SEQUENCES
- * IMPOSSIBLE TO DETECT THE EXISTENCE OF
VALID, BUT INFEASIBLE R-SEQUENCES

THE DETERMINISTIC EXECUTION APPROACH

- SELECT A SET OF R-SEQUENCES
- FOR EACH R-SEQUENCE S,
 - DETERMINE THE FEASIBILITY OF S FOR
P WITH INPUT X
 - IF S IS FEASIBLE, REPRODUCE S AND
CHECK THE RESULT

PROBLEM 2. REPRODUCIBLE TESTING PROBLEM

ASSUME THAT AN EXECUTION OF P WITH INPUT X HAS PRODUCED AN R-SEQUENCE S WITH AN INCORRECT RESULT.

(S IS A FEASIBLE R-SEQUENCE BECAUSE IT IS ALLOWED BY THE IMPLEMENTATION OF P.)

IN ORDER TO UNDERSTAND HOW THE INCORRECT RESULT WAS PRODUCED, WE WOULD LIKE TO REPRODUCE S.

HOWEVER, THERE IS NO GUARANTEE THAT S CAN BE REPRODUCED BY ONE OR MORE EXECUTIONS OF P WITH INPUT X.

HOW TO REPRODUCE A FEASIBLE R-SEQUENCE?

PROBLEM 3. FEASIBILITY PROBLEM

ASSUME THAT AN EXECUTION OF P WITH INPUT X HAS PRODUCED AN INVALID R-SEQUENCE S'.

(S' IS NOT ALLOWED BY THE SPECIFICATION OF P.)

AFTER MAKING CORRECTIONS, WE NEED TO KNOW WHETHER OR NOT S' CAN STILL BE PRODUCED BY THE MODIFIED P WITH INPUT X.

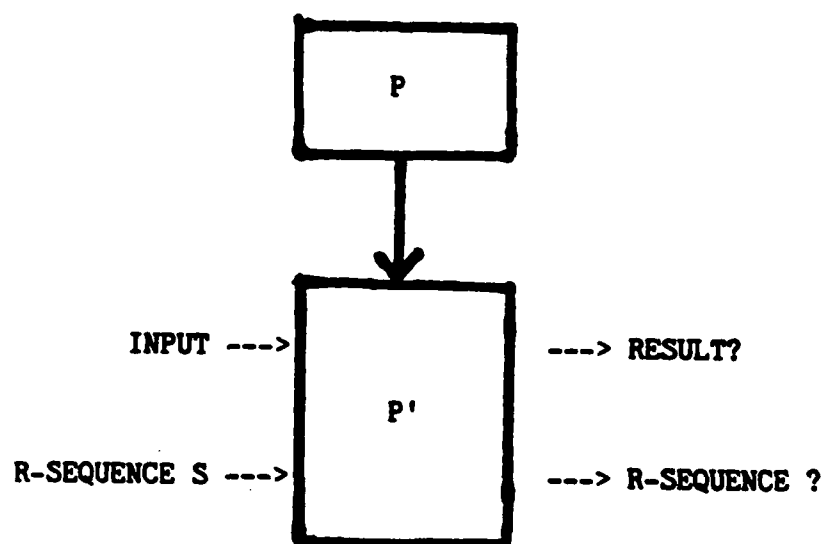
HOWEVER, EVEN IF MANY EXECUTIONS OF THE MODIFIED P WITH INPUT X HAVE NOT PRODUCED S', WE CANNOT SAY THAT S' IS INFEASIBLE FOR P WITH INPUT X.

HOW TO DETERMINE THE FEASIBILITY OF AN R-SEQUENCE?

THE STRATEGY FOR SOLVING THE FEASIBILITY AND REPRODUCIBLE TESTING PROBLEMS

LET P BE A CONCURRENT Ada PROGRAM. P IS
TRANSFORMED INTO ANOTHER CONCURRENT Ada
PROGRAM P' .

ANY EXECUTION OF P' WITH (X,S) AS INPUT
REPRODUCES S AS ITS R-SEQUENCE, IF AND
ONLY IF S IS FEASIBLE.



THE REPRODUCIBLE TESTING PROBLEM FOR Ada
IS TO REPRODUCE FEASIBLE R-SEQUENCES.

WHAT IS AN R-SEQUENCE OF A CONCURRENT Ada
PROGRAM P?

A SEQUENCE OF STARTS OF RENDEZVOUS

(..., (Ci, Ei), ...), WHERE

Ci: THE CALLER OF THE ith RENDEZVOUS

Ei: THE ENTRY NAME OF THE ith RENDEZVOUS

THE ABOVE DEFINITION OF AN R-SEQUENCE IS
INSUFFICIENT TO DETERMINE THE RESULT OF
AN EXECUTION OF P WITH INPUT X IF P
CONTAINS

- SELECTIVE WAIT STATEMENTS WITH DELAY
ALTERNATIVES OR ELSE PARTS
- SHARED VARIABLES
- ACCESSES TO THE COUNT ATTRIBUTES OF
TASK ENTRIES

(INTERRUPTS AND THE USE OF THE TIME CLOCK
ARE NOT CONSIDERED.)

HOW TO REPRODUCE A FEASIBLE R-SEQUENCE?

BASIC APPROACH:

TO CONTROL THE EXECUTION SEQUENCE OF
ENTRY CALLS

AN ENTRY CALL IS ALLOWED TO BE ISSUED
ONLY AFTER THE PREVIOUS RENDEZVOUS HAS
STARTED AND WHEN THIS CALL IS FOR THE
NEXT RENDEZVOUS.

- ISSUING OF THIS ENTRY CALL
- ARRIVAL OF THIS ENTRY CALL
- ACCEPTANCE OF THIS ENTRY CALL
- ISSUING OF THE NEXT ENTRY CALL

...

PROBLEMS:

- PSEUDO RENDEZVOUS
- CONDITIONAL/TIMED ENTRY CALLS

task type MESSAGE CARRIER is
 entry BEGIN_RECEIVE;
 entry END_RECEIVE;
 entry BROADCAST (M:MESSAGE);

end;

receivers: R1, R2, R3

broadcasters B1, B2

entry name abbreviation:

 B_R, E_R, BR, RE

valid R_sequences:

- ① (B1, BR, R1.B_R, R1.E_R, R2.B_R, R2.E_R)
- ② (B1.BR, R1.B_R, R2.B_R, R1.E_R, R2.E_R)
- ③ (R1.B_R, R2.B_R, B1.BR, R2.E_R, R1.E_R)

invalid R_sequences:

- ④ (B1.BR, R1.B_R, B2.Br,.....)
- ⑤ (R1.B_R, R2.B_R, B1.BR, R3.B_R,...)

⑤ is feasible if "/=" is changed to "=" in the
3rd alternative of the select stmt

REAL-RENDEZVOUS: EXECUTION OF AN ACCEPT
STATEMENT

PSEUDO-RENDEZVOUS:

- EXECUTION OF AN ELSE PART OR DELAY
ALTERNATIVE
- ACCESS TO A SHARED VARIABLE
- ACCESS TO AN ENTRY'S COUNT ATTRIBUTE

AN R-SEQUENCE IS REPRESENTED AS

(...,(Vi,Ci,Ei),...), WHERE

Vi: THE TYPE OF THE *i*th RENDEZVOUS

Ci: THE CALLER OF THE *i*th RENDEZVOUS

Ei: THE ENTRY NAME OF THE *i*th RENDEZVOUS

A PSEUDO-RENDEZVOUS MAY REQUIRE ZERO,
ONE, OR MORE ARRIVALS OF ENTRY CALLS.

THE REPRODUCTION OF AN R-SEQUENCE
REQUIRES THE REPRODUCTION OF ARRIVALS AND
RENDEZVOUS OF ENTRY CALLS.

A SEQUENCE OF ARRIVALS AND RENDEZVOUS OF ENTRY CALLS IS CALLED AN ARRIVAL-RENDEZVOUS SEQUENCE (AR-SEQUENCE).

AN AR-SEQUENCE CAN BE DESCRIBED AS $(..., A_i, (V_i, C_i, E_i), ...)$ WHERE A_i IS THE ARRIVAL SEQUENCE BETWEEN THE i th and $(i+1)$ th RENDEZVOUS.

AN ARRIVAL SEQUENCE IS $(..., (L_j, B_j), ...)$ WHERE

L_j : THE CALLER OF THE j th ARRIVAL

B_j : THE TYPE OF THE j th ARRIVAL

- BLOCKING ENTRY CALL

- CONDITIONAL/TIMED ENTRY CALL

(INDICATION OF SUCCESS/FAILURE)

A FEASIBLE AR-SEQUENCE CAN BE SIMPLIFIED TO $(..., A_i, (V_i, E_i), ...)$.

HOW TO REPRODUCE A FEASIBLE AR-SEQUENCE?

NOT EVERY FEASIBLE AR-SEQUENCE CAN BE
REPRODUCED DETERMINISTICALLY.

WHEN A SELECT STATEMENT IS EXECUTED, IF
THERE ARE TWO OR MORE POSSIBLE
RENDEZVOUS, ONE OF THEM IS SELECTED
RANDOMLY.

AN AR-SEQUENCE IS SAID TO BE A
DETERMINISTIC AR-SEQUENCE (DAR-SEQUENCE)
IF EACH RENDEZVOUS AT A SELECT
STATEMENT IS THE ONLY POSSIBLE
RENDEZVOUS.

IN GENERAL, A NON-DETERMINISTIC
AR-SEQUENCE CAN BE TRANSFORMED INTO A
DAR-SEQUENCE.

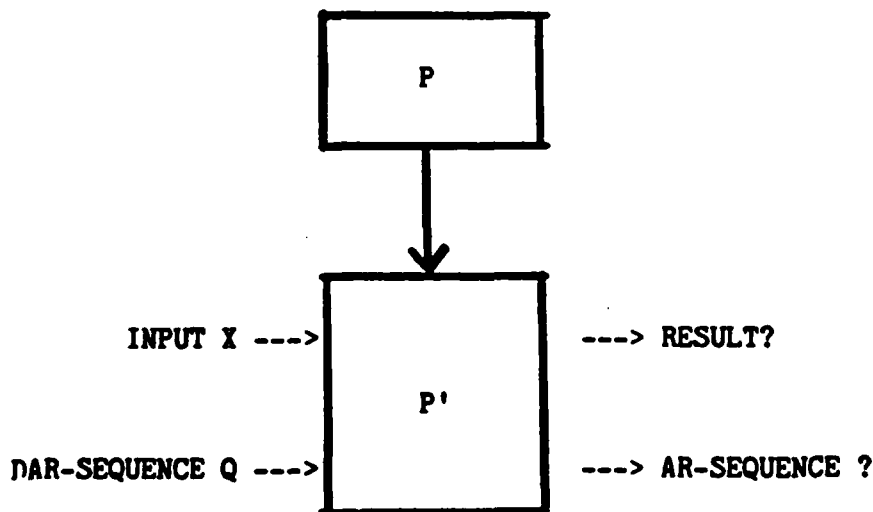
A DAR-SEQUENCE CAN BE DESCRIBED AS
(..., $A_i, (V_i, R_i), \dots$) WHERE
 R_i : THE NAME OF THE TASK EXECUTING THE
ith RENDEZVOUS

A DAR-SEQUENCE IS SAID TO BE A BATCH AR-SEQUENCE (BAR-SEQUENCE) PROVIDED THAT IF THE i th RENDEZVOUS IS EXECUTED BY TASK T, THEN ALL ENTRY CALLS BETWEEN THE $(i-1)$ th AND THE i th RENDEZVOUS ARE TO TASK T.

HOW TO REPRODUCE A FEASIBLE DAR-SEQUENCE OF P?

THE BAR_PERMIT METHOD:
TO TRANSFORM P INTO ANOTHER CONCURRENT Ada PROGRAM P' IN ORDER TO CONTROL THE EXECUTION SEQUENCE OF

- ENTRY CALL ARRIVALS
- RENDEZVOUS STARTS.



FUTURE RESEARCH

- DEVELOPMENT OF AUTOMATED TOOLS FOR REPRODUCIBLE TESTING
- GENERATION OF R-SEQUENCES EFFECTIVE FOR ERROR DETECTION

IN RECENT YEARS, A NUMBER OF MODELS FOR SPECIFYING CONCURRENCY HAVE BEEN DEVELOPED. THESE MODELS WILL BE STUDIED FOR TEST GENERATION.

- EMPIRICAL STUDY OF TESTING CONCURRENT Ada PROGRAMS

TELEPHONE NUMBER: (919) 737-7862

USENET ADDRESS: MCNC!NCSU!KCT

Title: Logical foundations and formal verification

Author: R.B.Jones Ref: DTC/RBJ/011 Issue: 2

Address: ICL Defence Systems
Eskdale Road
Winnersh
Berkshire RG11 5TT
ENGLAND, U.K.

Telephone: 0044-734-693131 x 6536

Telex: 847557

Usenet: ..!mcvax!ukc!stc!rbj

Date: 23th June 1986

Keywords: VERIFICATION LOGIC FOUNDATIONS

Abstract:

This position paper presents an approach to the design and development of environments for the production of computer systems for which we require to have very high degrees of assurance of correctness.

The approach is shaped by particular concern for:

- a) The soundness of the logical framework within which the correctness of the implementations is to be established.
- b) The inviolacy of the logical framework to errors on the part of the user.
- c) The means whereby the correctness of the implementation of the environment may be assured.

In consequence the following characterise the approach proposed:

- i) The approach is "foundational" rather than axiomatic. By this we mean that a single logical foundation is to be established during system design, and that users are permitted only definitional facilities which are guaranteed not to compromise the consistency of the foundation.
- ii) The foundation is supported by some philosophical examination of the nature of logical truth, and by careful examination of the intended domain of discourse (our "ontology") and the required expressiveness of the logic.

- iii) Both the foundation and its implementation are "reductionist". This means that the foundation is constructed from the simplest possible core, by the process of introducing new definitions, new syntactic forms, and exploiting (proven) derived rules of inference. The implementation is similarly to be built from a very small core in a carefully structured way. This reductionism is intended to provide maximal confidence in the consistency of our formal system, and the correctness of its implementation. We expect it to lead also to economies in implementation.
- iv) The foundation is type-free. A type system is to be constructed over the foundation for the purpose of providing transparent means of specifying the properties of the entities in the domain of discourse. These objects remain type-free, in the sense that self-application is permitted and polymorphic functions are "first class" entities.

Our philosophical position has an intuitionistic flavour. We take the (absolute) truths of logic to be those statements which correctly express the consequences of applying correctly some effective procedure. We suppose the correctness of execution of the elementary steps of an effective procedure to be supportable only by an appeal to the intuition. This philosophical position provides some of the motivation for reducing our formal system to the smallest possible core. This ensures that our intuitions are relied upon no more than is inescapable.

Three levels of language are currently envisaged, each corresponding also to a system architecture, and a stage in development. The lowest level is illustrated by a formal system corresponding closely to pure combinatory logic. The middle level is intended to be an application independent type theory. The types in this system correspond to (partial) specifications and to recursively enumerable sets of terms which satisfy the specifications. At the third level are the prime languages for system development including abstract specification languages and implementation languages. Where possible specifications for implementations will be expressed in extensions to the programming language type system. The semantics of all these languages is ultimately expressed in terms of our primitive logical foundation, and hence the development is axiomatic only in the core foundation system.

0. CONTENTS

1. INTRODUCTION
2. PHILOSOPHY AND ONTOLOGY
 - 2.1 Philosophical positions
 - 2.2 Ontology
 - 2.3 Logical Pluralism and Conventionalism
 - 2.4 Neo-constructive Ontology
3. PRIMITIVE FORMALISATION
 - 3.1 Introduction
 - 3.2 Syntax
 - 3.3 Axioms
 - 3.4 Inference Rules
 - 3.5 Abstraction
 - 3.6 Definitions for encodings
 - 3.7 Remarks on the Primitive Formalism
4. TYPES AND SPECIFICATIONS
 - 4.1 Introduction
 - 4.2 Recursive functions
 - 4.3 Recursive sets
 - 4.4 Recursively enumerable sets
 - 4.5 Function spaces
 - 4.6 Derived rules of inference
 - 4.7 Types as values
5. APPLICATION LANGUAGES
6. IMPLEMENTATION
7. VERIFICATION
8. CONCLUSIONS
9. REFERENCES

1. INTRODUCTION

This paper proposes an approach to the problem of building support environments for the development of very highly assured software. It does so, not from a pragmatic viewpoint, but from an idealist one. It represents an attempt to promote the convergence of computer science with constructive mathematics.

The ideas presented here are by no means fully worked out. They are presented as a basis for a programme of work, with a view to obtaining some feedback on the merits of the proposals.

The first four sections present our approach to logical foundations, and suggest how the most elementary logical basis might be built up by stages into sophisticated high level specification and implementation languages.

The first stage is philosophical, and consists in adopting an attitude about the nature of the mathematics of computing systems. The most important and concrete result of this philosophical stage is an ontology - a statement identifying the entities which we take to be the subject matter of computer science and which will constitute the domain of discourse of the formal systems we develop. Included in this stage is clarification of what sort of statements we wish to be able to make about the objects in our domain of discourse.

Having identified our ontology, we next construct a minimal formal language. In this language we may make statements about objects in our domain of discourse, and we may undertake formal derivations which establish the truth of some of these statements. In order that we may be able to obtain the highest degree of assurance of the correctness of our development environment, and in particular that we should be able to assure ourselves of the consistency of our formal systems and the correctness of their implementations, primary objectives in choosing this first level of formalisation are generality and simplicity. It is intended to be easy to reason about rather than easy to reason in, and hence consists of the simplest possible sufficiently expressive formal system. This system is type-free.

The third stage is the establishment of a more usable formal language in which the properties of entities may be specified and proven. This is done by providing a new syntax for constructs expressible in the basic formal system, and by establishing derived inference rules which facilitate proofs. The language in which specifications are expressed may be regarded as a type theory. The types in this theory are however purely a means of expressing specifications. They place no restrictions on term formation, and play no part in securing the consistency of the logic. The "types" express properties of terms. A term may have many properties, and hence many types. The objects in the domain of discourse might therefore better be described as polytypical than polymorphic.

The formal "type-theory" we propose to establish during the third phase is the mathematical foundation for a variety of (more or less problem oriented) development languages, among which Ada might number. These languages are addressed during our fourth phase. Such languages would be established by specifying their syntax and providing a denotational semantics in the type-theory. Along with programming languages such as Ada, specialised specification languages would be established, together with appropriate libraries of derived inference techniques. The programmer is thereby provided with a formal language in which he may reason in as natural a way as possible about the properties of his programs.

In section 6 we give some indication of how we propose to implement a support environment for these languages. Our final section addresses the problem of verifying of such an environment.

2. PHILOSOPHY AND ONTOLOGY

2.1 Philosophical positions

Any formal foundation system for mathematics is necessarily connected more or less intimately with some philosophical position upon the nature of mathematics.

The three principal 'schools' of philosophy of mathematics in the twentieth century have been logicism, intuitionism, and formalism.

Logicism, of which Bertrand Russell was one of the principal proponents, is the thesis that the whole of mathematics is ultimately reducible to symbolic logic. In "The Principles of Mathematics" [Rus03] (the manuscript of which was completed on the last day of the 19th century), Russell states that: "The fact that all Mathematics is Symbolic Logic is one of the greatest discoveries of our age; and when this fact has been established, the remainder of the principles of mathematics consists in the analysis of Symbolic Logic itself."

Intuitionism, a school of thought most prominently associated with Brouwer rejects classical mathematics in favour of the more spartan constructive mathematics. According to Bishop [Bis67] an important element of the intuitionist position is that: "every mathematical statement ultimately expresses the fact that if we perform certain computations within the set of positive integers, we shall get certain results". Intuitionists reject some of the principles of classical logic, notably the law of the excluded middle.

Formalism, a doctrine and a programme due to Hilbert, is characterised by the view that classical mathematics may be established by formal derivation from plausible axioms, provided that the consistency of the formal axiomatisation is established by "finitary" or "constructive" means.

Of these positions only the intuitionist position has survived intact to the present day, though it remains a position which the majority of working mathematicians find unacceptable.

The logicist position failed to be established primarily because two of the principles (axioms) necessary for the development of classical mathematics are difficult to establish as principles of logic. Neither the axiom of infinity nor the axiom of choice can be convincingly shown to be logically necessary propositions.

The formalist programme was shown to be unachievable by Kurt Gödel. He demonstrated that classical mathematics is not completely formalisable, and that no formalisation of arithmetic can be proven consistent by finitary means [God31].

2.2 Ontology

Associated with each of the philosophical positions outlined above are underlying ideas on the population of the universe of mathematics, on what, as far as mathematics is concerned, "exists". Logicism and formalism share similar ontologies, since they both aim to provide a foundation for "classical" mathematics. They differ to some degree in the formal system in which mathematics is derived, and differ widely in how the formalisation is to be philosophically justified, but ontologically they are broadly similar.

The underlying ontology is that of a hierarchy of sets, built up in stages from a (possibly empty) collection of individuals. Hatcher describes this in [Hat82] (speaking with reference to the Zermelo-Fraenkel axiomatisation of set theory) "...the hierarchy of sets we envisage consists of all the sets we can obtain by starting with the null set and iterating, in the indicated manner, our power set and union operations any transfinite number of times."

This process rehabilitates Cantor's informal set theory, after Russell's paradox had shown Frege's formalisation of it to be inconsistent, by restricting abstraction so that no set can be formed until after the formation of all the sets which it contains. (this is not the same as requiring that a description of a set may not mention any sets not formed before it). This may be restated as the requirement that the transitive closure of the membership relation is anti-reflexive (and hence its reflexive closure is a partial ordering on the universe of sets). Though this last condition is not fully adhered to by all the formal foundation systems for classical mathematics, (Quine's New Foundations [Qui63] being one counter-example), one of its consequences, that functions may not be members of their own domains, is present in all classical foundation systems of which I am aware.

The intuition behind this ontological position is probably attributable to Bertrand Russell. The first attempt to articulate the idea is in [Rus03], and results from Russell's attempts to identify the logical errors which give rise to the paradoxes. When the idea is elaborated in Russell's Theory of Types [Rus08], it is easily confused with the proscription of impredicative definitions, but seems still to be a part of the underlying intuition. Although "first-order" axiomatisations of set theory began without such a clear commitment to a hierarchy [Zer08], in the later axiomatisations known now as ZF and NBG (see [Hat82]), the hierarchy is cleaned up by the inclusion of an axiom of regularity.

Russell's intuition cuts across the intuitions which are encouraged by an acquaintance with digital computers (for which he can hardly be blamed). In considering the behaviour of computers it is perhaps more natural to consider types as ways of interpreting objects. We can consider an object stored in a computer memory, at one moment as a data value, and at the next as a program, rule or function.

In computing we can accept a single countable domain and interpret the members of this domain in terms of types. There appears to be no clear intuitive reason to proscribe applying a rule to itself, and this is practically very useful.

The significance of the problem of self application of functions has been argued by Dana Scott in [Sco70] and elsewhere. The theoretical underpinning of denotational semantics has required and resulted in resolution of these difficulties within a classical framework. This is done by slimming down function spaces until they are small enough to be isomorphic to the domains over which the functions range. Self-application then requires a non-standard account of the result of applying a function to an argument. (One which doesn't require a set directly or indirectly to be a member of itself, as proscribed by the axiom of regularity)

This solution carries rather too much baggage with it to be entirely satisfactory as a foundation system. If we take it to be founded on a first order axiomatisation of set theory, then we have first implicitly accept the need for a hierarchy of types. Next, by choosing "first-order" logic, we determine to do mathematics in just the first level of this hierarchy, the individuals. Then we construct a set theoretic hierarchy within this domain of "individuals", and finally collapse this hierarchy by slimming down function spaces until they become homeomorphic with their domains. Having twice accepted a system almost designed to prevent self-application, it is not surprising that some mathematical sophistication is required to construct yet again a type free notion of function application within this framework.

Even where the formalisation of classical mathematics is not required, as in the work of Martin-Löf [Mar75,82] in formalising constructive mathematics, and that of Constable in constructing formal systems for the verification of programs [Con80], constraining the ontology to be hierarchic seems to have proven necessary to avoid inconsistency.

We choose to start from the beginning with a type free system. The difficulty here is in giving any "mathematical" respectability to the system. The term "mathematical" is now so strongly associated with classical set theory, that an account of semantics which does not ultimately result in denotations in classical set theory is in danger of being considered not mathematical.

In constructing a foundation system, our ontological intuitions are crucial, and the indications are that the richness of classical ontologies is incompatible with a natural account of self application. In determining on a foundation we will first identify our domain of discourse, which we consider an important step in ensuring consistency in the foundation. Before we do this we will examine more closely the idea of logical truth, since this provides additional motivation for our selected ontology.

2.3 Logical pluralism and Conventionalism

Concern for single foundation systems has largely been displaced by a pluralistic attitude to foundations. Logicians and Philosophers study different foundation systems on their technical merits without feeling bound to choose between them, Mathematicians are mostly able to work in a way that can reasonably be interpreted in any of the classical systems, and Computer Scientists feel free to adopt or invent any formal system which suits their purposes.

These pluralistic attitudes have a philosophical counterpart in the linguistic-conventionalist account of the status of logical principles. This principle states (roughly) that a logical truth is true in virtue of the meanings of the terms it contains, i.e. in virtue of accepted linguistic conventions. Pluralism and conventionalism have in common that they seem to support the view that logical truths are not absolute, but are arbitrary.

If logical truth were entirely arbitrary, then it would likely be of limited utility. However, we know there to be, underlying the plurality of informal and formal logics, some common principles which we can claim to be absolute logical principles.

These principles are about conformance to rules. Either informally or formally, we suggest, that the idea of logical truth depends upon proof, that in the essence of the idea of proof is the view that proofs are checkable, and that the method of checking proofs be effective. There may be doubt about whether a statement has a proof, but given a putative proof of a statement there must be an "effective procedure" for testing whether it is indeed a proof.

We now know many languages within which effective procedures may be described, (lambda calculus, combinatory logic, Turing machines, recursive functions, Post productions...) and the fact that these have been shown to be equivalent in expressive power gives us a basis for claiming that the notion of effective computability is an absolute one. It is from this that our notion of absolute logical truth derives. The claim that a sentence is a theorem of a formal system is just the claim that a particular effectively computable partial function over sentences yields a token representative of "true" when evaluated on the given sentence.

The primitive formalism which we describe below, and hence the various languages which we construct from it, are capable of expressing and proving just those propositions which indicate the result of applying some effective procedure to some value.

2.4 Neo-constructive ontology

We now identify a domain of discourse and the properties we wish to express over this domain.

Since we are concerned to reason about the properties of computers and their programs we choose a more or less arbitrary denumerable domain, which we may consider as the collection of values storable in the memory of some ideal computing device. Functions are to be represented by conventions whereby the values in the domain may be interpreted as rules describing some computational procedure. Properties or predicates are identified with partial computable functions over the values in the domain into some subdomain designated as representing the truth values.

As an example, we select as a domain of discourse the free combinatory algebra generated from the constants K and S under the binary operation of application. By the use of an embedding of this domain into itself and a reduction process over the terms of the domain we are able to represent all partial computable functions over the domain using elements in the domain.

The reduction process is effected by the rules:

u	=> u	
((K u) v)	=> u	
((S u) v) w)	=> ((u w)(v w))	
(u v)	=> (x y)	if u => x and v => y

Where u,v,w,x,y are arbitrary values in our intended interpretation, and (u v) is the application of u to v.

=>* is defined as the transitive closure of =>.

Elements may be encoded into the domain using the rules shown below in section 3.6.

An element u of our domain is considered to satisfy the predicate represented by an element v if the application of v to the encoding of u , (which we write ' u ') is reducible to K , i.e. if

$$(v \text{ 'u' }) \Rightarrow^* K$$

The true propositions, are those elements of our domain which are reducible to K under the above reduction system.

3. PRIMITIVE FORMALISATION

3.1 Introduction

The primitive formal system is intended to be as simple as possible, so that we may have confidence in its consistency, and in the correctness of its implementation without either depending on proof in a less well founded formal system, or on proof within itself.

The opacity of the syntax and the inefficiency of the proof rules is acceptable at this level, both of these problems can be addressed without logical extension.

The key characteristics required at this stage are simplicity, consistency, expressiveness (with reference to what can be expressed, not how it is expressed), and completeness.

3.2 Syntax

atom ::= "S" | "K"

term ::= atom | "(" term term ")"

Henceforth:

$a, b, c, \dots$ are metavariables ranging over atoms.

$x, y, t, u, v, w, \dots$ are metavariables ranging over terms.

3.3 Axioms

$\vdash K$

The "standard interpretation"s of the terms "S" and "K" are the individuals S and K , and the juxtaposition of two terms denotes the application of the denotation of the one to the other. The algebra of terms is therefore isomorphic to our domain of interpretation. The axiom " $\vdash K$ " indicates that " K " is our version of the proposition "True".

Theorems of the form $(u \text{ 't' })$ may be interpreted as assertions that the term t satisfies the predicate represented by the term u .

3.4 Inference Rules

We first define the postfix substitution operator $[t/a]$:

$a[t/a] = t$
 $b[t/a] = b$ (provided $b \neq a$)
 $(u \ v)[t/a] = (u[t/a] \ v[t/a])$

Our inference rules are then:

$(K) \quad t[u/a] \quad \quad \quad |- \ t[(((K \ u)v)/a)]$
 $(S) \quad t[(((u \ w)(v \ w))/a)] \quad \quad \quad |- \ t[(((S \ u)v)w)/a]$

These rules are the inverse of the reduction relationship which determines the truth of a proposition. They therefore make theorems of just those terms whose denotation is reducible to K , and are therefore sound and complete. Since neither of the inference rules permits the derivation of an atomic theorem, "S" is not a theorem and the system is consistent in the sense of Post.

3.5 Abstraction

In the following sections for illustrative purposes we make liberal use of informal syntactic abbreviations. These are not a part of our primitive formal system, but we expect in due course to deal with such matters in fully formal ways. These include, dropping brackets (taking application as left associative), and using infix notation for some dyadic operations.

First we introduce abstraction as a notational abbreviation.

$I \quad \quad \quad \hat{=} \quad ((S \ K) \ K)$

We define $[a]t$ inductively as follows:

$[a]a \quad \quad \quad \hat{=} \quad I$
 $[a]b \quad \quad \quad \hat{=} \quad (K \ b)$
 provided $a \neq b$
 $[a](u \ v) \quad \hat{=} \quad ((S \ [a]u) \ [a]v)$
 $[a, b]t \quad \quad \hat{=} \quad [a][b]t$

and in general

$[a_1, a_2, \dots, a_n]t \hat{=} [a_1][a_2] \dots [a_n]t$

In definitions we may write:

$a \ b \hat{=} t$

for

$a \hat{=} [b] \ t$

or:

$a \ a1 \ a2 \ \dots \ an \ \hat{=} \ t$

for

$a \hat{=} [a1, a2, \dots, an] \ t$

Now, in order to permit recursive definitions we introduce the fixed point operator.

$Sap \ F \hat{=} F \ F$

$Y \ F \hat{=} Sap \ [X]F(XX)$

Note that $Y \ F = [X]F(XX) ([X]F(XX)) = F ([X]F(XX) [X]F(XX)) = F (Y \ F)$
i.e. $Y \ F$ is a fixed point of F .

Henceforth we will admit recursive definitions, writing:

$F \hat{=} E$

instead of

$F \hat{=} Y \ [F]E$

and

$F \ X \ Y \hat{=} E$

for

$F \hat{=} Y \ [F, X, Y]E$

etc.

3.6 Definitions for encodings

In this section we define an encoding of terms into normal terms. We use the notation 't' for the encoding of a term t.

$T \hat{=} K$

$F \ X \ Y \hat{=} Y$

$If \ X \ Then \ Y \ Else \ Z \hat{=} X \ Y \ Z$

$X \ And \ Y \hat{=} If \ X \ Then \ Y \ Else \ F$

$X \ Or \ Y \hat{=} If \ X \ Then \ T \ Else \ Y$

$X \ \Rightarrow \ Y \hat{=} If \ X \ Then \ Y \ Else \ T$

$X \ \Leftrightarrow \ Y \hat{=} (X \ \Rightarrow \ Y) \ And \ (Y \ \Rightarrow \ X)$

$Not \ X \hat{=} If \ X \ Then \ F \ Else \ T$

$\langle X, Y \rangle \ Z \hat{=} Z \ X \ Y$

```

Fst X      ≡    X T
Snd X      ≡    X F
'K'        ≡    <T,T>
'S'        ≡    <T,F>
Mk_app X Y ≡    <F,<X,Y>>
'XY'       ≡    Mk_app 'X' 'Y'
Is_app X   ≡    Not (Fst X)
Fun X      ≡    Fst (Snd X)
Arg X      ≡    Snd (Snd X)
    
```

The encoding is naturally extended to encode any defined term as the encoding of its definiens. This also allows double encodings such as "<T,T>". Note that the encoding of terms cannot be expressed as a term.

The decoding of encoded terms may now be defined as an example of a term representing a predicate over terms:

```

Decode X ≡    If   Is_app X
               Then (Decode (Fun X)) (Decode (Arg X))
               Else If Snd X
                     Then K
                     Else S
    
```

A partial encoding algorithm, defined over the set {T,F} may be defined:

```

Encode X = If X Then 'T' Else 'F'
    
```

Using the full encoding a every recursively enumerable set of terms RET may be represented by some term REP_RET such that:

```

for any term t,    |-REP_RET 't'
                   iff t ∈ RET.
    
```

3.7 Remarks on the primitive formalism

The key characteristics identified in section 3.1 were simplicity, consistency, expressiveness and completeness.

The system is evidently simple.

Its consistency in the sense of Post is immediately evident.

For each recursively enumerable subset of our intended domain of interpretation there is a term which represents that set. For each individual and recursively enumerable set of individuals there is a term which represents the proposition that the individual is a member of the set.

In any formal logic, the ground terms which can be proven to satisfy any given predicate defined in that logic, are recursively enumerable. There is therefore a formal sense in which our logic is as expressive as any formal logic can be. For any arbitrary formal system (assuming a reasonable definition of "formal system") the property of formulae known as theoremhood is expressible in our primitive logic. We therefore believe that the formalism is sufficiently expressive for our purposes, and constitutes a foundation system on which sufficiently rich theories can be constructed by the use of definitions only.

Finally, we can say that the system is complete in the following sense: all the propositions expressible are provable iff true. We cannot express classical negation in the system. i.e. there is no term which denotes the set of unprovable terms. This reflects the fact that the complement of a recursively enumerable set is not in general recursively enumerable.

4. TYPES AND SPECIFICATIONS

4.1 Introduction

By adding "syntactic sugar" and by building a library of definitions, our primitive language can be built up into a more sophisticated and usable one. The process of building a functional programming language in this way is well understood (or rather, the inverse problem, of implementing a functional language using combinators). For details see [Tur79a,b]. In our case the process is complicated by the need to use combinators for everything, whereas in implementations of functional languages combinators have been used only for passing parameters, and other environmental data. Examples of the use of combinators for arithmetic may be found in [Cur72] or [Hin72].

In this section we consider what it is that we are to take for sets, predicates, types and specifications, and how these may be constructed in our primitive formal system.

In first approaching this problem the value of the distinction between predicates and types was not clear.

The argument against a distinction between types and predicates in this context is as follows.

In logic, a primary role of types has been to constrain ontological commitment in order to secure the consistency of a logic. This has proved to be a simple and effective way of avoiding logical paradoxes. More recently radical divergence from this limited role has been adopted by extending the expressiveness of type systems and identifying types with propositions ([Mar75,82],[Con80]).

The use of type systems for securing consistency inhibits abstraction by guarding against the circularities inherent in polymorphism. For this reason we have adopted a type-free logical core (which is nevertheless consistent). In our view, whatever its technical merits, the identification of types with propositions is counter-intuitive, and a type system extended could not be more expressive than we would expect to be by the use of predicates in our logic. It is therefore not clear what we could expect to express in our logic by introducing types, which could not be expressed without them.

We have in fact found (what we consider to be) sufficient reason for introducing types as distinct from predicates, even though in our logic both types and predicates will correspond (in their own ways) to recursively enumerable sets of terms. The reason lies primarily in the opacity of terms representing predicates in our formalism. This renders much more difficult the definition of functions whose domain is intended to be a type of types, and makes all reasoning about types, and hence the properties of functions very tedious.

We therefore propose to introduce a type system as an encoding of predicates in a form which more transparently represents the intended interpretation of the terms which are members of the type. This is intended to provide the necessary transparency of the specifications expressed in our system. It will also provide a means of expressing, proving and applying derived rules of inference. We observe that, while "syntactic sugar" might be used to make specifications more transparent to the user, it is not sufficient to enable specifications to be data values upon which operations may (transparently) be performed.

It may therefore be noted that while strongly motivated by ideas closely related to intuitionism, our formal system, and our notion of 'type' has very little in common with the presentation of intuitionism due to Martin-Lof. [Mar75,82]

Within our framework any number of independent type systems could be introduced without danger of inconsistency. Each system would represent an alternative encoding of the recursively enumerable sets of combinatory terms. In this way we propose to provide coherent support for the linguistic pluralism necessary to provide optimal application development productivity.

In our system however, one type system will have a particular priority in having been designed to express derived rules of inference, and in having its type of derived rules built into an additional rule of inference permitting the application of any rule which has been proven sound.

By a derived rule in this context we do not mean derived rule in the sense in which the term is used in LCF ([Gor79]) and its variants. In these languages a derived rule is a procedure written in the metalanguage ML, (possibly using a library of "tactics" and "tacticals"), which computes a proof. In these systems there is no way of shortening a proof, but there are powerful facilities for automatic generation of proofs. In our system the primitive formalism is so primitive (more primitive than LCF), that proofs, even if automatically generated, would be too complex if genuine shortcuts were not available.

We therefore propose that any rule of inference which can be proven sound may be invoked to establish a theorem without the need to furnish a proof in the primitive system. A similar feature has been included in the Boyer-Moore theorem prover, [Boy81].

4.2 Recursive functions

Having chosen an encoding of terms into terms with distinct normal forms (this is a necessary, but possibly not a sufficient condition) we can represent recursive functions over terms, by terms which when applied to the encoding of a term in the domain, reduce to the encoding of the image of that term in the codomain.

That this is not possible without such an encoding follows from the Church-Rosser theorem, which has the consequence that two terms with similar normal form have the same image under any third combinatory term. The encoding enables terms with the same normal form to be distinguished by mapping them onto terms with distinct normal forms. This encoding algorithm is not, of course, expressible as a term, though there is a term (described above) which effects a decode (modulo weak inequality), and encoding over limited subsets of the terms is expressible (notably over $\{T, F\}$).

In the following subsections we show how type systems may be established by defining various operations over partial recursive functions which we describe as "type constructors". These are strictly the denotations of type constructors (unless we consider the special type-system with the identity function as a semantic mapping). Our terminology is still sub judice.

4.3 Recursive sets

By choosing representatives for the boolean values "true" and "false", (e.g. those given in section 3), we may represent characteristic functions by terms which represent boolean valued total recursive functions. These characteristic functions are the characteristic functions of recursive sets of terms.

Type constructors are easy to define over recursive sets. For example the following constructors can easily be seen to be definable as terms.

The unit type constructor U such that:

$U 't'$ is the characteristic function of $\{t\}$

for any term t , is just an algorithm for checking (intensional) equality of terms.

For each dyadic truth function there is a corresponding operation on recursive characteristic functions which can be simply constructed from the term representing a (possibly strict) implementation of the truth function (such as those in section 3.6), viz:

Union $X Y Z \quad \hat{=} (X Z) \text{ Or } (Y Z)$

Intersection $X Y Z \quad \hat{=} (X Z) \text{ And } (Y Z)$

Also:

Complement $X Y \hat{=} \text{ Not } (X Y)$

Under these operations recursive sets form a boolean algebra.

Unfortunately, as soon as we wish to introduce sets which are not decidable (with characteristic functions which are not total), the ease of constructing operators over types disappears. In the case of partial recursive functions non-strict logical operators are required, which can only be defined over encodings of characteristic functions.

4.4 Recursively enumerable sets

Partial characteristic functions are mappings which for any encoded term will yield either a term which reduces to the encoding of true, or of false, or a term which has no normal form. Such functions may be regarded as representing partial predicates, which correspond to pairs of disjoint recursively enumerable sets.

These are effectively closed under all truth functional logical operations and hence all these operations are themselves representable as terms.

Since partial characteristic functions sometimes fail to terminate, and the logical operators defined in section 3.6 are strict in their first argument, the methods used in section 4.3 fail to give satisfactory implementations of operations over recursively enumerable sets.

Furthermore, logical operations which are strict in neither argument are not expressible directly as terms in pure combinatory logic. More precisely, there is no term O such that for any pair of terms t, u :

$| - O t u \quad \text{iff} \quad | - t \text{ or } | - u$

We can express this function however if we use encodings of t and u . If t and u are available in an encoded form, then their non termination can be guarded against by emulating interleaved evaluation. So there is a term Ore (representing Or over encodings) such that for any pair of terms t, u :

$| - 't' Ore 'u' \quad \text{iff} \quad | - t \text{ or } | - u$

Furthermore, Ore can be defined in such a way that:

$| - Not ('t' Ore 'u') \quad \text{iff} \quad | - Not 't' \text{ and } | - Not 'u'$

So that $Ande$ (And over encodings) may be defined:

$Ande X Y \quad \hat{=} \quad Not ((Mk\_app 'Not' X) Ore (Mk\_app 'Not' Y))$

(giving: $Ande 'X' 'Y' = Not ('Not X' Ore 'Not Y')$)

By the use of encodings we can therefore express non-strict logical operations, with which well behaved operations over partial characteristic functions may be defined. (To do this we need a partial encoding function $EncEnc$, which can be defined over encoded terms. The definition is omitted.)

$(X Orp Y) Z \quad \hat{=} \quad (Mk\_app X (EncEnc Z)) Ore (Mk\_app Y (EncEnc Z))$

(giving: $('X' Orp 'Y') 'Z' = 'X' 'Z' Ore 'Y' 'Z'$)

$(X \text{ Andp } Y) Z \hat{=} (\text{Mk\_app } X (\text{EncEnc } Z)) \text{ Ande } (\text{Mk\_app } Y (\text{EncEnc } Z))$

(giving: ('X' Andp 'Y') 'Z' = 'X 'Z'' Ande 'Y 'Z''')

Orp and Andp correspond to the operations of union and intersection of recursively enumerable sets.

$\text{Notp } X Y \hat{=} \text{Not } (X Y)$

Notp provides a complement, but not a true complement. The classical complement of a recursively enumerable set is not in general recursively enumerable. The recursively enumerable set whose characteristic function is obtained by applying Notp to some characteristic function is not uniquely determined by the recursively enumerable set determined by the characteristic function. An interpretation of Notp in terms of operations on sets can only be given if partial characteristic functions are taken to represent, not single recursively enumerable sets, but pairs of disjoint recursively enumerable sets. In this case the logical operators correspond to set operations as follows:

$\langle a1, a2 \rangle \text{ Andp } \langle b1, b2 \rangle \Rightarrow \langle \text{intersection of } a1 \text{ and } b1, \text{union of } a2 \text{ and } b2 \rangle$

$\langle a1, a2 \rangle \text{ Orp } \langle b1, b2 \rangle \Rightarrow \langle \text{union of } a1 \text{ and } b1, \text{intersection of } a2 \text{ and } b2 \rangle$

$\text{Notp } \langle a, b \rangle \Rightarrow \langle b, a \rangle$

The logic thus obtained is not classical. For example, the law of the excluded middle does not hold.

Nor is it intuitionistic, since:

$A = \text{not not } A$

It is a three valued logic which, in its finite operations, corresponds to the three valued logic due to Kleene.

A merit of considering types as partial characteristic functions in this way, with complement defined, is that the total characteristic functions are also closed under these operations, and the restriction of these operations to total characteristic functions gives a true complement and a classical logic.

Other type constructors can be defined from these.

Using the pairing operation defined in section 3.6 we can define a product type constructor:

$(X \text{ prod } Y) Z \hat{=} (\text{Mk\_app } X (\text{EncEnc } (\text{Mk\_app } 'Fst' Z))) \text{ Andp } (\text{Mk\_app } Y (\text{EncEnc } (\text{Mk\_app } 'Snd' Z)))$

(giving: ('X' prod 'Y') 'Z' = 'X '(Fst Z)'' Andp 'Y '(Snd Z)'')

And a dependent product type constructor:

```
(X dprod Y) Z ≡      (Mk_app X (EncEnc (Mk_app 'Fst' Z))
                      Andp(Mk_app (Mk_app Y (Mk_app 'Fst' Z))
                              (EncEnc (Mk_app 'Snd' Z))))
```

where Y is a function which maps a value of type X onto a type.

The dependent product type constructor takes any type X, and a function which maps elements of type X to types, and delivers the type of pairs such that the type of the first component is X and the type of the second is determined by the value of the first component under Y. Dependent products types are important as candidate representatives of abstract data types.

The idea for dependent product constructors comes (to me) from Martin-Löf's ITT [Mar75,82], (where it is called "disjoint union of a family of types"), similar type constructors also occur (among other places) in PL/CV3 [Con80] and Pebble [Bur84], from which our terminology is derived.

4.5 Function spaces

The definition of function space constructors is more difficult.

We have so far been rather vague about which terms may be used as representatives of recursively enumerable sets. This is possible because all the type constructors we have illustrated so far behave well even if all terms are taken to represent recursively enumerable sets. Every term can be interpreted as determining a recursively enumerable set of terms, and so we could take the "type of types" to be the universe (represented by the term (K K)). When we come to constructing function spaces however, we have found no construction which is as insensitive to the representative chosen as is the case for the previous constructors.

A key question, (but not one which affects the viability of our proposal) is whether the space (A→B) of (total) computable functions from a recursively enumerable domain A into recursively enumerable codomain B is in general recursively enumerable. Similarly we would like to know whether dependent function spaces (which we write A-?>B) are recursively enumerable. If these spaces are enumerable, and if the operation of forming a representative combinator for a (dependent) function space from representative of the domain and codomains is computable, then we need only to determine one of the combinators which represents this computation and we have the basis for a maximally expressive type system.

If the function spaces are not enumerable, or if the type constructors are not effective then we will have to settle for an approximation (from below, i.e. a subset) to these spaces for which effective constructors can be discovered. We have not yet resolved this problem.

4.6 Derived rules of inference

Once we have decided how to define function spaces we expect to be able to use the function space constructor in an extra inference rule which will legitimise the use of derived rules of inference.

Since

$\vdash$ Decode 'Z' iff $\vdash$ Z

"Decode" is the type of theorems.

Consequently, for any encoded type X,

$\vdash$ Decode ('X' $\rightarrow$ 'Decode') 'Y'

and

$\vdash$ X 'Z'

implies

$\vdash$ Y Z

i.e. if Y maps elements of X into theorems, and Z is in X, then Y Z is a theorem. We therefore propose to add this one further rule of inference, which, provided our definitions are carried through correctly, will add no further theorems but will permit shorter proofs.

We should then be able to establish type inference rules as derived rules of inference.

4.7 Types as values

If types are identified with encodings of terms which represent partial characteristic functions over encodings of terms, they are data values. However, the form of such types bears little relationship to the constructors which were used to construct the type. If we require to be able to examine the type of an object, and discover with ease whether or not it is a product (for example) then a more transparent representation of types is required.

Such representations may be defined and may be given a denotational semantics by furnishing a semantic mapping into our clumsy representation. In particular, for any application language a type system may be devised specifically to express the types in that language, or to provide an extension of the programming language type system sufficiently rich to serve as a specification language.

5. APPLICATION LANGUAGES

The expressiveness of our formal system is sufficient we believe to define the denotational semantics of application development languages. By providing a logically secure framework within which specialised type theories with matching derived inference rules may be established, we hope to enable a close fit between application languages and specification languages. This may enable a development methodology in which specifications are evolved into implementations by stages which are supported by automatic verification.

6. IMPLEMENTATION

We provide here a very brief outline of how we propose to implement a support environment using our foundation systems.

Combinators have been used in the implementation of functional programming languages [Tur79a,79b]. The algorithm for reducing combinators is also a proof tactic for theorems in our primitive logic. To prove a putative theorem in our logic, we simply evaluate it. If it evaluates to K, then it is a theorem, and by reversing all the reductions we obtain a proof.

The system used by Turner differs from our sample primitive logic. It does not attempt to reduce all computation to pure combinatory reduction. The combinators are used instead of more traditional methods of passing parameters by maintaining environments. In addition to pure combinators, a combinator graph may include data values from primitive value sets, and primitive operators on such values.

Furthermore, Turner uses more complex combinators than ours. His implementation would not otherwise be sufficiently efficient to be usable for any practical purpose. Even with these combinators and primitive operations combinator implementations of functional languages may be two orders of magnitude less efficient than fully compiled imperative languages.

More recent work on the implementation of combinator reduction systems has shown that the efficiency of implementation can be considerably improved by compiling combinators into machine code [Joh84]. We propose to use combinator graph reduction as an implementation technique for a formal methods development environment (without prejudice to the target execution environment).

In order to achieve reasonable efficiency we will make some adjustments to the primitive combinators to permit an efficient mapping onto the memory of a von-Neumann computer. We will also make provision for the compilation of combinators of arbitrary complexity. This provision will displace the use of built in data types and primitive operations.

We then have a machine which is attempting (and failing) to prove a theorem of our primitive logical formalism. The theorem to be proven is a term (held in a persistent store) part of which is an unbounded structure representing all the data input to the machine (in the manner of a lazy list).

The remainder of the term consists of two main elements. The first is a function which may be regarded either as representing the combined operating system and application development software of the machine or as a derived inference rule. The second may be regarded either as a functional database (as in [Nik85]) containing all the users data, or as a compound proposition expressing the content of the users "knowledge base".

The theorem which the machine is trying to prove is the application of the derived inference rule to the conjunction of the input data with the knowledge base. The theorem proving strategy is essentially reduction of a term of pure combinatory logic to its normal form, (the theorems of our primitive logic are just those terms of pure combinatory logic which are reducible to K).

The hypothesis however has no normal form, and the reduction process results in the generation of an infinite term. The head of this term at any stage in the reduction consists of all the outputs of the system, while the tail represents the knowledge base, the remainder of the input list (new facts), and the operating system (inference rule). All of these are iteratively updated during the evaluation process, so that the changes to the knowledge base are the effects of the commands occurring in the input list, and the output at the head of the term grows as further information is presented to users.

The implementation of our formal system on this engine will in some respects resemble that of LCF and its variants, with the following modifications.

Firstly the primitive logic is type-free, and hence much simpler. Secondly, the same language will be in use both for meta-language and object language, resulting in further economies. Also, as previously noted, derived inference rules will be established after the manner of [Boy81], rather than as proof generation algorithms, this is essential to achieving tolerable efficiency in the proof facilities. We will support extensions to the abstract syntax to match the establishment of abstract data types, and will allow flexibility of concrete syntax as in the Mule system [Nip85].

7. VERIFICATION

There are well known and serious problems in verifying verification systems.

As a result of Gödel's work [God31], we know that a formal foundation system cannot usefully be used to verify itself. We are therefore bound ultimately to accept a formal foundation system which has not itself been formally verified.

Our reductionist approach to foundations is intended in part as a rational response to this situation. We suggest that confidence in our ultimate formal foundations will be maximised in the following ways:

1. The formal system should be as simple as possible.
2. The system should be transparent to our intuitions.
3. Essentially the same foundation should be subjected to theoretical scrutiny and practical exposure over a long period of time.

Our confidence in the consistency of first order axiomatisations of set theory is largely based upon their having survived over a long period of time without having been found inconsistent. We believe that the approach to foundations outlined in this paper satisfies points 1 and 2. We believe also that our formalism is sufficiently flexible to underpin a wide variety of more specialised formal systems and that this will increase its chances of receiving the exposure that will in due time contribute to confidence in its sufficiency and consistency. In fact we are essentially formalising recursive function theory, a subject which has now had some 50 years of scrutiny.

Our foundational reductionist approach results in the step from one logical level of the system to the next being achieved by definition rather than axiomatisation. This converts problems of consistency into logically less severe problems of opacity. This way of building on foundations is guaranteed not to compromise the consistency of the system, but if there are errors in the definitions then the concepts defined will not be the ones intended.

The implementation will be constructed in an analogous way. We therefore expect to implement the core foundation system as a logically secure bootstrap. This results in subsequent levels of development being logically guaranteed not to compromise the consistency or correctness of the implementation.

Implementation of higher levels of the system will also be provably correct against their specifications, but this only begins to be helpful once we have established specification languages which are significantly more perspicuous than our implementation languages.

8. CONCLUSIONS

We have outlined an approach to logical foundations for the formal development of computer systems which we believe when fully developed will offer:

- a) The highest possible levels of assurance of the correctness of systems developed.
- b) High levels of flexibility.
- c) Economy of implementation.

This foundation offers a particular advantage in the exploitation of abstraction, increasingly seen as an important tool for formal development. We offer a foundation within which mathematical concepts, without qualifications relating to cardinality, have denotations.

Considerable further work is necessary before we can be wholly confident that this approach can be made to deliver what we believe it to offer. It is inherent in our approach that once the definition of the formal systems has been carried through in a fully formal way, an inefficient implementation will be obtainable at trivial cost. Provided sufficient care is taken in design, we believe that tolerably efficient implementations will then be relatively inexpensive.

9. REFERENCES

- [Atk85] Atkinson, Malcolm P.; Morrison, Ronald: Types, Bindings and Parameters in a Persistent Environment. In: Persistent and Data Types, Persistent Programming Research Report 16, University of Glasgow, 1985
- [Boy81] Boyer, R.S.; Moore, J.S.: Metafunctions: proving them correct and using them efficiently as new proof procedures. In "The Correctness Problem in Computer Science" (R.S.Boyer and J.S.Moore, eds.). Academic Press, New York 1981.
- [Bur84] Burstall, R.; Lampson, B.: A Kernel Language for Abstract Data Types and Modules. Proc. Int. Symp. on Semantics of Data Types, 1984.
- [Con80] Constable, R.L.: Programs and Types. Proceedings of the 21st Annual Symposium on Foundations of Computer Science. Syracuse, N.Y. 1980.
- [Cur72] Curry, H.B.; Hindley, J.R.; Seldin, J.P.: Combinatory Logic Volume II. North Holland Publishing Company, 1972.
- [God31] Gödel, Kurt: On Completeness and Consistency. In [Hei67].
- [Gor79] Gordon, M.; Milner, R.; Wadsworth, C.: Edinburgh LCF. Springer-Verlag, Lecture Notes in Computer Science, Vol. 78.
- [Hat82] Hatcher, William S.: The Logical Foundations of Mathematics. Pergamon Press 1982.
- [Hei67] van Heijenoort, Jean: From Frege to Gödel, a sourcebook in Mathematical Logic, 1879-1931. Harvard University Press, 1967.
- [Hin72] Hindley, J.R.; Lercher, B.; Seldin, J.P.: Introduction to Combinatory Logic. Cambridge University Press, 1972.
- [Joh84] Johnsson, Thomas: Efficient Compilation of Lazy Evaluation. SIGPLAN Notices Vol.19, No. 6, June 1984.
- [Lam80] Lambek, J.: From lambda-calculus to Cartesian Closed Categories. In: To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism. Edited by J.P.Seldin and J.R.Hindley. Academic Press 1980.
- [Mar75] Martin-Löf P.: An intuitionistic theory of types: predicative part. In Logic Colloquium '73, pp 73-118, North Holland 1975.
- [Mar82] Martin-Löf, Per: Constructive Mathematics and Computer Programming. In Logic, Methodology and Philosophy of Science, VI (Proc. of the 6th Int. Cong., Hanover, 1979), North Holland Publishing Company, Amsterdam (1982).
- [Nik85] Nikhil, R.S.: Functional Databases, Functional Languages. In: Persistent and Data Types, Persistent Programming Research Report 16, University of Glasgow, 1985.

- [Nip85] Nipkow, T.N.: Mule: Persistence and Types in an IPSE. In: Persistent and Data Types, Persistent Programming Research Report 16, University of Glasgow, 1985.
- [Qui63] Quine, W.V.O.: Set Theory and its Logic. Harvard University Press, 1963.
- [Rus03] Russell, B.: The Principles of Mathematics. George Allen & Unwin Ltd., 1903.
- [Rus08] Russell, B.: Mathematical Logic as based on the Theory of Types. American journal of Mathematics 30, 222-262. Also in [Hei67].
- [Sco70] Scott, Dana: Outline of a Mathematical Theory of Computation. Oxford University Computing Laboratory. PRG-2, Nov 1970.
- [Tur79a] Turner, D.A.: Another Algorithm for Bracket Abstraction. Journal of Symbolic Logic, Vol. 44, No. 2, June 1979.
- [Tur79b] Turner, D.A.: A new implementation technique for applicative languages. Software - Practice and Experience, Vol. 9, 31-49 (1979)
- [Tur84] Turner, D.A.: Functional programs as executable specifications. Phil. Trans. R. Soc. Lond. A 312, 363-388 (1984).

Trusting Compilers - A Pragmatic View

Scott Hansohn
Honeywell Secure Computing Technology Center
2855 Anthony Lane So.
Suite 130
St. Anthony, MN 55418
(612) 782-7144
Hansohn@HI-Multics.ARPA

- Life-critical and security-related applications require a high-degree of assurance that the software will behave properly. The use of high-level languages and verification environments automate the software development process, and decrease the probability of human error. However, the "correctness" of the resulting system becomes dependent upon the integrity of the tools.
- Although the use of low-level languages (e.g. assembly) is labor intensive, the software manager is able to use design reviews and code walkthroughs to avoid placing complete trust in any one individual. With high-level languages, reviews take place at the source code level, but gaining assurance that the machine code actually does what the source code says it does is usually an act of blind faith.
- This paper considers some of the issues that must be addressed when selecting a compiler for an application requiring a high degree of assurance.

Keywords: Security, Ada Implementation Issues, Software Assurance, Trusted Systems.

1 Background

For life-critical or security-critical applications, we would like to have some degree of confidence that the application software performs in a known manner. At the start of the program, requirements stating what the system is to do are defined; at the end of the program we have a working system. The question is, "what assurance do we have that the resulting system operates in accordance with its requirements"?

Techniques have been developed for formally specifying what we want a system to do (specification/assertion languages), for specifying how a system is to do it (programming languages), and for proving a correspondence between the "what" and the "how" (verification tools).

Verification techniques are useful for providing some level of assurance that the software engineer has not made any design errors by ensuring that the source

code conforms to its specification. However, the software that actually makes the system work (machine code) is not the software that was verified (source code). Errors in the source-to-machine code translation process can invalidate the proof.

One problem is that "compilers contain bugs" tends to be the rule rather than the exception. There doesn't seem to be any such thing as an "error-free" compiler. A typical mode of operation is: find the bug, fill out a bug report and send it to the vendor, and then find a workaround so that development can continue. Since the engineers using the compiler are rarely intimately familiar with it, when something goes wrong they are left to rely on engineering skills, instinct, and blind luck to isolate the error and collect enough information to write a bug report that is more specific than "It doesn't work". Worse yet, the error could go undetected and end up incorporated in the final system.

Some languages (such as Ada) have compiler validation suites, but even these cannot assure that the compiler will never make any translation errors. There is a possibility that verification techniques could be applied to compilers and other software development tools to prove that they are proof preserving transforms, but this is beyond the current state of the art.

So the implementor is faced with the following dilemma. On the one hand, high level languages can help to reduce life cycle costs by automating much of the code generation process and help to eliminate several types of human errors. On the other hand, it forces a greater dependency upon the software development tools, resulting in less control over the generated code. The tools (and especially the compiler) become black boxes and must be blindly trusted. It is difficult to check the work of these tools. It is difficult to determine whether or not this trust is warranted.

2 Compiler Errors

As mentioned earlier, compilers tend to be complicated programs and frequently contain errors. Some errors may be inconsequential. Some may be inconvenient. Some may be critical. When selecting a compiler, the software engineer must attempt to determine the probability of that compiler containing a critical error - one that could jeopardize lives or compromise information. This risk must be weighed against the potential benefit of using that compiler (lower life cycle cost, better design, etc.).

There are no objective ways of determining what this probability is. In the end it comes down to a judgement call. A validation suite will ensure that one class of errors has been eliminated, and will decrease this probability. Thorough testing by the compiler vendor will also provide more assurance.

Maturity is another criteria which may be used. If the compiler has successfully been used on a variety of projects, it is (usually) reasonable to assume that many of bugs have been worked out. Testimonials can also provide useful

information.

The following sections describe some types of errors which may cause a source code proof to be invalidated.

2.1 Mapping Errors

A compiler is responsible for mapping high level source code functions onto the more primitive instructions of the target machine. The more nebulous the definition of the language and the machine instruction set, the more subject they become to interpretation, and the more likely the chance for an interpretation error.

For example, typically the only documentation available for defining a machine instruction set is a programmer's reference manual for that machine. This sort of documentation tends to be descriptive rather than definitive. It does not define the exact behavior of the machine.

Without an accurate formal semantic definition for both the source language and the machine language, it is impossible to perform any meaningful check of the accuracy of the mapping between the two. In the case of Ada, ongoing efforts to develop a formal semantic definition of the language are an important first step. In the case of the target processor, more emphasis must be placed on the use of formal verification techniques on processor design and implementation. (For more information on these techniques, see [1].)

2.2 Trojan Horses

As Thompson noted, it is impossible to determine exactly what a program is doing by source code inspection [2]. He further suggests that "You can't trust code that you did not totally create yourself". Today it is rare that an application software developer will build a compiler rather than buy one "off the shelf". The application developer must now trust the people working for the company that developed the compiler. It is also necessary to institute stringent controls to protect this compiler from being tampered with. All other tools used for software development must be similarly trusted and protected.

One approach that has been suggested to reduce the probability of "Thompson attacks" is to freeze the software development configuration at the beginning of the program. The reasoning behind this is that this sort of attack requires knowledge of the application program in order to determine exactly where to place the trojan horse to get the desired effect. An obvious problem with freezing the compiler is that it becomes impossible to fix any compiler bugs that are detected during the development effort. So the designers must decide at the beginning of the project whether or not they can live with a particular snapshot of the compiler. Compiler maturity is an obvious consideration when making a "lock in" decision such as this.

2.3 Security Considerations

A good design practice is to keep your critical functions as simple as possible so that it is easier to make sure that they work properly. This practice is applicable to trusted (security critical) functions in an A1 secure computer system. Similarly, it is desirable to keep the tools used to develop critical software as simple as possible to increase the probability that they will function correctly. The more complex the programming language, the more complex the tools and the greater the likelihood of errors.

Considering Ada for the moment, there are many language features that are simply not needed for the implementation of security kernels or trusted software. For example, Anderson [3] has pointed "there exists a quite usable (for the Kernel) subset of Ada that requires virtually no [runtime support libraries]". It seems reasonable to assume that it may be possible to define a subset of the language that not only minimizes the size of the runtime support library, but also minimizes the size of the compiler, and is still useful for implementing highly critical functions. Since it is likely that formal verification techniques will be applicable to small compilers before they are applicable to large ones, this may provide a powerful interim tool.

3 Conclusions

There has been a fair amount of interest in determining the amenability of Ada to formal verification techniques. But little attention has been paid to ways of ensuring that the compiler performs an accurate mapping of a high-level language onto the low-level language of the target machine. The reality of the situation is that it is entirely possible that the compiler or other software development tools could act to invalidate the proofs of the source code.

One technique that has become increasingly popular for automated formal verification is to use the verification tools as proof checkers rather than relying on them to generate the proof directly. Unfortunately, it is not possible to use a compiler as a checker for hand-compiled code. Similarly, checking the code produced by a compiler (especially an optimizing one) is difficult and time consuming. The software engineer is forced to depend upon the tools.

For life or security critical applications, a risk analysis must be performed to determine the trustworthiness of all personnel, tools, and materials involved in the development effort. Tools such as compilers are invaluable for automating much of the software implementation process, but it is difficult (if not impossible) to determine whether or not they are deserving of the trust required to use them for critical applications.

The final decision must be made based on a cost/assurance tradeoff. The potential life-cycle cost reductions which may result from the use of a particular compiler must be estimated. The effect that relying on that compiler will have on the assurance of the correct operation of the final system must be assessed.

These figures must be compared in the context of the particular application to determine whether the benefits outweigh the risks.

It is important to not lose sight of the fact that the use of verification techniques is not an end, but rather a means toward increasing assurance in the correct operation of a system. The correctness of the resulting system depends on other factors as well.

4 References

1. "FM8501: A Verified Microprocessor", Warren A. Hunt, Jr., Institute for Computing Science, The University of Texas at Austin, Technical Report 47, December 1985.
2. "Reflections on Trusting Trust", Ken Thompson, Communications of the ACM, Vol. 27, No. 8, August 1984.
3. "Ada's Suitability for Trusted Computer Systems", Eric R. Anderson, Proceedings of the 1985 Symposium on Security and Privacy, Oakland, Ca., April 22-24, 1985.

TRUSTING COMPILERS -- A Pragmatic View

**Scott Hansohn
Honeywell Security Computing Technology Center**

idea —————→ reality

ENGINEERING TRADEOFFS:

FUNCTIONALITY			
COST			
time			
money			
RISK			
safety			
security			
soundness			
	PYRAMIDS	CATHEDRALS	MOON TRAVEL

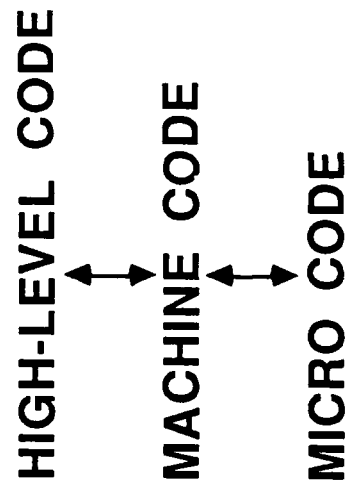
PRIORITIES CHANGE WITH TIME & TECHNOLOGY

STRUCTURED DESIGN (LAYERING)

WAY OF DEALING WITH COMPLEXITY

RISK/COST REDUCTION METHOD

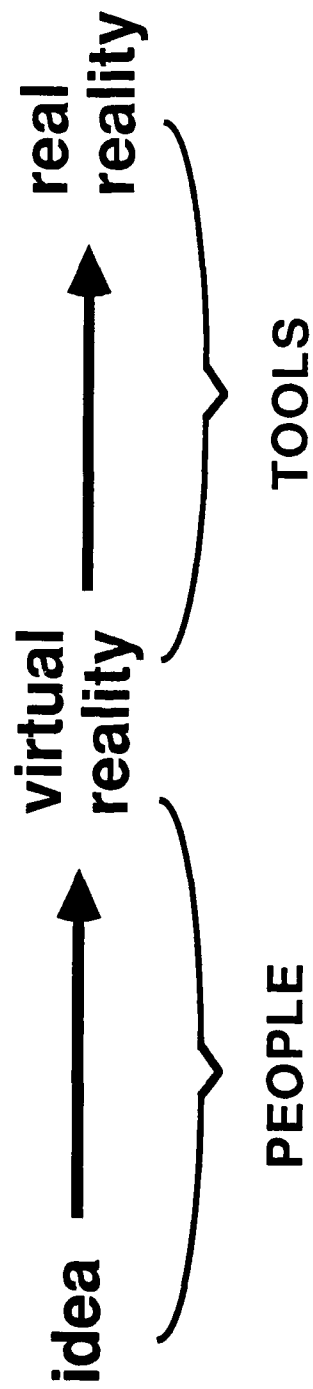
EXAMPLE



FORMAL VERIFICATION

- RISK REDUCTION METHOD
- EXPENSIVE
 - JUSTIFIED FOR SOME APPLICATIONS
- COMPENSATE FOR HUMAN ERROR
- PRIMARY BENEFIT:
 - INCREASED INSIGHT

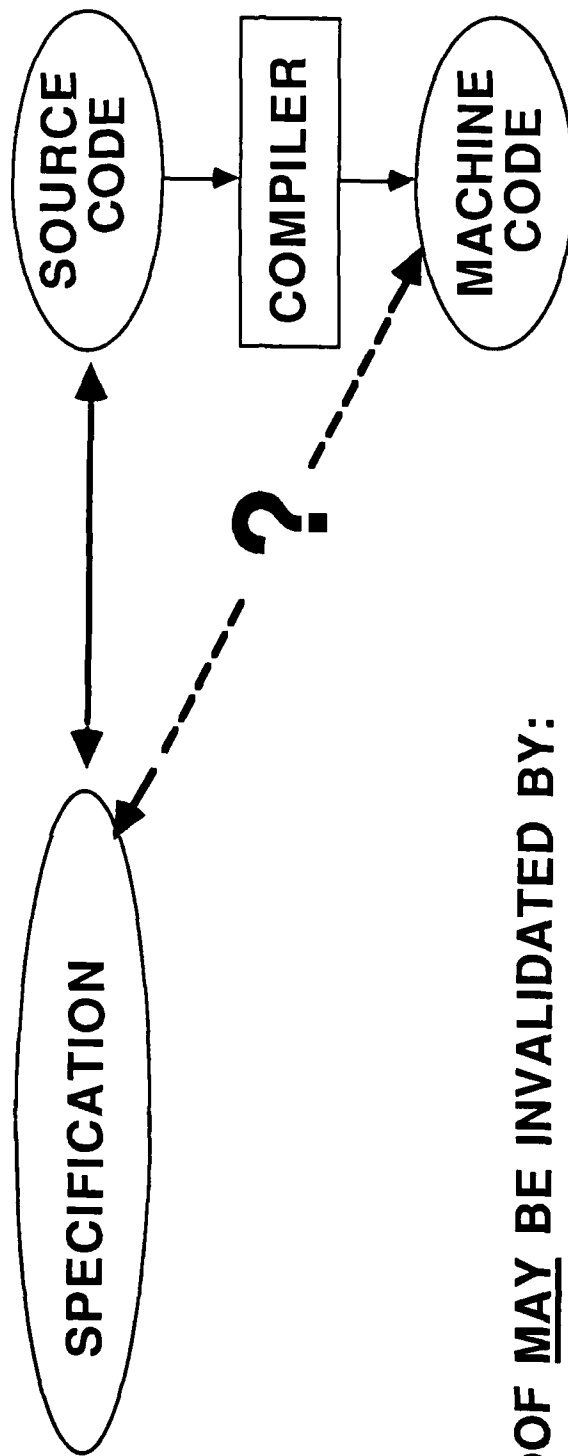
NOT
QED



TOOLS & AUTOMATION

- **COST REDUCTION MEASURE**
- **FUNCTIONALITY INCREASING MEASURE**
(LEVERAGE)

SIMPLIFIED EXAMPLE



PROOF MAY BE INVALIDATED BY:

- **ERRORS (MISTEAKS, MISUNDERSTANDINGS)**
- **TROJAN HORSE**
- **TRAP DOOR**
- **HEREDITARY DEFECT**
- **VIRUS**

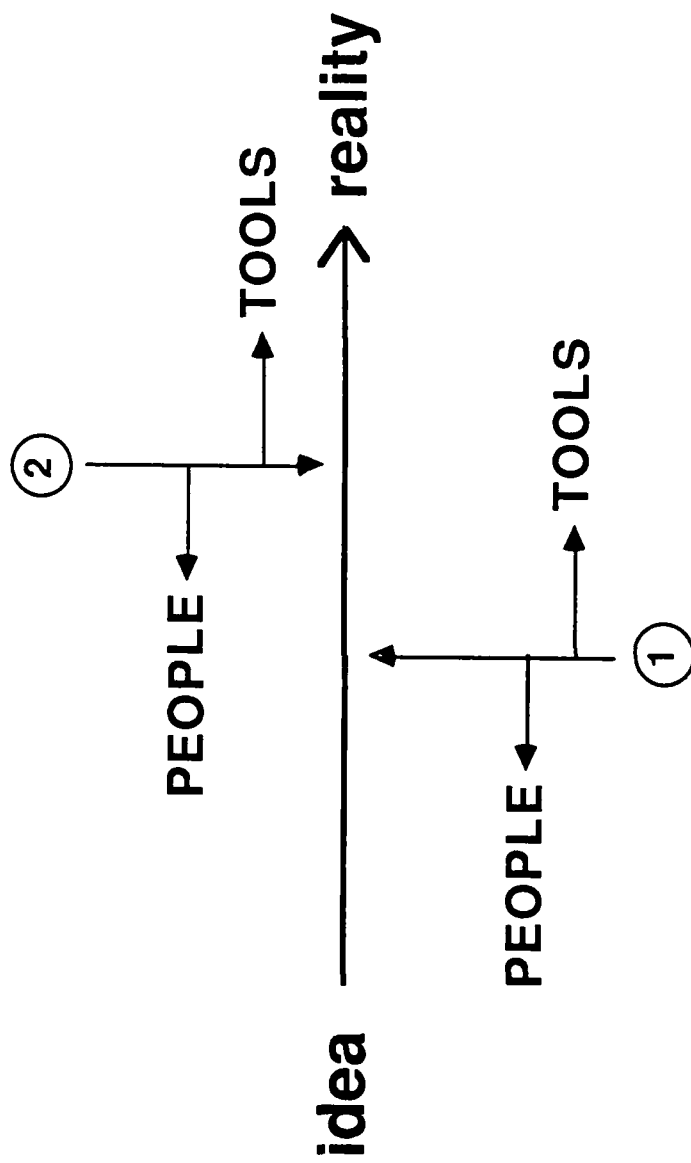
TWO OPPOSING VIEWS

VIEW 1

Automate as much as possible

VIEW 2

Rely upon People, not Untrustworthy Tools



CONCLUSIONS

- ENGINEERING TOOLS INTRODUCE RISKS
- RISKS INCREASE WITH AUTOMATION
- MUST DETERMINE WHETHER BENEFITS OF AUTOMATION OUTWEIGH THE RISKS
- NEED HIGH ASSURANCE IN TOOLS AS WELL AS DEVELOPED SYSTEM

SERVING ALL YOUR PHOBIA NEEDS

SINCE 1954

**"WHEN IT COMES TO SAFETY,
DON'T EXPECT TO SAVE MONEY"**

**-- THE FARMONGER SHOPPE
LAKE WOBEGON, MN**

An Introduction to The Draft Formal Definition of Ada®

Egidio Astesiano

Universita' di Genova
Dipartimento di Matematica
Via L B Alberti 4
I-16132 Genova
Italy

and

Jan Storbanks Pedersen

Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby
Denmark

® Ada is a registered trademark of the U.S. Government
(Ada Joint Program Office)

This work has been partly supported by the CEC MAP project:
"The Draft Formal Definition of ANSI/MIL-STD 1815A Ada".

1. Introduction

This paper provides a short introduction to the project "The Draft Formal Definition of Ada" and the results obtained so far. The first section gives an overview of the project. The second and third sections are more technical and contain introductions to the results obtained in the areas of static semantics and dynamic semantics methodologies. The static semantics formally defines the rules that Ada programs must obey and that correspond to compile time checks. The dynamic semantics defines the run-time behaviour of an Ada program.

2. Project Background and Status

The official definition of Ada, Reference Manual for the Ada Programming Language [RM], contains more than 300 pages of technical English. It is known to contain inconsistencies and ambiguities, some of which can be attributed to the English language.

To resolve inconsistencies and to clarify ambiguities, the AJPO in co-operation with ISO has set up the Language Maintenance Committee (LMC), from which comments are passed to the Ada Board (AJPO) and to ISO/TC/97/SC22/WG9 for approval, before becoming official. But the task of making such vast amount of English text consistent and unambiguous is herculean, and it should be facilitated by a formal definition of Ada (an AdaFD).

In saying this, it should be pointed out that an AdaFD is not to be seen as a replacement of the natural language description, which is able to reach a much larger audience, but as an aid in the precise definition of Ada and in the clarification of the natural language description.

These considerations have led the EEC to sponsor the Draft Formal Definition of ANSI/MIL-STD 1815A Ada, which aims at making a formal definition of Ada as it is described in the reference manual, pointing out inconsistencies and ambiguities; but attempting to resolve these is outside the scope of the current project.

The first part of the project was a test phase in which an underlying model was constructed, reflecting the structures of Ada. A method and a meta-language for the definition were defined with the aim of getting a mathematically well-founded frame which is suitable for the definition of Ada. Finally, a trial formal definition of a subset of Ada was made in order to assure the expressive power of the proposed techniques. The trial definition has now been completed and is documented in [DDC, CRAI 86].

The second part of the project is the full formal definition of Ada. This is the first attempt to give a precise formal definition of a language the size of Ada, but it will be facilitated by the knowledge that was obtained in the test phase. A preliminary study will be made of the feasibility of using the AdaFD to prove certain aspects of the ACVC test suite.

In parallel with the actual development of the AdaFD, a natural language description of the AdaFD will be made, and the formulas will be cross-referenced with their corresponding paragraphs of the reference manual. The natural language description will be in English, but due to the formal definitions' independence of natural languages it would have been easy to derive descriptions in other languages (Danish, Italian, Japanese, French, etc.).

Finally in connection with the scientific work, tools will be developed for writing, checking, and maintaining formulas, for cross-referencing, and for browsing the reference manual.

The project is carried out by:

Dansk Datamatik Center	Denmark (main contractor)
CRAI	Italy (contractor)
I.E.I- C.N.R.	Italy (sub-contractor)

Furthermore, consultants are used from the universities of Pisa (Prof. U. Montanari) and Genoa (Prof. E. Astesiano and Ms. G. Reggio) and the Technical University of Denmark (Prof. D. Bjørner, Dr. H. Bruun and Dr. H. H. Løvengreen).

3. Static Semantics Definition Methodology

Formally expressing the static semantics of a programming language is normally either done using an attribute grammar approach like [J. Uhl et al 82] or using a syntax directed denotational style like [Bjørner & Oest 80]. Within this project, it was felt that the latter style is easier to comprehend for the human reader and easier to relate to the natural language description of rules and conditions stated in the RM. We have, hence, chosen an approach following this style.

3.1 Well-formedness

Following the approach chosen, an Ada static semantics can be considered to be just one large function that given any complete Ada program delivers the boolean value true or false depending on whether the program is statically correct. Since, however, Ada is fairly large (many syntactic productions) and complex (many semantic rules to be obeyed), it is natural to decompose the function into a number of smaller functions, each expressing the well-formedness of a smaller unit.

In the case of Ada with separate compilation, a library etc., the concept of "a program" is not as closely bound to a particular piece of syntax as in most languages. An Ada program is made up by a main program and the units needed from the library. Hence, the concept of a library and the rules for when and how a compilation affects the library becomes a part of the static semantics. This implies that the overall static semantics cannot be just a "large boolean function" but will consist of both the traditional boolean well-formedness formulas applying to compilations, compilation units and parts thereof, using the current library; and formulas describing the effect on the library of compiling a list of compilation units. The latter have, however, not been included in the trial definition.

3.2 Abstract Syntax

The syntactic units themselves can either be described by their concrete syntax or using an abstract syntax. In this definition we have, following the tradition of [Bjørner, Oest 80], used an abstract syntax, called AS1. The abstract syntax is close to the concrete syntax in that all constructs that cannot be distinguished without a semantic analysis are not assumed to be disambiguated in AS1, e.g. one cannot in all cases syntactically distinguish between a function call, an indexed component and a type conversion.

The abstract syntax is hence derived from the concrete syntax in a straightforward manner. Since we are using a number of abstract data types (see section 3.4) and some of these have operations using AS1 constructs as parameters, we have chosen to express AS1 itself as a number of abstract data types - one for each production rule of the grammar; so that each of the abstract data types has a main sort corresponding to a domain normally defined by an abstract syntax. This means that the abstract data types that refer to AS1 constructs need only contain an enrich 'some abstract data type from AS1' + ... by ..., and they will then be able to use the sorts and operations (selection, composition and decomposition).

3.3 Syntax Directed Approach

Letting the decomposition of the formulas closely follow the structure of AS1 is normally referred to as syntax directed, and that approach has been adopted within this definition since it makes the relation between the formulas and the LRM simpler.

3.4 Semantic Information and Data Types

The well-formedness of a syntactic sub-part of an Ada program generally depends on the context in which the sub-part occurs. This means that when one, as stated earlier, wants a decomposition of the formulas, the formulas need information from the context in order to express the (local) well-formedness. This information could of course be provided by having the abstract syntax representation of the whole program (including package STANDARD and all the (predefined) library units) as an additional parameter when expressing the well-formedness of the construct. Even with all that information, it would be difficult to determine for instance the set of visible declarations at the point of the construct whose well-formedness is to be expressed. A better solution is to have a less syntactic and more semantic parameter to the well-formedness formulas.

The above arguments have lead to the introduction of an auxiliary data structure called 'the surroundings'. In order to keep a clear distinction between the formulas defining the well-formedness and those building, maintaining, and extracting information from 'the surroundings', this data structure together with all formulas operating upon it has been made into a data type called SUR. This means that the data type SUR will provide the well-formedness formulas with a number of sorts (descriptors of different kinds) and operations thereon.

Using such a structure containing information relevant at any point of an Ada program in our opinion places those persons defining the structure under an obligation to include only information that is actually necessary and in a form that is as close to the Ada concepts as possible in order to avoid confusion. Furthermore, we adopt an applicative (functional) style for the well-formedness functions so that they will always have 'the surroundings', or information extracted from it, as explicit parameters so that no changes to the structures will be made that are not immediately visible to the reader.

The surroundings itself can be decomposed into smaller data types each handling the description of a particular aspect of Ada.

Overload Resolution

Having introduced the concept of data types makes it natural to use such data types also for describing properties that are not directly part of the surroundings.

One central Ada concept in particular seems to be suited for being handled by such a data types namely overload resolution, in particular the part expressing the conditions under which overloading can be resolved. The data type `EXPR_DESCR` defines the structures (expression descriptors) and operations necessary to handle overload resolution.

Scope and Visibility

One important part of the surroundings is that which describes all declarations in scope at any point of an Ada program and also keeps track of which declarations are actually visible directly or via selection. This is handled by the data type `NAME_VIS`.

Nested Program Structures

Another part of the surroundings (`ENCL_CONSTRUCTS`) is used to describe the nesting of program structures. It is used when expressing that given constructs may only occur within certain composite constructs, e.g. a return statement with an expression may only occur within a function body but not within any inner body of a task, package or generic unit.

Generic Units

In order to handle the information needed to express the parameter matching conditions and the information necessary to construct the new surroundings after generic instantiations, especially the implications on `NAME_VIS` (a new unit is introduced and new local types are created etc.), a data type `GENERICIS`, that is used by `SUR`, has been introduced.

Entity Descriptors

Within the NAME_VIS data type, descriptors for all explicitly or implicitly declared identifiers are present. The individual descriptors are defined by the data type DESCR. The data type offers operations for creating descriptors based on the constituent parts and for extracting the information (later) when needed.

Operations on Types

Each type in Ada has a set of operations that are derivable. They include the basic operations, the predefined operators, the derivable user-defined subprograms and in the case of an enumeration type, also the enumeration literals. To keep track of the derivable operations and operations like assignment, that do not overload subprograms, a data type TYPE_OPERATIONS is introduced.

Types and Subtypes

In order to facilitate type checking, all types are given a unique identification. Also information on the internal structure of composite types, for example the index subtypes of an array type, is associated with a type in the form of a so-called 'type_structure'. Finally, subtypes are characterized by their (base) type and an optional constraint descriptor. All of this is defined in the data type TYPES.

Values

A simple data type called VALUES is introduced to handle the Ada values needed in the static semantics definition for static index constraints, choices etc.

Dependences between Data Types

As mentioned earlier, the data type SUR depends on all the other data types; but also among the other data types dependences exist. Ideally, the data types would constitute a hierarchy with VALUES and ID (from AS1) at the bottom, SUR at the top and all the others topologically sorted (following the enrich relation) in between. This is, however, not possible when reflecting the concepts of Ada. Let us consider the following example:

The data type TYPES describes the structure of all types including in particular task types.

The DESCR data type uses TYPES (primarily the sort Subtype) in order to describe declared entities. Such an entity can for example be a declared variable, a formal parameter or, for this example, an entry in which case the descriptor contains information on its formal part.

These entry descriptors however constitute a natural part of the type structure of a task type which is defined within TYPES.

Hence, the specification of DESCR contains an 'enrich TYPES' and TYPES contains an 'enrich DESCR'. This is, however, only a mutual recursion between the specifications of the data types and the specifications will in the particular cases used in this definition still denote well-defined data types.

Additional Data Types

Most of the above described data types provide several sorts to the outside world. For the final definition, we are considering sub-dividing the data types further so that each of them only exports one or a couple of sorts. This would also make each of the data types smaller and, hence, facilitate getting a full understanding of their contents.

Defining Data Type Operations

Currently, the operations of the data types are described as functions within a particular model of the data type. Splitting the data types into smaller ones, may make it feasible to express the operations in terms of axioms, thus yielding a more abstract description of certain data types, but without affecting the rest of the static semantics.

4. Dynamic Semantics Definition Methodology

4.1 The Problem

The major difficulties of a formal definition of Ada are encountered in the dynamic semantics. The fundamental difficulty is due to the concurrent structure, which is not only far more complicated than that of previous languages, but also deeply mixed and interfering with the sequential aspects of the language. The problem of interference makes for example the semantics of an expression a very complex object, since for evaluating an expression some tasks and procedures can be started, and hence all the facets of semantics have to be taken into account. Not even the evaluation of a variable, since variables are potentially shared, can be done in isolation without considering the overall environment.

However, the problems with Ada are not confined to the interference problem: implementation dependent features, incorrect constructs and erroneous executions, interaction of a program with the external environment (e.g. external flags) are some of the new aspects to be tackled. Because of these problems the previous valuable attempts of a formal definition of Ada (Inria 1982, Dewar et al. 1983, Bjorner & Oest 1980) have failed on some aspects of the dynamic semantics. A completely new approach was required and that has led to use of a new methodology, the SMO LCS methodology (see Astesiano & Reggio 1986 a, 1986 b, Astesiano et al. 1986), which seems to handle most of the above problems satisfactorily.

4.2 The Underlying Methodology: SMO LCS

For the SMO LCS semantics of concurrent languages, the SMO LCS methodology is based on a two-step approach, combining a denotational overall schema with algebraic techniques for the specification of abstract data types, here applied to the specification of concurrent systems, which are formalized following the SMO LCS operational schema.

Essentially the first step connects the abstract syntax to an underlying model for concurrency, formalized in a suitable language for describing processes (behaviours) and their mutual interactions in a concurrent system. This is done by a set of denotational clauses, where in a typically denotational style each well-formed construct of the source language has associated a term in another language.

Altogether the denotational clauses can be seen as defining, inductively on the structure of the abstract syntax, a syntax-directed translation into an intermediate language for representing processes and concurrent systems. The approach goes further. The semantics of the intermediate language is given by the algebraic specification of a concurrent algebra (the second step), representing a concurrent system modelling program executions.

Following the SMO LCS approach for the specification of concurrent systems, a concurrent system is a labelled transition system built from some component subsystems; each subsystem is in turn modelled as a labelled transition system. A state of a concurrent system is modelled as a set of states corresponding to the subsystems plus some global information, the transitions are inferred from the transitions of the component subsystems in three steps: synchronization, parallelism, and monitoring.

- Synchronization defines the transitions representing synchronized actions of sets of subsystems and their effects on the global information.
- Parallelism defines the transitions representing admissible parallel executions of sets of synchronized actions and the compound transformations of the global information (mutual exclusion problems, for example, are handled here).
- Monitoring defines the transitions of the overall system respecting some abstract global constraints (like interleaving, free parallelism, priorities, etc.).

This SMO LCS schema is expressed in an algebraic parameterized way so that every instantiation on the appropriate parameters, defining the information for synchronization, parallelism and monitoring, is an abstract data type. More precisely the definition of a SMO LCS specification of a system is modular, hierarchical and parameterized.

A pragmatic advantage of this parameterized approach is that it is enough to give the user the directions for the specification of the parameters. An appropriate friendly syntax has been developed for encouraging structured and correct specifications. Moreover, some rules are given ensuring the existence of initial models for each specification.

Together with an initial algebra semantics, corresponding to an operational semantics, the SMO LCS approach supports, with explicit linguistic constructs, the definition of an observational semantics, again via a parameterized abstract data type specification, where the parameters correspond to a formalization of the observations. Every instantiation of such schema admits a terminal model, the Concurrent Algebra, in which two states of the concurrent system are equivalent if and only if they satisfy the same observations; moreover, the underlying mathematics guarantees that every subcomponent of the state gets an observational semantics.

Thus the terms of the intermediate language, obtained by the denotational clauses in the first step, can be interpreted in the concurrent algebra. In this way the denotational clauses define a homomorphism from the algebra of the abstract syntax into a semantic algebra, some carriers of which are the carriers of the corresponding sorts in the concurrent algebra.

Some advantages of this approach are:

- in the first step it is brought to evidence what is truly sequential and what is hiddenly concurrent, thus resolving the basic interference between sequential and concurrent features;
- the algebraic technique of the second step permits a high level of modularity and abstraction in the definition of the many structures encountered in a language; moreover, it also allows one to express the dependence of the semantics on some parameters formalizing the implementation dependent features;
- the observational semantics allow one to represent different semantics depending on what we want to observe of a program and to abstract from the details of the description of the transitions of the concurrent system, which on the other hand is needed, if we want to keep a close local correspondence with the usually informal operational approach of a language reference manual;
- since the specification of the concurrent system embodies an operational view, seeing it as a labelled transition system, also an alternative operational approach to the semantics of the concurrent system can be taken: the axioms can be interpreted as defining labelled transition systems, to which we can then associate observational equivalences directly, a' la Milner [Milner 1980, 1983].

4.3 Overall Structure for Dynamic Semantics.

The overall structure, following the SMO LCS methodology, consists of the denotational clauses (1st step), of the concurrent algebra (2nd step) and of auxiliary structures.

Denotational Clauses

We assume as a starting point an abstract syntax, AS2, which is the result of a transformation of AS1. This transformation (e.g. solving overloading, assigning unique static identifiers) is performed in order to make the dynamic semantics clauses simpler, so that they can really deal with dynamic issues.

Then to every syntactic construct a formal clause is associated, using a denotational style.

To a program the clause associates, given some global initial information, the initial state of a concurrent system modelling the program execution. To every other construct, which is then a part of a task (including the main task corresponding to a program), the corresponding clause associates, given some local information, local to that task, a behaviour, i.e. an expression which corresponds to a process, a fragment of the activity of that task.

The association is compositional, inductively on the structure of the construct. A formal clause is split into parts, and an explanation in natural language is given following the splitting closely, so that one can have a quite precise description of the clause even ignoring most of the formalities used in the clause.

Auxiliary Structures

As usual in denotational semantics, some auxiliary structures are needed, in order to define the semantic domains and express the clauses. The novelty here is that the auxiliary structures, due to the overall algebraic structure of SMO LCS, are algebraic specifications, so that we have algebraic specifications, as abstract data types, of local informations, global information and of related substructures, like denotations.

Concurrent Algebra

The concurrent algebra is the specification of an abstract data type corresponding to a concurrent system modelling program execution. For modelling Ada we have two labelled transition systems, the Task Transition System (TTS) and the Program Concurrent System (PCS).

The states of TTS are behaviours, i.e. elements of the intermediate language built inductively as terms from some combinators. Again inductively on the structure of behaviours, the axioms formalizing the transitions rules are given.

The states of PCS are couples; each couple consists of a set of behaviours (states of TTS) and a term representing some global information. The states of PCS are themselves terms of the intermediate language, target of the denotational clauses; the initial state corresponding to a program consists of just one behaviour corresponding to the system task and an initial global information. Then the rules corresponding to creation and termination of tasks generate states with a varying number of behaviours. The transitions of a state s of PCS, say $s = (\{bh_1, \dots, bh_m\}, i)$ are obtained by composing the transitions associated by TTS to $bh_1, \dots, bh_m$, in three steps. First the (synchronous) transitions which correspond to synchronizations of TTS transitions are produced (e.g. rendezvous between tasks); then the transitions which correspond to allowed parallel executions of synchronous transitions are produced (thus resolving problems like e.g. mutual exclusion); finally, general monitoring conditions are imposed (e.g. priorities between tasks) in order to select those parallel transitions which become transitions of PCS.

Semantics

The semantics can be given operationally and observationally.

The operational semantics consists in associating, starting from the transitions, a (possibly infinite) labelled tree to each state of the system and hence also to programs via the corresponding initial states. This gives a meaning to an Ada program in the sense that the tree (modulo some permutations and identifications of subtrees) already represents an abstraction (for the acquainted reader, corresponding to Milner's strong equivalence [Milner 1980]). We say that two states (programs) are operationally equivalent iff their trees coincide.

However, it may happen that two programs modelled by different trees have to be considered equivalent depending on the kind of observations which can be made. Thus the whole specification includes also an observation part, which contains a set of observation functions that check whether a state satisfies an observation. Then two states (programs) are observationally equivalent iff they satisfies the same observations (formally if the observation functions coincide on the two given states). Once we have fixed the observations, it can be proved that under some conditions (see Astesiano & Reggio 1986a, Astesiano et al. 1985) an algebra exists, the Concurrent Algebra, in which two states are equivalent iff they are observationally equivalent; this algebra thus gives a semantics to the states of PCS and to each of the subcomponents of those states, like behaviours (i.e. tasks), terms corresponding to subprograms and so on (formally: the observational equivalence is a congruence). Since the right-hand sides of the denotational clauses, which are terms of the intermediate language, are subcomponents of the states of PCS, they can be interpreted in the Concurrent Algebra. In this way the denotational clauses defines a homomorphism from the algebra of the abstract syntax into a semantic algebra some carriers of which are the carriers of the corresponding sort in the Concurrent Algebra.

References

[Astesiano et al. 1985]

E. Astesiano, G.F. Mascari, G. Reggio, M. Wirsing:

"On the Parameterized Algebraic Specification of Concurrent Systems"

Proc. CAAP '85 - TAPSOFT Conference, Springer Verlag, LNCS Vol. 185, 1985

[Astesiano & Reggio 1986a]

E. Astesiano, G. Reggio:

"An Introduction to the SMO LCS Methodology"

Internal report, 1986.

[Astesiano & Reggio 1986b]

E. Astesiano, G. Reggio:

"A Syntax-directed Approach to the Semantics of Concurrent Languages"

To appear in Proc. '86 IFIP World Congress, (Dublin, Ireland), North Holland, 1986.

[Astesiano et al. 1986]

E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, E. Zucca:

"The Ada Challenge for New Formal Semantic Techniques"

In Proc. of the 1986 Ada International Conference,
Cambridge University Press, 1986.

[Bjørner & Oest 80]

D. Bjørner, O.N. Oest:

"Towards a Formal Description of Ada",
Springer Verlag, LNCS Vol. 98, 1980.

[DDC, CRAI 86]

E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini,
K.W. Hansen, P. Inverardi, E.W. Karlsen, F. Mazzanti, G. Reggio,
J. Storbak Pedersen, E. Zucca:
"Static Semantics of a 'Difficult' Example Ada Subset" and
"Dynamic Semantics of a 'Difficult' Example Ada Subset" (two volumes)

[Dewar et al. 1983]

R. Dewar, R.M. Froelich, G.A. Fisher, P. Kruchten:
"An Executable Semantic Model for Ada"
Ada/Ed Interpreter Ada Project, Courant Institute, NYU, 1983

[INRIA 1982]

INRIA:
"Formal Definition of the Ada Programming Language"
Honeywell Inc, CII Honeywell Bull

[Milner 1980]

R. Milner:
"A Calculus of Communicating Systems"
Springer Verlag, LNCS Vol. 92, 1980

[Milner 1983]

R. Milner:
"Calculi for Synchrony and Asynchrony"
TCS 25, 267-310, 1983

[RM]

U.S. Department of Defense:
"Reference Manual for the Ada Programming Language",
ANSI/MIL-STD 1815A,
January, 1983.

[Uhl et al. 82]

J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann,
G. Winterstein, W. Kirchgaessner:
"An Attribute Grammar for the Semantic Analysis of Ada",
Springer Verlag, LNCS Vol. 139, 1982.

The Draft Formal Definition of Ada

An Introduction

Jan Storbank Pedersen

1. Background

2. Aims of the Project

3. Project Description

4. Technical Introduction

CHILL

- Student project at the Technical University of Denmark
- Master thesis on static semantics (1979)
- Final version accepted as supplement to the CCITT Z.200 recommendation (Oct. 1981)
- DDC has derived a CHILL compiler from the formal definition

Ada

- Several master theses on Ada Semantics (1980)
- Formal Definition of 1980 Ada (81-82) using VDM
- CEC multiannual programme (DDC, Olivetti, CR)
- DDC derived its validated Ada compiler

Lead DDC to:

- "Formal Methods to Industry" project
ESPRIT project: RAISE
- Formal definition of
ANSI/MIL-STD 1815A Ada
MAP project

Aims of the Project

- Highly readable
- Implementation independent
- Unambiguous definition
- Basis for proof system
- Basis for document derivation

Project Description

- 13 man years

- 2 year project (85-86)

- DDC,CRAI

IEI/CNR

Technical University of Denmark

University of Pisa

University of Genoa

- Sponsored by CEC

Two phases:

- A trial definition
- The full definition

1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525

-

Ada
Ambiguous
Grammar

AS1
Data type

AS2
Data type

LRM text

Static
Semantics

Dynamic
Semantics

The Draft Formal Definition of Ada

Trial Definition of the
Static Semantics

Jan Storbank Pedersen

To illustrate:

- **A style of definition**
- **A definition structure**
- **Ways of modelling
specific Ada concepts**

Ada Concepts Covered

- Strong typing**
- Overload resolution**
- Information "hiding"**
- Properties of objects**
- Derived operations**
- Generic units**
- Static expressions**
- Implicit declarations**
- Scope and visibility**

Overall Structure of the Trial Definition

- **Well-formedness formulas**
- **Abstract syntax**
- **Semantic structures**

Underlying Data Types

- **Modular approach**
- **Information hiding**
- **Concept oriented**

Data Types

SUR

EXPR_DESCR

NAME_VIS

ENCL_CONSTRUCTS

GENERICS

DESCR

TYPE_OPERATIONS

TYPES

VALUES

INTG

ID

QUOT

Well-formedness

3.2<1> Objects and Named Numbers

OBJECT_DECL

Concrete Syntax:

```
object_declaration ::=  
  identifier_list : subtype_indication [:= expression];
```

```
identifier_list ::= identifier
```

Abstract Syntax:

```
OBJECT_DECL :: ID × SUBTYPE_INDIC × [EXPR]
```

Comments:

In the subset, identifier lists contain only one identifier. Hence the identifier (alone) appears in the abstract syntax. Furthermore, constant declarations are not part of the subset.

Semantics:

is_wf_Object_decl: Object_decl $\rightarrow$ (Sur $\rightarrow$ Bool)

```

0  is_wf_Object_decl(mk-Object_decl(id,subt_indic,oexpr)) (sur)  $\hat{=}$ 
1    (let decl_sur = introduce_id_in_sur(id)(sur) in
2      is_wf_Subtype_indic(subt_indic)("OBJ-DECL") (decl_sur)
3      ^ (let subtype = subtype_from_Subtype_indic(subt_indic)(decl_sur) in
4          (oexpr  $\neq$  nil)  $\supset$  (wf_Expr_type_match(oexpr)(subtype)(decl_sur)
5                               ^
6                               is_wf_Expr(oexpr,expr_descr_from_Subtype(subtype))
7                               (decl_sur))))

```


- [1] The object identifier is introduced in the surroundings so that potential references to it within the declaration can be detected. [LRM 8.3(22)].
- [2] The subtype indication must denote a subtype that is allowed to appear in the context of an object declaration [LRM 3.2.1(1), LRM 3.3.2(5), LRM 3.6.1(6), LRM 3.8.1(4)].
- [3-7] "If the object declaration includes an assignment compound delimiter followed by an expression, ..., the type of the expression must be that of the object". [LRM 3.2.1(1)].

References:

introduce_id_in_sur [FD ...], *is_wf_subtype_indic* [FD ...],
subtype_from_subtype_indic [FD ...], *wf_expr_type_match* [FD ...],
is_wf_expr [FD ...], *expr_descr_from_subtype* [FD ...]

Users:

is_wf_Basic_decl [FD 3.1<1>]

Visibility

- 'Visible by selection'
- 'Directly visible'
- Hiding
- Homographs

Example of Visibility

package P is

A : INTEGER;

package Q is

B : BOOLEAN;

C : INTEGER;

end Q;

end P;

package body P is

C : INTEGER := A;

-- directly visible

D : BOOLEAN := Q.B;

-- visible by selection

package body Q is

A : BOOLEAN;

-- hiding

E : INTEGER := P.A;

-- visible by selection.

-- expanded name

end Q;

use Q;

F : BOOLEAN := B;

-- directly visible

G : INTEGER := C;

-- directly visible

end P;

Data Type: NAME_VIS

Sorts: NameVis, Usemap

Operations:

- To maintain the visibility
- To extract information related to an identifier.

model NAME_VIS

domains:

NameVis = DirVis | SelVis

DirVis :: locdict: Dict × Dict_constr

SelVis :: Dict

Dict_constr = Dict → Dict

Dict :: Decl_descrs × Encl_units

Encl_units = Id ↗ (Dict | "TOO-MANY")

Usemap = Id ↗ Decl_descrs

declare_id: Id Descr NameVis $\Rightarrow$ NameVis

```
0 declare_id(id,d,vis)  $\triangleq$   
1   new_locals(define_id(id,d),vis)
```

new_locals: Decl_descrs NameVis $\Rightarrow$ NameVis

```
0  new_locals(decl_descrs, mk-DirVis(loc_dict, dc))  $\hat{=}$   
1    (let new_loc_dict = new_decl_descrs(loc_dict, decl_descrs) in  
2      mk-DirVis(new_loc_dict, dc))
```


new_decl_descrs: Dict Decl_descrs $\rightarrow$ Dict

0 *new_decl_descrs*(mk-Dict (dd, enclu), newdd) $\triangleq$
1 mk-Dict (*join_Decl_descrs*(dd, newdd), enclu)

Remarks:

assert: enclu = []

enter_unit: Id NameVis $\Rightarrow$ NameVis

```
0  enter_unit(id, mk-DirVis(loc_dict, dict_constr))  $\hat{=}$ 
1  (let dict_constr_in_entered_unit =
2     $\lambda$  loc_dict_in_unit.
3      (let new_loc_dict = new_encl_unit(loc_dict, id, loc_dict_in_unit)
4        let inherited_dict = dict_constr(new_loc_dict) in
5          override_dicts(inherited_dict, loc_dict_in_unit)) in
6      let entered_unit_dict = mk-Dict(empty_Decl_descrs(), []) in
7      mk-DirVis(entered_unit_dict, dict_constr_in_entered_unit))
```

new_encl_unit: Dict Id Dict $\rightarrow$ Dict

0 *new_encl_unit*(mk-Dict(dd, enclu), id, encl_unit_dict) $\triangleq$
1 mk-Dict(dd, enclu + [id $\mapsto$ encl_unit_dict])

Remarks:

assert: enclu = []

override_dicts: Dict Dict $\rightarrow$ Dict

```
0  override_dicts(mk-Dict(ddl,encl1),mk-Dict(dd2,encl2)) =
1  (let indict2 =  $\lambda$  id.is_defined(id,dd2)  $\vee$  id  $\in$  dom encl2 in
2  let encl = [id  $\rightarrow$  encl1(id) | id  $\in$  dom encl1  $\wedge \neg$  indict2(id)]  $\cup$  encl2
3  + [id  $\rightarrow$  "TOO_MANY" | id  $\in$  dom encl1  $\wedge$  id  $\in$  dom encl2
4  ^ (is_Function_descr(encl1(id))
5  v encl1(id) = "TOO_MANY"
6  v is_Entry_descr(encl1(id)))
7  ^ (is_Function_descr(encl2(id))
8  v encl2(id) = "TOO_MANY"
9  v is_Entry_descr(encl2(id)))] in
10 mk-Dict(override_Decl_descrs(ddl,dd2),encl))
```

591

Remarks:

[3-9] LRM 4.1.3(17)

```

expr_descr_from_Selected_component: Selected_component → (Types Usemap
                                NameVis → Expr_descr)

```

```

0  expr_descr_from_Selected_component
1      (mk-Selected_component(prefix,selector)) (types,usemap,vis) ≡
2      (let edescr = expr_descr_from_Prefix(prefix) (usemap,vis,types) in
3      let (expanded,vis') = check_expanded_Prefix(prefix,usemap,vis) in
4      (expanded ^ ~ is-All(selector) →
5      expr_descr_from_Simple_name(selector,usemap,vis'),
6      is-Simple_name(selector) →
7      select_appropriate_type_component(selector,edescr,types)
8      )
9      (let fedescr = expr_descr_from_Function_call
10         (mk-Function_call(prefix,<>) (types,usemap,vis) in
11         select_appropriate_type_component(selector,fedescr,types))
12      )
13      select_appropriate_Entry(selector,edescr,types),
14      is-All(selector) →
15      select_all(expr_descr_from_Prefix(prefix) (usemap,vis,types),
16      true
17      {}))

```

```

check_expanded_Prefix: Name → (Usemap NameVis → Bool [NameVis])

0  check_expanded_Prefix(name) (usemap, vis) ≡
1  cases name:
2    (mk-Simple_name(id)
3     if is_expanded_vis(id, usemap, vis) →
4     then (true, select_expanded_vis(id, usemap, vis))
5     else (false, nil))
6  mk-Selected_component(prefix, id) →
7  cases check_expanded_Prefix(prefix) (usemap, vis):
8    ((true, expvis) →
9     if is_expanded_vis(id, usemap, expvis)
10    then (true, select_expanded_vis(id, usemap, expvis))
11    else (false, nil),
12    (false, nil) →
13    (false, nil)),
14  true
15  (false, nil)

```

```

is_expanded_vis: Id Usemap NameVis → Bool

0  is_expanded_vis(id, usemap, nvis) ≡
1  (let mk-Dict (dd, encl) = get_vis_dict (usemap, nvis) in
2   id ∈ dom encl
3   ∨ is_package_descr (get_Descr (id, dd) )

```

select_expanded_vis: Id Usemap NameVis $\rightarrow$ NameVis

```
0 select_expanded_vis(id, usemap, nvis)  $\triangleq$   
1   (let mk-Dict (dd, encl) = get_vis_dict(usemap, nvis) in  
2   let selected_dict =  
3     if id  $\in$  dom encl  
4     then encl(id)  
5     else (let visdd = select_package_vis_descrs(get_descr(id, dd)) in  
6       mk-Dict(visdd, [])) in  
7   mk-SelVis(selected_dict))
```


get_vis_dict: Usemap NameVis → Dict

```
0 get_vis_dict(usemap,nvis) =  
1   (let no_use_dict = get_no_use_vis_dict(nvis) in  
2   let use_descr = get_use_vis_dict(usemap,no_use_dict) in  
3   new_decl_descrs(no_use_dict,use_descr))
```

get_no_use_vis_dict: NameVis $\rightarrow$ Dict

```
0  get_no_use_vis_dict(nvis)  $\triangleq$   
1  cases nvis:  
2    (mk-DirVis(locdict,dc)  $\rightarrow$  dc(locdict),  
3    mk-SelVis(d)  $\rightarrow$  d)
```

get_use_vis_dict: Usemap Dict $\rightarrow$ Decl_descrs

```
0  get_use_vis_dict(usemap,dict)  $\hat{=}$   
1    (let mk-Dict(no_use_descr, ) = dict in  
2      get_use_descrs(usemap,no_use_descr))
```

get_vis_descrs: Usemap NameVis $\Rightarrow$ Decl_descrs

```
0  get_vis_descrs(usemap,nvis)  $\hat{=}$   
1  (let mk-Dict (decl_descrs,) = get_vis_dict(usemap,nvis) in  
2  decl_descrs)
```

get_use_descrs: Usemap Decl_descrs $\rightarrow$ Decl_descrs

```
0  get_use_descrs(map,no_use)  $\hat{=}$   
1    join_multiple_Decl_descrs(remove_homographs(  
2      remove_homographs(dds,no_use),dds3) |  
3      ( $\exists$  id  $\in$  dom map) (dds = map(id)) ^  
4      dds3 = join_multiple_Decl_descrs  
5        ({dds2 | ( $\exists$  id  $\in$  dom map\{id})  
6          (dds2 = map(id2)))))
```

THE SMoLCS METHODOLOGY AND ITS APPLICATION TO ADA FD - DYNAMIC SEMANTICS

EGIDIO ASTESIANO

Dept. of Mathematics - University of Genova
CRAI - AdaFD Genova Group

CEC-MAP Project
The Draft Formal Definition of
ANSI/MIL-STD 1815A ADA®

DDC - Prime Contractor CRAI - Contractor
IEI - Subcontractor
Universities of Genova and Pisa - Consultants

FORMAL DEFINITION OF THE SEMANTICS OF PROGRAMMING LANGUAGES

The Ada Challenge :

- COMPLICATED CONCURRENT STRUCTURE
- "SEQUENTIAL" SYNTAX HIDING THE CONCURRENT STRUCTURE
- INTERFERENCE BETWEEN SEQUENTIAL AND CONCURRENT ASPECTS
- OTHER HOT ISSUES:
 - Erroneous Executions & Incorrect Constructs
 - Implementation Dependent Properties
 - Time
 - Partially Defined Interactions with external Environment
- SIZE PROBLEMS OF THE FD
 - Readability
 - Use
 - Maintenance

Overall Structure of the Ada FD Dynamic Semantics

Two-Steps SMoLCS approach

1st Step (Definition of the Denotational Clauses):

to each Ada program is associated a

program-state-expression

(a term in an "intermediate language")

(an expression of type STATE)

2nd Step (Definition of the Concurrent Algebra):

the actual meaning

of that *program-state-expression*

(the semantics of the term)

(the value of the expression)

is defined

COMPOSITIONALITY

First step

The *program-state-expression* associated to a program is obtained by the composition of the sub-expressions associated to its constructs

Second step

A value is provided not only to the *program-state-expressions*, but also to every constituting sub-expression

STRUCTURE OF *program-state-expressions*

({ *behaviour-expression* } , Initial-Global-Inf)



sub-expression associated to the
program by the *exec-System-Task*
clause.



constant

- *behaviour-expression* is an expression of type BEHAVIOUR
- Example of *exec-Program* clause:

***exec-Program*: PROGRAM -> STATE**

***exec-Program* (prog) -**

({ *exec-System-Task* (prog) } , Initial-Global-Inf)

- Example of *exec-System-Task* clause:
(for a program constituted by a single subprogram)

exec-System-Task (procedure P is ... end P) =

def li - *elab-Package-Standard*

in **def** li' = *elab-Dcl*(procedure P is ... end P) li

in *exec-Main-Program*(P) li'

where:

elab-Package-Standard: -> BEHAVIOUR

elab-Dcl: DECL -> LOCAL_INF -> BEHAVIOUR

exec-Main-Program: ID -> LOCAL_INF -> BEHAVIOUR

...

exec-Stmt: STMT -> LOCAL_INF -> BEHAVIOUR

eval-Expr: EXPR -> LOCAL_INF -> BEHAVIOUR

...

DYNAMIC SEMANTICS

OVERALL STRUCTURE OF THE FORMAL DEFINITION

DENOTATIONAL CLAUSES (1st step)

Principles

- compositional translation into an intermediate language;
- denotational inductive style (semantics as homomorphism);
- local correspondence with the Language Reference Manual

Structure

- correspondence with chapters and sections of LRM;
- for each clause
 - concrete and abstract syntax;
 - formal clause;
 - line-by-line natural language explanation (with detailed quotations from LRM);
 - technical remarks;
 - complete cross reference with other parts of the Formal Definition.

CONCURRENT ALGEBRA (2nd step)

Principles

static structures as abstract data types.
transition system as an instantiation of a
parameterized abstract data type;
observational semantics given by a Concurrent
Algebra satisfying some observational constraints

Structure

- behaviour part (Task Transition System):
- global information;
- synchronization; (Program Concurrent System)
- parallelism;
- monitoring;

semantics

In general every part can consist of a set of algebraic specifications modelling structures (e.g. behaviours, global information, action flags) and/or a set of axioms defining transitions (behaviour transitions, synchronous transitions...).

ASSIGNMENT STATEMENT CLAUSE

exec-Unlabelled-Stmt : UNLABELLED-STMT ->

LOCAL-INF -> BEHAVIOUR

exec-Unlabelled-Stmt(name := expr)li =

```
0  let task = Get_Task(li) in
1  def (left-val,type-den) = eval-Name(name)li
2  and v = eval-Expr(expr)li
3  in def v' = make-Subtp-Conv(v,type-den)li in
4      choose UPDATE-STORAGE(left-val,v',task) Δ
5      skip
6      or ERR-UPDATE(left-val,task) Δ
7      start-erroneous-execution
8      or MAKE-UNDEFINED(left-val,task) Δ
9      complete-abnormally(li)
```

ASSIGNMENT STATEMENT NATURAL LANGUAGE LINE-BY-LINE EXPLANATION

The execution of an assignment consists of:

- 1 evaluating the variable name
- 2 and the expression, in some order which is not defined by the language;
- 3 then the needed checks and subtype conversions are performed;
- 4 finally, either the variable is updated with the value of the expression
- 5 and the execution continues,
- 6 or, if an assumption on shared variables is violated,
- 7 then an erroneous execution starts,
- 8 or, if the task is abnormal, then the value of the variable can become undefined as effect of
- 9 the abnormal completion of the task

ASSIGNMENT STATEMENT REMARKS

Get_Task is an operation on the local information which returns the name of the executing task

The functions *eval-Name* and *eval-Expr* give a behaviour which returns respectively the denotation associated to the given name and the value of the given expression.

The function *make-Subtp-Conv* gives a behaviour which converts the given value to the given subtype, making the needed checks.

The functions *start-erroneous-execution* and *complete-abnormally* give a behaviour which models respectively the start of an erroneous execution and the abnormal completion of a task.

UPDATE-STORAGE, **ERR-UPDATE** and **MAKE-UNDEFINED** are three actions which model the fact that the task respectively updates the given left value with the given value in the storage, attempts erroneously to update a shared variable and updates the given left value with an undefined value (in the case of an abnormal completion).

SYNCHRONIZATION

$bh_1 \underline{\text{QUEUED-CALL}(eid, called, par-ass, pr, calling)} \rightarrow bh_1'$

$bh_2 \underline{\text{ACCEPT}(eid, called, p-ass, pr)} \rightarrow bh_2'$

$((bh_1, bh_2), i) \underline{\text{RENDEZVOUS}(called, pr)} \rightarrow ((bh_1', bh_2'), i')$

cond: $\text{Is\_First\_Of\_Queue}(calling, called, eid, i) \ \&$
 $\text{Is\_Not\_Abnormal}(calling, i)$

transf: $i' = \text{Make\_In\_Rendezvous}(calling, i)$

SYNCHRONIZATION COMMENT

The intuitive meaning of the given rule is that the two given behaviour transitions (labelled respectively by **QUEUED-CALL** and **ACCEPT**) can synchronize under the condition specified in the "**cond:...**" part, producing a synchronous transition (labelled by **RENDEZVOUS**) which changes the global information as specified in the "**transf:...**" part.

The condition on the global information is that the calling task is the first in the queue associated to the entry eid of the called task and that the calling task is not abnormal.

The transformation of the global information consists in removing the calling task from the queue and recording that it is suspended in a rendezvous.

OPERATIONAL SEMANTIC

Operational meaning

program-state-expression:

interpretation as an initial state of the
Program Concurrent System

(hence as the labelled execution tree starting
from that initial state)

behaviour-expression:

interpretation as a state of the
Task Transition System

(hence by the labelled execution tree starting
from that state)

Methodological comment:

The operational meaning of states and behaviours
is the main guide during the definition of the
denotational clauses, and gives an immediate
intuition to the reader of the first step of the FD.

OBSERVATIONAL SEMANTICS

- Depending on what we want to observe some observations are defined on the states of the Program Concurrent System
(hence on the labelled execution trees starting from those states)
- The observational value of any *program-state-expression* is defined by its interpretation in the Concurrent Algebra.
- Two *program-state-expressions* have the same value in the Concurrent Algebra if and only if their operational values (as states of the Program Concurrent System) satisfy the same observations.

- The observational value of any *behaviour-expression* (and of any other sub-expression) is defined by its interpretation in the Concurrent Algebra.
- Two *behaviour-expressions* (or any other sub-expression) $bh1$, $bh2$ have the same value in the Concurrent Algebra if and only if for every state context $s[x]$, the two states $s[bh1]$, $s[bh2]$ of the Program Concurrent System satisfy the same observations.

Observational Semantics (Explicit characterization)

Proposed Schema

Basis define programs (and tasks, procedures,...)
as labelled trees modulo strong equivalence
(i.e. unordered branching and identification of
equivalent sons)

(an abstraction of operational/initial algebra semantics)

Explicit Equivalence

- 1) define an appropriate equivalence on program trees
- 2) define an appropriate set of formal contexts
(a concurrent system formalizing the external
environment)
- 3) define, depending also on the answers to a series of
formal questions about LRM, what should be an
observational semantics expressing the result
of a program in a context; presumably the result
should be parameterized on some equivalences
- 4) show that the equivalence in 3 coincides with the
one in 1, just showing that two programs different
for 1 have a distinguishing context formalized as
in 2 and 3.

UNDERLYING CONCURRENT MODEL

Overall Structure : " flat structure"

The activity of a program is modelled
as a concurrent system

(Program Concurrent System - PCS)

Each component subsystem models the
activity of a single task

(Task Transition System = TTS)

Motivations for the flat structure:

Program execution driven by several different
structures :

ENVIRONMENT, DEPENDENCES, DYNAMIC CONTROL
FLOW.

No one of them evidently the most important

The flat model results as the most balanced,
allowing to model all the Ada aspects without
any particular effort

PROGRAM CONCURRENT SYSTEM

Is a labelled transition system modelling
all the possible program executions

Labels identify the interactions of the program
with the external environment

- **Interactions with external files**
- **Dependences from the value of some
global time**

A state of the PCS defined from the states of the subsystems corresponding to the executing task, plus a global information

Global Information =

information needed by more than one task which is not exchanged by means of synchronizations

- GLOBAL ENVIRONMENT
- STORAGE
- TASKING INFORMATION
- DEPENDENCES INFORMATION
- FILE INFORMATION
- OTHER INFORMATION ON
IMPL. DEP. ASPECTS

TASK TRANSITION SYSTEM

Is a labelled transition System modelling
the activity of a task in isolation

Labels identify the interactions of the task
with the rest of the program :

- **access to the information shared
among tasks (GLOBAL-ENV, STORAGE,...)**
- **synchronizations with other tasks
(RENDEZVOUS,...)**
- **interactions with the environment
external to the program
(I/O ACTIONS, ...)**

GLOBAL ENVIRONMENT

The environment in which a task is executed is not "stable" (i.e. the denotation of some of the program units it can use can be concurrently "completed")

The **Global Environment** is a global structure recording the **denotation** of **all** the Ada entities **declared during the program execution**

(Intuitively based on a map
from unique-dynamic-entity-identifications
to denotations)

The resolution of **text-identifiers** (i.e. the detection of the unique-dynamic-entity-identification associated to an identifier appearing in the text at any point of the execution) is performed **locally** by each task using a private (dynamic) structure called **LOCAL ENVIRONMENT**

(Intuitively based on a map
from text-identifiers
to unique-dynamic-entity-identifications)

STORAGE

Objects created by a task are not
private to the task

(because of nesting and parameter passing
by reference)

The **Storage** records the values of **all** the **objects**
created during the program execution

To each **object** is associated in the **Storage** a
"sharing information" allowing to detect
erroneous accesses to shared variables

TASKING INFORMATION

Records the information about the status of the entries of the task (queues) and the information about the status of the tasks

DEPENDENCES INFORMATION

Records the dependences structure among masters
Records the information about the master status

FILE INFORMATION

Records the status of file objects (open, ...) and the information on the associated external file (name, ...)

IMPLEMENTATION DEPENDENT INFORMATION

Records the needed structures for the support of implementation dependent operations
(e.g. the 'SIZE attribute for objects and types)

Composing TTS into PCS

SYNCHRONIZATION

- Defining the "minimal program actions"

Examples:

- Any truly sequential task action defines a corresponding synchronous action (e.g. `ADD_DENOTATION`)
- The start and the end of a rendezvous define a synchronous action involving the caller and the called task (e.g. `CALL & ACCEPT`)
- Normal completion of a task defines a synchronous action together with all the tasks suspended in the queues of the completing task
(e.g. `COMPLETE-TASK`
 `& QUEUED-FAILURE`
 `& ...`
 `& QUEUED-FAILURE`)
- ...

- Defining the **conditions** on the Global Information allowing those "minimal program actions"

Example:

- A task can complete abnormally only if it is abnormal
(COMPLETE-ABNORMALLY & ... & ... & QUEUED-FAILURE)
- Defining the **transformation** of the Global Information involved by those "minimal program actions"

Example:

- The final effect of the elaboration of a declaration is the definition of a new denotation in the global environment
(ADD-DENOTATION)
- Defining the possible **interactions** with the rest of the environment

Example:

- A write action defines an interaction with the environment external to the program

Composing TTS into PCS

PARALLEL COMPOSITION

- **Mutual Exclusions** among the previous "minimal program actions"

Examples

- a task cannot enter into a queue while the called task is completing
 - to different tasks concurrently creating new objects are returned different left values.
-
- **Compound effect on the Global Information** of a group of not mutually exclusive "minimal program actions"
-
- #### General Rule:
- The resulting effect on the Global Information of the execution of a group of not mutually exclusive "minimal program actions" is as if they would have occurred in a sequential order

Composing TTS into PCS

MONITORING

- Defining the allowed (or forced) degree of parallelism (effect of priorities)

General Rule:

- Any group of eligible tasks are allowed to execute
 - If a delay has expired, the suspended task must be made aware of that
 - If several rendezvous are possible for the same task, then that one with the highest priority is chosen
- Interactions with the external environment

Example

- I/O action

SMoLCS
METHODOLOGY OF SPECIFICATION

EGIDIO ASTESIANO - GIANNA REGGIO

University of Genova - Italy

in cooperation with

MARTIN WIRSING

University of Passau - W.Germany

APPLICATION TO ADA FD - CEC-MAP project

**C. BENDIX NIELSEN, N. BOTTA, E. W. KARLSEN,
J. STORBANK PEDERSEN (DDC-Denmark)**

**A. GIOVINI, F. MAZZANTI, G. REGGIO, E. ZUCCA
(CRAI - Genova group, Univ. of Genova, Italy)**

A. FANTECHI, P. INVERARDI (IEI - Italy)

AN INTEGRATED APPROACH

SPECIFICATION OF CONCURRENT SYSTEMS

**FORMAL SEMANTICS OF CONCURRENT
LANGUAGES**

METALANGUAGE

(TOOLS)

AIM

Specification

Techniques

Abstraction mechanisms

TARGET

Large (Multilevel) Systems.

*Modularity/Hierarchy
Parameterization*

Languages with inteference of sequential
and concurrent features

2 steps approach

- *connecting syntax to an
underlying concurrent model*
- *making concurrency explicit*

SPECIFICATION OF CONCURRENT SYSTEMS

Concurrent system modelled as a
labelled (flagged) transition
system

Specification of a labelled
transition system as an abstract
data type

Specification of a concurrent
system as instantiation of a
parameterized abstract data type

Parameters → SMoLCS → Concurrent system

LABELLED TRANSITION SYSTEMS

STATES $\{ s, s', \dots, s1, s2, \dots \}$

FLAGS $\{ f, f', \dots, f1, f2, \dots \}$

TRANSITIONS triples (s, f, s')

usually written as $s \xrightarrow{f} s'$

and axiomatized as:

$$\{ s \xrightarrow{f} s' = \text{true} \}$$

Algebraic Specification of Transition Systems

- Abstract data types STATE, FLAG

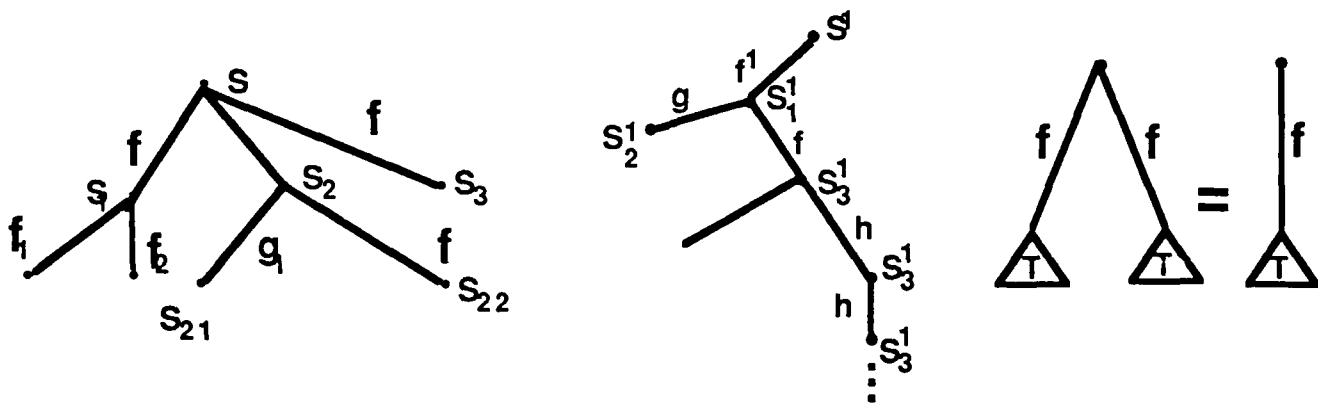
- Transitions by axioms of the form

$$\text{cond} \supset s \xrightarrow{f} s' = \text{true}$$

(universally quantified on the state or flag variables,
or their components)

Labelled trees

associated to labelled transition systems



- to each state S
an associated labelled (possibly infinite) tree

Meaning of a label (flag) in $S \xrightarrow{f} S'$

- f represents (part of) the environment in which that action can take place

$$S \xrightarrow{\text{call } T, E} S'$$

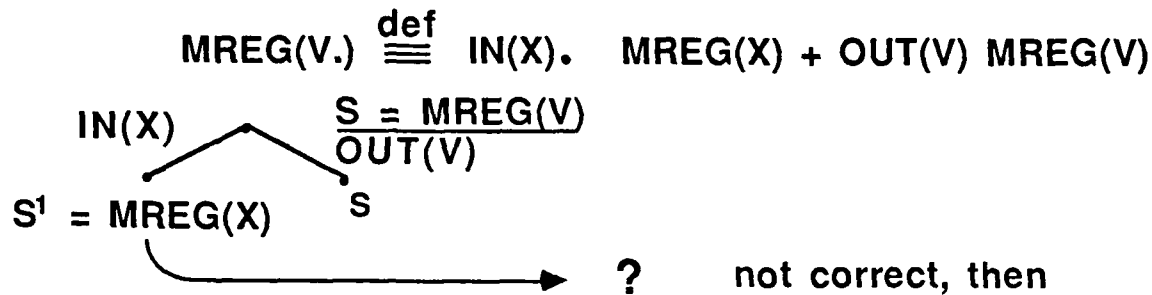
$$S \xrightarrow{\text{read } I} S'$$

f can be seen as

"what can be observed of the system from the external environment in the state S "

Where does the branching come from?

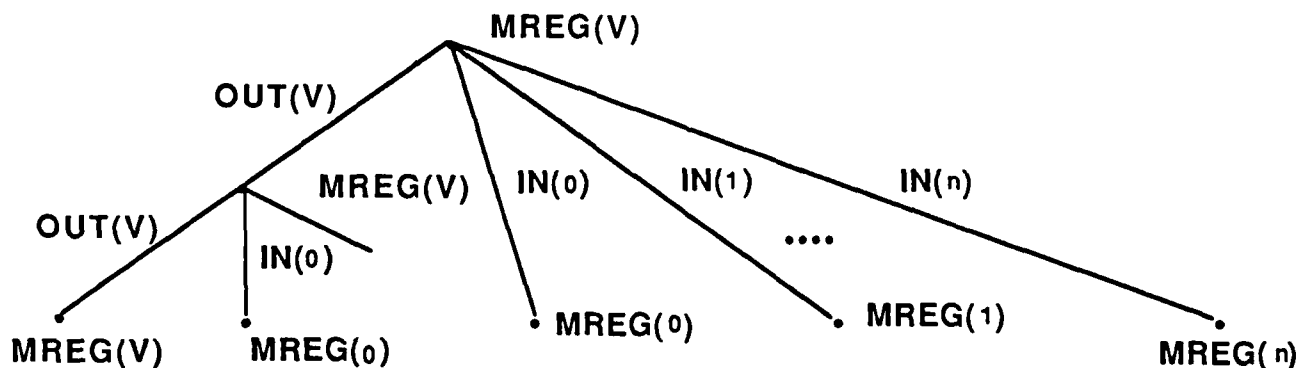
- intrinsic multcapabilities of actions



- parameterized capability
consider X as a parameter

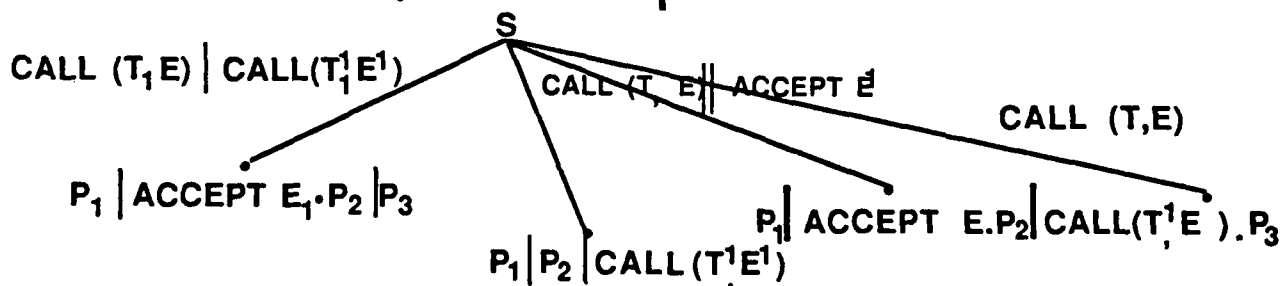
then

$$\text{MREG}(V) \stackrel{\text{def}}{=} \sum_{X=N} \text{IN}(X) \cdot \text{MREG}(X) + \text{OUT}(V) \cdot \text{MREG}(V)$$



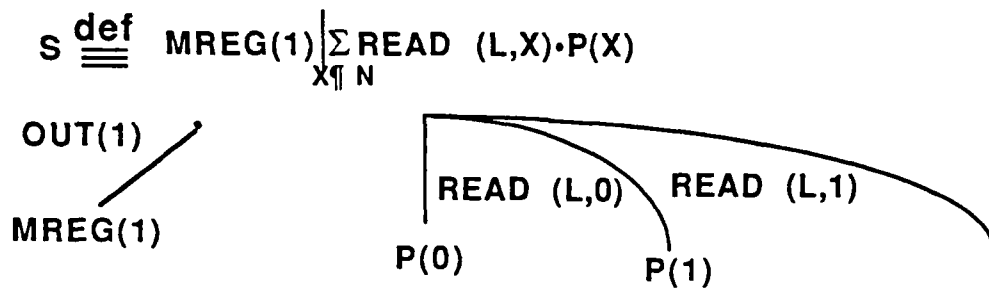
nondeterminism from parallelism

$$\text{CALL}(T_1, E) \cdot P_1 \mid \text{ACCEPT } E \cdot P_2 \mid \text{CALL}(T, E) \cdot P_3 \stackrel{\text{def}}{=} S$$



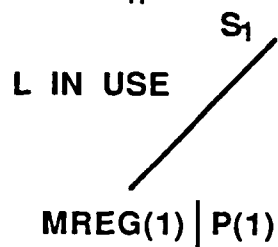
COOPERATION BY LABELLED TREES

$L(ocation) = MREG$



RESULT OF COOPERATION

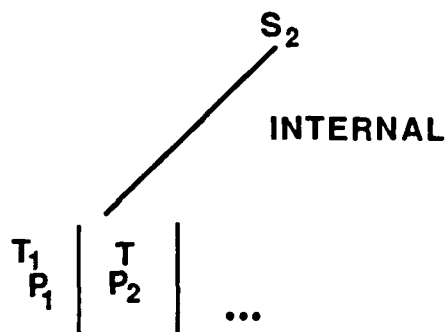
$OUT(1) \parallel \text{READ}(L,1) = L \text{ IN USE}$



$$S_2 \stackrel{\text{def}}{=} \text{CALL}(T_1, E).P_1 \mid \text{ACCEPT } E.P_2$$

$\text{CALL}(T_1, E) \parallel \text{ACCEPT } E$

$T_1 \quad T$



SMoLCS OPERATIONAL MODEL

BASIC SCHEMA

CONCURRENT SYSTEM as a labelled transition system constructed from some component subsystems; each subsystem modelled itself as a labelled transition system.

STATE of a concurrent system:.

- a set of states (of the component subsystems)
- some global information

$\langle s_1 / s_2 / \dots / s_n , inf \rangle$

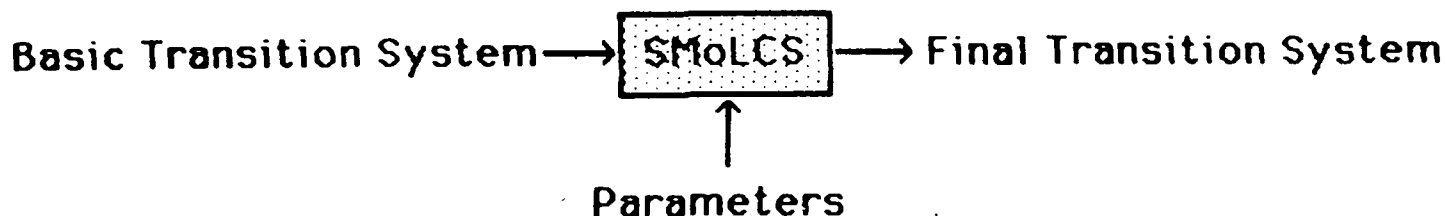
TRANSITIONS inferred from the transitions of the component subsystems in three steps:

SYNCHRONIZATION

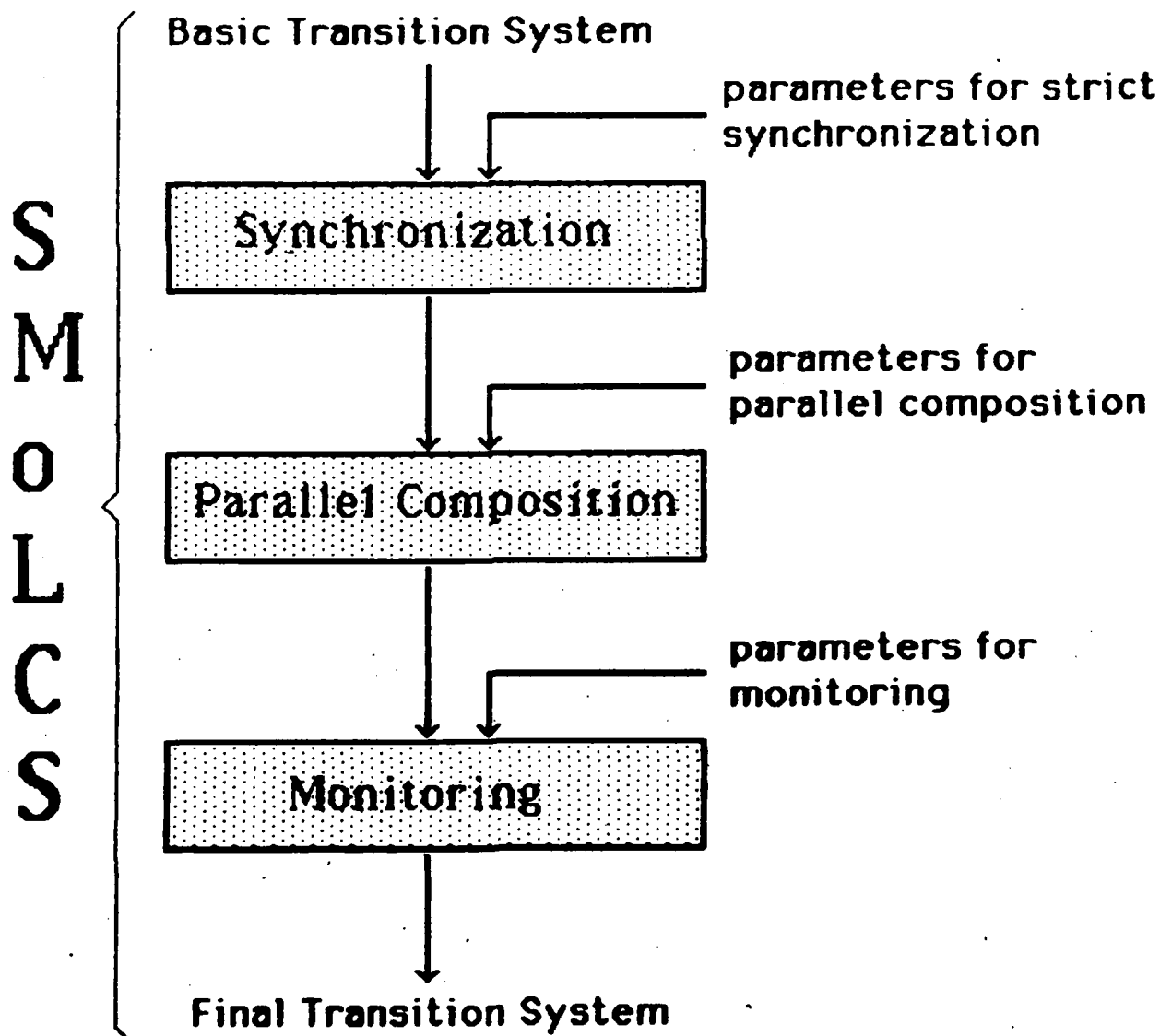
PARALLELISM

MONITORING

PARAMETERIZED SCHEMA



Three Steps Composition



SYNCHRONIZATION

Transitions representing synchronized actions of set of processes and their effect on global information.

PARALLELISM

Transitions representing admissible parallel executions of sets of synchronized actions and the compound transformation of global information (mutual exclusion handled here).

MONITORING

Global abstract scheduling strategies imposed on the system.
(e.g. interleaving, free parallel, priorities, ...)

SYNCHRONIZATION SCHEMA

Starting from

BTS basic transition system with states bs and
 arrow $\xrightarrow{bf}$

Compose a new system **STS** with:

- **states** $\langle bs_1 | \dots | bs_n, inf \rangle$, where inf is defined in **INF**
- **transitions**, with flags sf defined in **SFLAG**, defined by an axiom schema

$$Iss(sf, bf_1 | \dots | bf_n, i) = \text{true} \wedge$$

$$\left(\bigwedge_{1 \leq j \leq n} bs_j \xrightarrow{bf_j} bs'_j \right) \wedge Sit(sf, i, i') = \text{true} \supset$$

$$(bs_1 | \dots | bs_n, i) \xrightarrow{sf} (bs'_1 | \dots | bs'_n, i')$$

where the operations Iss , Sit are defined by the axioms E_{SYNC}

STS as a function

$$STS : BTS, INF, SFLAG, E_{SYNC} \longrightarrow STS$$

BASIC SYSTEM

$$\underbrace{\text{WRITE}(l,v) \Delta bs}_{bs'} \text{---} \text{WRITE}(l,v) \text{---} \rightarrow bs''$$

SYNC-SYST

$$bs' \text{---} \text{WRITE}(l,v) \text{---} \rightarrow bs''$$

$$bs' \text{---} \text{WRITE}(l,v) \text{---} \rightarrow bs''$$

cond: *NOCOND*

transf: $\text{stg}[v/1]$

$$bs_1 \text{---} \text{SEND}(ch,v) \text{---} \rightarrow bs_1' \wedge bs_2 \text{---} \text{REC}(ch,v) \text{---} \rightarrow bs_2'$$

$$bs_1 | bs_2 \text{---} \text{TAU} \text{---} \rightarrow bs_1' | bs_2'$$

cond: *NOCOND*

transf: *NOTRANSF*

PAR-SYST

$$\begin{array}{l} \langle \text{bms}_1, \text{stg} \rangle \xrightarrow{a} \langle \text{bms}_1', \text{stg}' \rangle \wedge \\ \langle \text{bms}_2, \text{stg} \rangle \xrightarrow{\text{TAU}} \langle \text{bms}_2', \text{stg}' \rangle \end{array}$$

$$\langle \text{bms}_1 | \text{bms}_2, \text{stg} \rangle \xrightarrow{a // \text{TAU}} \langle \text{bms}_1' | \text{bms}_2', \text{stg}' \rangle$$

$$\begin{array}{l} \langle \text{bms}_1, \text{stg} \rangle \xrightarrow{a} \langle \text{bms}_1', \text{stg}' \rangle \wedge \\ \langle \text{bms}_2, \text{stg} \rangle \xrightarrow{\text{WRITE}(l, v)} \langle \text{bms}_2', \text{stg}' \rangle \wedge \\ \text{is-updating}(l, a) = \text{false} \end{array}$$

$$\begin{array}{l} \langle \text{bms}_1 | \text{bms}_2, \text{stg} \rangle \xrightarrow{a // \text{WRITE}(l, v)} \\ \langle \text{bms}_1' | \text{bms}_2', \text{stg}' [v/l] \rangle \end{array}$$

SYST

$$\langle \text{bms}_1, \text{stg} \rangle \xrightarrow{a} \langle \text{bms}_1', \text{stg}' \rangle$$

$$\langle \text{bms} | \text{bms}_1, \text{stg} \rangle \xrightarrow{a} \langle \text{bms} | \text{bms}_1', \text{stg}' \rangle$$

PROBLEM

SEMANTICS OF LANGUAGES WITH

- STRONG INTERFERENCE BETWEEN SEQUENTIAL AND CONCURRENT FEATURES

- COMPLEX STRUCTURING

TYPICAL EXAMPLE ADA:

- Syntax devised for static checks not corresponding to underlying concurrent model
- Pseudo-sequential constructs:
(declarations, expressions,
assignments, procedures)
involving possibly interactions of
tasks (communications,
abortions, shared variables)
- Program as a collection of modules
- Semantics parameterized on
implementation dependent
features

AIM

SEMANTIC SPECIFICATION METHODOLOGY SATISFYING SOME REASONABLE REQUIREMENTS:

- SYNTAX-DIRECTED/COMPOSITIONAL
/DENOTATIONAL
 - Meaning of a construct depending only on the meanings of its components
 - formally a homomorphism from (abstract) syntax algebra to a semantic algebra
- DENOTATIONAL STYLE
 - close to functional denotational style on pseudo-sequential constructs
 - interpreted as functional denotational semantics on purely sequential programs
- COMPOSITION OF ABSTRACT SPECIFICATIONS
 - static structures specification
(storage, environment, state)
 - specification of a concurrent
underlying model close to the level of the language
(i.e. not a translation into a low level language)

PROPOSED APPROACH

- COMBINATION OF
DENOTATIONAL
OPERATIONAL
ALGEBRAIC TECHNIQUES
- OVERALL STRUCTURE: DENOTATIONAL
IN TWO STEPS
 - 1ST STEP DENOTATIONAL CLAUSES
 - SEMANTIC DOMAINS AND FUNCTIONS
 - CLAUSES (LOOKING LIKE FUNCTIONAL
DENOTATIONAL CLAUSES ON
PSEUDO-SEQUENTIAL CONSTRUCTS)
 - 2ND STEP CONCURRENT ALGEBRA CSEM
 - CONCURRENT SYSTEM SYST REPRESENTING
PROGRAM EXECUTIONS
 - OBSERVATIONAL SEMANTICS FOR SYST
REPRESENTED BY CSEM
 - LINK BETWEEN THE TWO STEPS
THE CARRIERS OF THE CONCURRENT
ALGEBRA CSEM ARE DOMAINS FOR THE
SEMANTIC FUNCTIONS DEFINED BY THE
DENOTATIONAL CLAUSES

SEMANTIC FUNCTIONS AND DOMAINS

- **Prog : PROGRAM $\longrightarrow$ ANSWER**
ANSWER = states of the concurrent system
(interpreted in the concurrent algebra)
= CSEM_{state}

- **Stat : STAT $\longrightarrow$ ENV $\longrightarrow$ CONT $\longrightarrow$ CONT**
CONT = states of processes (interpreted in CSEM)
= CSEM_{bh}

- **Exp : EXP $\longrightarrow$ ENV $\longrightarrow$ ECONT $\longrightarrow$ CONT**
ECONT = (VAL $\longrightarrow$ CONT)

- **Dec : DECS $\longrightarrow$ ENV $\longrightarrow$ DCONT $\longrightarrow$ CONT**
DCONT = (ENV $\longrightarrow$ CONT)

DENOTATIONAL CLAUSES

Prog[program bl] - initial(Stat[bl] ρ_0 nil)

initial : **CONT** $\longrightarrow$ **ANSWER**

initial(θ) = $\langle \theta, \text{stg}_0 \rangle$

{ initial(θ) = $\theta(\sigma_0)$ }

Stat[$\text{st}_1; \text{st}_2$] $\rho\theta$ = **Stat**[st_1] ρ (**Stat**[st_2] $\rho\theta$)

Exp[x] ρk = contof($\rho(x), k$)

contof : **LOC** $\times$ **ECONT** $\longrightarrow$ **CONT**

contof(1, k) = $\lambda v. \text{READ}(1, v) \Delta k(v)$

VAL

{ contof(1, k) = $\lambda \sigma. (k(\sigma(1)))\sigma$ }

$\lambda v. \text{READ}(1, v) \Delta k(v) \in \text{FUNCT}(\text{VAL}, \text{BEHAVIOUR})$

Stat[while be do st] $\rho\theta$ = .

fix $\lambda y. \text{Exp}[be]\rho(\lambda bv. \text{cond}(bv, \text{Stat}[st]\rho y, \theta))$

fix : **funct**(bh, bh) $\longrightarrow$ bh

{ **fix** $\in [|\text{CONT} \longrightarrow \text{CONT}| \longrightarrow \text{CONT}]$ }

S[create process bl] $\rho\theta$ =

CREATE(**Stat**[bl] ρ nil) $\Delta \theta$

SEMANTIC FUNCTIONS AND DOMAINS

- **Prog : PROGRAM $\longrightarrow$ ANSWER**
ANSWER = states of the concurrent system
 (interpreted in the concurrent algebra)
 = **CSEM_{state}**

- **Stat : STAT $\longrightarrow$ ENV $\longrightarrow$ BEHAVIOUR**
BEHAVIOUR = states of processes
 (interpreted in CSEM)
 = **CSEM_{bh}**

- **Exp : EXP $\longrightarrow$ ENV $\longrightarrow$ BEHAVIOUR**

- **Dec : DECS $\longrightarrow$ ENV $\longrightarrow$ BEHAVIOUR**

DENOTATIONAL CLAUSES

Prog[program bl] = <Stat[bl] ρ_0 , stg $_0$ >

Stat[st $_1$; st $_2$] $\rho\theta$ = (Stat[st $_1$] ρ) ; (Stat[st $_2$] ρ)

Exp[x] ρ = choose VAL in READ(1,v) Δ return(v)

**Stat[while be do st] ρ =
 trap [End-While $\rightarrow$ skip] in
 cycle
 def bv - Exp[be] ρ in
 cond(bv, Stat[st] ρ , exit End-While)**

**S[create process bl] ρ =
 CREATE(Stat[bl] ρ) Δ nil**

METALANGUAGE ASSOCIATED TO THE METHODOLOGY MAIN FEATURES

APPLICATIVE KERNEL + syntactic sugar
(denotational clauses)

ALGEBRAIC SPECIFICATION CONSTRUCTS
(SMoLCS concurrent system)

PRIMITIVE SPECIFICATIONS (signature + axioms)
CONSTRUCTORS (+ , enrich, derive,...)
LIBRARY OF PARAMETERIZED SPECIFICATIONS
SYNTACTIC SUGAR

**INTEGRATION OF APPLICATIVE AND
ALGEBRAIC** by associating models to the algebraic
specifications, connecting syntax to the
observational semantic algebra.

BEHAVIOUR COMBINATORS AND RELATED AXIOMS

$$\text{ACT } \Delta \text{ bh } \underline{\text{ACT}} \rightarrow \text{bh}$$

ACT Δ **bh** is a behaviour which can perform the action **ACT** and become the behaviour **bh**

$$\text{bh } \underline{\text{ACT}} \rightarrow \text{bh}'$$

$$\text{bh ; bh}_1 \underline{\text{ACT}} \rightarrow \text{bh}' ; \text{bh}_1$$

bh ; bh₁ is a behaviour which consists in the sequential composition of **bh** and **bh₁**.

$$\text{skip ; bh} = \text{bh}$$

skip is a behaviour modelling the normal transfer of control to the next behaviour

$$\text{exit ev ; bh} = \text{exit ev}$$

exit ev is a behaviour which interrupts the normal execution flow, specifying an abnormal treatment, depending on the value **ev**.

trap... in ... combinator

trap hnd in skip = skip

trap hnd in exit ev = ^{hnd}hnd(ev) IF $ev \in \text{Dom}(\text{^{hnd}ev}) = \text{true}$

trap hnd in exit ev = exit ev IF $ev \in \text{Dom}(\text{^{hnd}ev}) = \text{false}$

bh ACT > bh'

trap hnd in bh ACT > trap hnd in bh'

The behaviour **trap hnd in bh** behaves as **bh** until **bh** perform an exit to a label **ev**; if **ev** is trapped by the handler ^{hnd}**hnd** (a map from labels in behaviour) then it behaves as specified by the handler otherwise the exit is propagated.

cycle combinator

cycle bh = bh ; cycle bh

The behaviour **cycle bh** executes infinitely the activity of the behaviour **bh**.

def...in COMBINATOR

$$bh \underline{ACT} \rightarrow bh'$$

$$\mathbf{def\ x = bh\ in\ bh_1} \underline{ACT} \rightarrow \mathbf{def\ x = bh'\ in\ bh_1}$$
$$\mathbf{def\ x = return\ v\ in\ bh_1} = \mathbf{bh_1[v/x]}$$
$$\mathbf{def\ x = exit\ ev\ in\ bh_1} = \mathbf{exit\ ev}$$

def x = bh in bh₁ is a behaviour which consists in **bh**, normally terminating with the production of a value **v**, followed by **bh₁[v/x]**. If **bh** terminates abnormally then the second behaviour is not executed and the control is transferred to some enclosing behaviour.

choose...or... COMBINATOR

$$\frac{bh_1 \xrightarrow{ACT} bh_1'}{\text{choose } bh_1 \text{ or } bh_2 \xrightarrow{ACT} bh'}$$

$$\text{choose } bh_1 \text{ or } bh_2 = \text{choose } bh_2 \text{ or } bh_1$$

choose bh1 or bh2 is a behaviour consisting in the nondeterministic choice between **bh1** and **bh2**.

def x=bh1 and y=bh2 in bh'

is an abbreviation for

choose def x=bh1 in def y=bh2 in bh'
or def y=bh2 in def x=bh1 in bh'

and models a nondeterministic choice of the order in which the first two behaviours are executed.

-
-
- The Draft Formal Definition of Ada

-
- Other Dynamic Semantics Aspects
-
-
-
-
-
-
-
-

Jan Storbak Pedersen

Extent of the AdaFD dynamic semantics.

The program must be legal in every implementation.

- It must not use definitions predefined by an implementation.
 - Implementation defined types (eg SHORT_INTEGER and ADDRESS).
 - 'Undefined' predefined packages (eg LOW_LEVEL_IO and MACHINE_CODE).
 - Implementation defined attributes.

- It must not violate any legal restrictions.

- No pragma INTERFACE.
- No unchecked conversion.
- Main program must be a parameterless procedure.
But input-output is included!

- It must not use constructs without well-defined semantics.
 - Some of the above.
 - Implementation defined pragmas.

The dynamic semantics 'difficult' example Ada subset

- Predefined INTEGER type.
- Subtypes.
- One-dimensional arrays.
- Records with at most one discriminant and variant part.
- Access types.
- A few operators and attributes.
- Statements (except case and code, and restrictions on loop).
- Subprograms with positional parms of modes In and In Out.
- Packages and private types.
- Renaming of objects.
- Tasking (except tasks as components and entry families).
- No subunits.
- Exceptions.
- Generic packages with single private type parm.
- Simplified representation clauses and attributes.
- Unchecked programming.
- Simplified packages SYSTEM, DIRECT_IO and STANDARD.

The extent of the trial formal definition of Ada.

- The intersection between the subset language and what can be modelled.

Implementation Dependent Features

Possible Characteristics

- Abstract implementation aspects
- External interaction
- No abstract effect

Ways of Modelling

- Explicit parameters
- Explicit external interaction

AS2 and the $AS1 \rightarrow AS2$ transformation.

Simplifies the dynamic semantics formulae.

- Moves information to where it is needed.
- Resolves syntactic ambiguities
(eg function call / indexed component / type conversion).
- Subdivides syntactic categories
(eg full type declaration).
- Resolves overload resolution.
- Adds extra components to constructs.
(eg an extra typing component to aggregates).

AS2 and the AS1 $\rightarrow$ AS2 transformation.

- Removes unnecessary information.
- Ordering
(eg of compilation units).
- Syntactic distinctions
(eg basic / later declarative item).
- Precedence levels in expressions.

AS2 and the $AS1 \rightarrow AS2$ transformation.

- Further simplifications.
- Makes all identifiers unique.
- Removes and introduces syntactic categories (eg subprogram specification and library compilation unit).
- Changes optionality into emptiness (eg $\text{Frame} :: \text{Decl-Part} \times \text{Seq-of-Stmt} \times \text{Excp-Handler}$).
- Performs 'technical' transformations (eg identifier $\rightarrow$ generic identifier).

Underlying structures.

- $\text{Local\_Env} = \text{Identifier} \rightarrow \text{Global\_Name}$
- $\text{Global\_Env} = \text{Global\_Name} \rightarrow \text{Denotation}$
- $\text{Denotation} = \text{Left\_Value} \times \dots | \dots$
- $\text{Storage} = \text{Left\_Value} \rightarrow \text{Value}$

Subprograms.

- LRM is ambiguous and inconsistent.
- What is a call?
- When are certain checks performed?
- Values as parameters?
- Call by reference or call by name?
- We have modelled a reasonable interpretation.

6.1<1> Subprogram Declarations

Procedure Declaration

```
elab-Declarative-Item: Declarative-Item →  
    LOCAL_INF      → LOCAL_INF-BEHAV, OUR  
  
0 elab-Declarative-Item[ mk-Procedure_Dcl( prc-id, form-p)] li ≐  
1   def prc-den = make-Procedure-Den( form-p, Empty_Frame_Den)(li) in  
2   def li' = add-Denotation(prc-id, prc-den)(li) in  
3   return li'
```

6.X<1> Auxiliary Functions

make-Procedure-Den

```
make-Procedure-Den: Formal-Part x FRAME-DEN →  
  LOCAL_INF      → PROCEDURE_DEN-BEHAVIOUR  
  
0 make-Procedure-Den( form-p, frame-den) li ≡  
1   def formp-den = elab-Formal-Part( form-p) (li) in  
2   return mk-Procedure_Den( formp-den, frame-den, li)
```

6.1<3> Subprogram Declarations.

elab-Formal-Part

```
elab-Formal-Part: Formal-Part →  
    LOCAL_INF → (FORMAL_PART_DEN)-BEHAVIOUR  
  
0 elab-Formal-Part(form-part) li ≡  
1   if Is_Empty(form-part) then  
2     return Empty_Forma_Part_Den  
3   else  
4     def parm-spec-den = elab-Parm-Spec( hd form-part) (li)  
5     and form-part-den' = elab-Formal-Part( tl form-part) (li) in  
6       return <parm-spec-den> ^ form-part-den'
```


6.1<4> Subprogram Declarations.

elab-Parm-Spec

elab-Parm-Spec: Parm-Spec → LOCAL_INF → PARM_SPEC_DEN-BEHAVIOUR

```
0 elab-Parm-Spec[mk-Parm_Spec( id, mode, type, expr)] li ≐
1   let df-expr = compose-Default-Expr( expr) (li) in
2   def tp-den = read-Denotation( type ) (li) in
3   return mk-Parm_Spec_Den( id, mode, tp-den, df-expr)
```

6.3<1> Subprogram Bodies

Procedure Body

```
elab-Declarative-Item: Declarative-Item →  
    LOCAL_INF      → (LOCAL_INF) BEHAVIOUR  
  
0 elab-Declarative-Item[ mk-Procedure_Body( dcl, frame) ] li ≐  
1   let mk-Procedure_Dcl(id, form-p) = dcl in  
2   let frame-den = li' . exec-Frame(frame)(li') in  
3   def prc-den   = make-Procedure-Den( form-p, frame-den)(li) in  
4   def li''      = add-Body-Den(id, prc-den)(li) in  
5       return li''
```

5.6<2> Block Statements

```
exec-Frame: Frame → (LOCAL_INF → BEHAVIOUR)

0 exec-Frame(mk-Frame( dcl-part, seq-of-stmts, excps)) li ≐
1   def li' = elab-Declarative-Part(dcl-part)(li) in
2   trap cr-Frame-handler(li) in
3       trap Construct-Excp-Handler(excps)(li') in
4       exec-Seq-Of-Stmts-In-Frame(seq-of-stmts)(li')
```

5.6<3> Block Statements

cr-Frame-handler: LOCAL_INF → HANDLER

```
0 cr-Frame-handler(li) ≡  
1   [ e → if is-Abnormal_Completion(e) then  
2     exit e  
3     else  
4       | await-Task-termination(li); exit e  
5       | e ∈ EXIT-VALUE ]
```

5.6<4> Block Statements

exec-Seq-Of-Stmts-In-Frame: Seq-Of-Stmts $\rightarrow$ (LOCAL_INF $\rightarrow$ BEHAVIOUR)

```
0 exec-Seq-Of-Stmts-In-Frame seq-of-stmts ] li  $\hat{=}$ 
1   if Is_Task(li) then
2     end-Task-activation(li);
3   else
4     skip ;
5   activate-Tasks(li);
6   exec-Seq-Of-Stmts(seq-of-stmts) (li)
```

5.1<1> Simple and Compound Statements - Sequences of Statements

exec-Seq-Of-Stmts: *Seq-Of-Stmts* $\rightarrow$ (*LOCAL_INF* $\rightarrow$ *BEHAVIOUR*)

```
0 exec-Seq-Of-Stmts(seq-of-stmts) li  $\hat{=}$   
1   let goto-handler = Construct-Goto-Handler(seq-of-stmts) li in  
2   rec trap goto-handler in  
3       exec-Seq-Of-Labelled-Stmts(seq-of-stmts) li
```

5.1<3> Simple and Compound Statements - Sequences of Statements

Construct-Goto-Handler: Seq-of-Stmts $\rightarrow$ (LOCAL_INF $\rightarrow$ HANDLER)

```

0 Construct-Goto-Handler[seq-of-stmts] li  $\triangleq$ 
1   if Is_Empty(seq-of-stmts) then
2     []
3   else
4     let mk-Stmt( lab, unlab-stmt) = hd seq-of-stmts in
5     if Is_Null(lab) then
6       Construct-Goto-Handler( tl seq-of-stmts) (li)
7     else
8       let handler =
9         [ mk-Goto(lab)  $\mapsto$ 
10           exec-Seq-Of-Labelled-Stmts( seq-of-stmts) li ] in
11       handler + Construct-Goto-Handler( tl seq-of-stmts) (li)

```

5.1<2> Simple and Compound Statements - Sequences of Statements

exec-Seq-Of-Labelled-Stmts: Seq-Of-Stmts $\rightarrow$ (LOCAL_INF $\rightarrow$ BEHAVIOUR)

```
0 exec-Seq-Of-Labelled-Stmts[seq-of-stmts] li  $\triangleq$ 
1   if  $\neg$  Is_Empty( seq-of-stmts) then
2     let unlab-stmt = s-Unlabelled_Stmt( hd seq-of-stmts) in
3       exec-Unlabelled-Stmt( unlab-stmt);
4       exec-Seq-Of-Labelled-Stmts( tl seq-of-stmts)
5   else
6     TAU  $\Delta$  skip
```


6.4<1> Subprogram Calls

Procedure Call

```
exec-Unlabelled-Stmt: Unlabelled-Stmt → LOCAL_INF → BEHAVIOUR

0 exec-Unlabelled-Stmt[ mk-Procedure_Call_Stmt( id, parm-p) ] li ≡
1   def den = read-Denotation( id )(li) in
2   let mk-Procedure_Den( fp-den, frame-den, org-li ) = den in
3   if Is_Null( frame-den ) then
4     raise( Program_Error )(li)
5   else
6     let li' = Enter_Procedure( id, org-li, li ) in
7     def fp-den' = convert-Formal-Part-Den(fp-den)(li') in
8     def li'' = eval-Parm-Assoc( fp-den', parm-p )(li') in
9     trap [ mk-Return( nil ) →
10          copy-Back-Params( fp-den', parm-p )(li'') ] in
11       ( frame-den( li'' ) );
12     exit mk-Return( nil ) )
```

6.2<1> Formal Parameter Modes

convert-Formal-Part-Den

convert-Formal-Part-Den: *FORMAL_PART_DEN* →

LOCAL_INF → *CONV_FORMAL_PART_DEN_BEHAVIOUR*

```
0 convert-Formal-Part-Den( formp-den) li ≡  
1   if Is_Empty(formp-den) then  
2     return Empty_Converted_Formal_Part_Den  
3   else  
4     def pspec-den' = convert-Parm-Spec-Den( hd formp-den) (li);  
5     def rest-den = convert-Formal-Part-Den( tl formp-den) (li);  
6     return <pspec-den'> ^ rest-den
```

6.2<2> Formal Parameter Modes

convert-Parm-Spec-Den

convert-Parm-Spec-Den: *PARM_SPEC_DEN* → *LOCAL_INF* →
 CONV_PARM_SPEC_DEN-BEHAVIOUR

```
0 convert-Parm-Spec-Den[parm-spec-den] li ≡
1   let mode = s-Mode( parm-spec-den)
2   type = s-Subtype_Den( parm-spec-den) in
3   if Is-Scalar-Type-Den(type)(li) then
4     return Make_Conv_Parm_Spec_Den( parm-spec-den, Copy)
5   else
6     choose mechanism : { Copy, Reference } in
7     return Make_Conv_Parm_Spec_Den( parm-spec-den, mechanism)
```

11.1<1> Exception Declarations

```
elab-Declarative-Item: Declarative-Item →  
LOCAL_INF          → LOCAL_INF-BEHAVIOUR  
  
0 elab-Declarative-Item (mk-Exception_Dcl(id)) li ≐  
1 return li
```

11.2<1> Exception Handlers

Construct-Excp-Handlers

Construct-Excp-Handlers: Exception-Handlers $\rightarrow$ (LOCAL_INF $\rightarrow$ HANDLER)

```
0 Construct-Excp-Handlers( excp-hand) li  $\hat{=}$ 
1   [ excp  $\mapsto$ 
2     let mk-Exception(id) = excp in
3     if id  $\in$  dom excp-hand then
4       let li' = Update_Exception( id, li) in
5         exec-Seq-Of-Stmts( excp-hand(id))(li')
6     else if Others  $\in$  dom excp-hand then
7       let li' = Update_Exception( id, li) in
8         exec-Seq-Of-Stmts( excp-hand(Others))(li')
9     else
10      exit mk-Exception(id) | is-Exception(excp) ]
```

11.3<1> Raise Statements

exec-Unlabelled-Stmt: Unlabelled-Stmt $\rightarrow$ LOCAL_INF $\rightarrow$ BEHAVIOUR

```
0 exec-Unlabelled-Stmt [ mk-Raise_Stmt (id) ] li  $\hat{=}$   
1   if  $\neg$  Is_Null(id) then  
2     raise(id) (li)  
3   else  
4     raise(Get_Exception(li)) (li)
```

11.3<2> Raise Statements

raise

raise: *Exception-Id* → LOCAL_INF → BEHAVIOUR

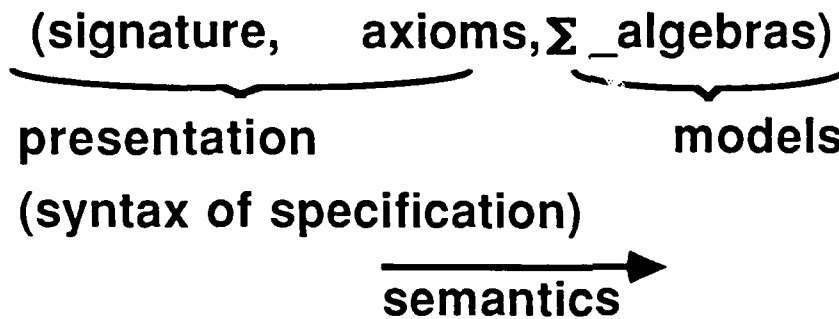
```
0 raise(id) li ≡
1   let created-tasks = Get_Created_Tasks(li) in
2   if Is_Empty(created-tasks) then
3     exit mk-Exception(id)
4   else
5     CAUSE-TERMINATION(created-tasks) Δ
6     exit mk-Exception( id)
```

FORMAL DEFINITION
SPECIFICATION
VERIFICATION

- 593

USING, FOR EXAMPLE

the algebraic framework



for a fixed semantics

signature = syntax of the language/specification

axioms = semantics " " "

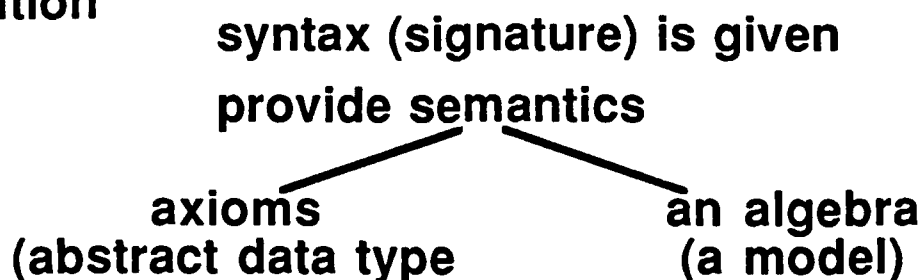
specification

presentation = abstract data type
(semantics⁺)

verification

prove that a concrete algebra
implements a specification
(that a [derived] property is satisfied)

definition



LESS ABSTRACT AND TECHNIQUE-FREE VIEW

MAIN AIMS OF A $\longleftrightarrow$ FORMAL DEFINITION

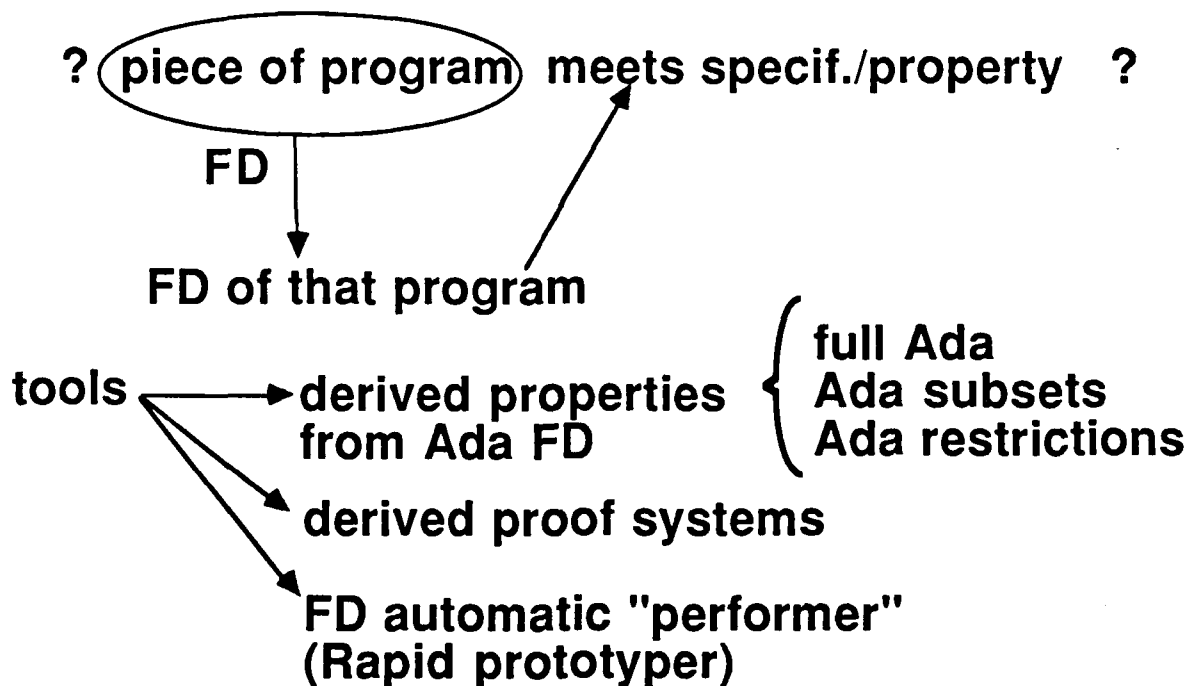
- **standard reference for implementors
(specification)**
- **semiformal guide for users
(developing and deriving natural
language explanations)**
- **basis for formal methods of proofs
(verification)**

A CLOSER LOOK

GUIDE FOR SEMIFORMAL CHECKS

(operational semantics with natural language explanations, particularly suited to common users)

FORMAL PROOFS



Development of correct programs from a specification via FD

(gap to be filled)
probably by transformations

TOWARD VERIFICATION & SPECIFICATION OF PROGRAMS

LIFTING THE LEVEL OF ABSTRACTION

Usually, Auxiliary Structures

(states, storage, environment, ---)

hence enlarged signature

(e.g., "auxiliary domains" in denotational semantics)

Theoretical Approaches for Abstraction

**find a specification of which the given FD
is an implementation**

- keeping the language syntax (O.K.)
(and compositionality)
(essentially modifying auxiliary structures
& related axioms)**
- changing the language syntax**
 - not O.K. for semantic def.**
 - may be O.K. for other purposes**

special case: observational semantics

practical formal approach

derive properties (higher level prop)

- equivalences**
- semantics of composed constructs**

FD AS A SPECIFICATION

- **basic constraint on an FD**
keep the language syntax
- **secondary, but worldwide accepted, constraint**
compositionality/syntax_directed semantics
(mathematically : semantics is a homomorphism)
give meaning to each construct by composing
canonically the meanings of subconstructs
- **optional Constraint**
semantics with local correspondence to an
informal, usually operational, explanation

compare to

- **Specification of Programs**
no compositionality constraints/global properties
- **Verification Methods**
 - **aimed at higher level properties**
(derived properties)
 - **at their best compositional, but possibly**
w.r.t. different syntax (preprocessing
transformation)

LIST OF ATTENDEES

Egidio Astesiano
Department of Mathematics
University of Genova & CRAI
Via L.B. Alberti 4
16132 Genova
ITALY
+39-10-515142
Telex: 271114 UNIVGE I

adafd @ icnucevm.bitnet

Patrick de Bondeli
CR2A and AEROSPATIALE/Space Division
14 Boulevard Jean Mermoz
92200 Neuilly-S-Seine
FRANCE
Office: 33-1-47689797
Home: 33-1-47220614

Richard Carver
Department of Computer Science
Box 8206
N.C. State University
Raleigh, NC 27695-8206

William L. Caudle
Sperry Corporation
3500 Parkway Lane
Suite 600
Norcross, GA 30092
(404) 263-1605

John Chludzinski
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 824-5518

JChludzinski@Ada20.isi.edu

Norman Cohen
SofTech, Inc.
One Sentry Parkway
Suite 6000
Blue Bell, PA 19422-2310
(215) 825-3010

NCohen@Ada20.isi.edu

Paul M. Cohen
AJPO/DoD Technology Analysis Office
Pentagon 3D139
Fern Street
Washington, DC 20301-3081
(202) 694-0211

PCohen@Ada20.isi.edu

Mark R. Cornwell
Naval Research Lab
Code 7590
Naval Research Lab
Washington, DC 20375
(202) 767-6698

cornwell@nrl-css

Don Couch
8 Lakeland Dr.
Lawton, OK 73501

Robert Cutting
Sperry
Atlanta Development Center
3500 Parkway Lane
Norcross, GA 30092
(404) 263-1676

Patricia W. Daggett
Data General Corporation
62 T.W. Alexander Drive
Research Triangle Park, NC 27709
(919) 248-6144

Steve Eckmann
SDC
2525 Colorado
Santa Monica, CA
(213) 820-4111

Eckmann@Dockmaster

John Faust
Rome Air Development Center
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241

Faust@RADC-MULTICS

Clarence "Jay" Ferguson
NCSC
ATTN C324
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 850-7161

CFerguson@Dockmaster

Peter M. Fonash
Defense Communications Agency
1860 Wiehle Avenue
Reston, VA 22090
(703) 437-2189

FONASH@ISIF
FONASH@EDN-VAX

Helen Gill
MITRE
1820 Dolley Madison Boulevard
McLean, VA 22102
(703) 883-7980

gill@mitre

Dale J. Goumer
Magnavox Electronic Systems
1313 Production Road
Fort Wayne, IN 46808
(219) 429-6000

Marc Graham
Sperry Corporation
3500 Parkway Lane
Suite 600
Norcross, GA
(404) 263-1621

Jeani Hackett
SofTech, Inc.
3100 Presidential Dr.
Fairborn, OH 45324-2039
(513) 429-2771

Scott Hansohn
Honeywell SCTC
2855 Anthony Lane So.
Suite 130
St. Anthony, MN 55418
(612) 782-7144

Hansohn@HI Multics

Bret Hartman
Research Triangle Institute
Research Triangle Park, NC 27709-2194
(919) 541-6110

hartman@rti-sel

Brian Holland
NCSC, C3
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-4494

brian@TYCHO
Holland@Dockmaster

Steven Holtsberg
System Development Corporation
2525 Colorado Place
Santa Monica, CA 90406
(213) 821-4111 x5672

sdcrdcf!steve@ucla-cs

Ronald D. Hubbard, Jr.
Odyssey Research Associates
1283 Trumansburg Road
Ithaca, NY 14850
(607) 277-2020

Dale M. Johnson
The MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-8436

dmj@mitre-bedford

Roger B. Jones
International Computers Limited
Eskdale Road
Winnersh, Wokingham
Berkshire RG11 5TT
ENGLAND
0044 734 693131

USENET:...!mcvax!ukc!stc!rbj

John M. Kinsley
Link Flight Simulation Division of Singer
P.O. Box 1237
MS 521
Binghamton, NY
(607) 772-4009; secretary 772-4126

Kenneth Kung
Hughes Aircraft 618/Q315
P.O. Box 3310
Fullerton, CA 92634
(714) 732-0262

KKUNG@USC-ECLA

Nancy Leveson
University of California at Irvine
ICS Department
Irvine, CA 92717
(714) 856-5517

Nancy@ics.uci.edu

Timothy E. Lindquist
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965-2783

Lindquis@asu (CSNET)

Davkd Luckham
Stanford University
ERL 456
Electrical Engineering
Stanford, CA 94305
(415) 723-1242

Luckham@SAIL

Andrew D. McGettrick
Computer Science Department
University of Strathclyde
Livingstone Tower
Glasgow G1 1XH
Scotland, U.K.
(within UK) 041-552-4400 (x3305)

John McHugh
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709-2194
(919) 541-7327

MCHUGH@UTEXAS

W.T. Mayfield
Institute for Defense Analyses
Computer and Software Engineering Division
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 824-5524

TMayfield@Ada20.isi.edu

Mark Miller
Commander - CECOM
ANSEL-SDSC-SS
Fort Monmouth, NJ 07703

Harlan D. Mills
IBM
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1495

Charles S. Mooney
Grumman Aerospace Corp.
Bethpage, NH 11714
(516) 575-8203

Gilbert Myers
Naval Ocean Systems Center
Code 423
San Diego, CA 92152
(619) 225-7401

GMYERS@ISIF

LCDR Philip A. Myers
U.S. DoD (DUSD R&AT)
Ada Joint Program Office
3D139
1211 Fern Street, C107
Pentagon
Washington, DC
(202) 694-0209

PMYERS@Ada20.isi.edu

Karl A. Nyberg
Grebyn Corporation
P.O. Box 1144
Vienna, VA 22180
(703) 281-2194

NYBERG@ISIF

Myron Obaranec
U.S. Army
CECOM
AMSEC-COM-AF
Fort Monmouth, NJ 07703
(201) 544-4962

Jan Storbak Pedersen
Dansk Datamatik Center
Lundtoftevej 1c
DK-2800 Lyngby
DENMARK
+45-2-872622

Richard Platek
Odyssey Research Associates
1283 Trumansburg Road
Ithaca, NY 14850
(607) 277-2020

David Preston
IITRI
4550 Forbes Boulevard
Suite 300
Lanham, MD 20706
(301) 459-3711

Edmond Schonberg
New York University
251 Mercer Street
New York, NY 10012
(212) 460-7482

Michael Schwartz
Martin Marietta Information &
Communications Systems
Mail Stop L0402
Denver, CO 80201
(303) 977-0421

Jerry Shelton
Verdix Corporation
14130-A Sullyfield Circle
Chantilly, VA 22021
(703) 378-7600

Craig David Singer
Research Triangle Institute/Duke University
1315 Morreene Road #2G
Durham, NC 27705
(919) 383-4287

KHansen@USF-ISIF.ARPA

RPLATEK@ISIF

ebooth@Ada20.isi.edu

Schonberg@NYU

MSchwartz-@Dockmaster

vr dxhq!jhs@seismo

cds@duke or cds@rti

Brian Siritzky
New York University
251 Mercer Street
New York, NY 10012
(212) 460-7239

siritzky@acf2 or
cmcl2!acf2!siritzky

Gary R. Smith
Texas Instruments
Electronic Warfare Systems
5825 Mark Dabbling Boulevard
MS 3719
Colorado Springs, CO 80919
(303) 593-5359

Ryan Stansifer
Purdue University
Department of Computer Science
West Lafayette, IN 47907
(317) 494-7281

ryan@Purdue.edu

K.C. Tai
North Carolina State University
Department of Computer Science
Box 8206
Raleigh, NC 27695-8206
(919) 737-7862

USENET mcnc!ncsu!kct

Bill Tiegs
Sperry Corporation
Software Systems, Computer Systems Division
Sperry Park
P.O. Box 64525, MS Y41A6
St Paul, MN 55164-0525
(612) 456-7385

Friedrich W. von Henke
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-2560

vonHenke@SRI-CSL

Steve Welke
107-B Cresap Road
Charlottesville, VA 22903

SWelke@Ada20.isi.edu

Distribution List for IDA Memorandum Report M-241

NAME AND ADDRESS	NUMBER OF COPIES

Sponsor

Dr. John P. Solomond Director, Ada Joint Program Offices The Pentagon, Room 3D139 Washington, D.C. 20301	2

Other

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2

IIT Research Institute 4550 Forbes Blvd., Suite 300 Lanham, MD 20706	1

Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1

IDA

General W. Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Anne Douville, CSED	1
Dr. Richard L. Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	2
Ms. Katydean Price, CSED	2
Mr. Steve Welke, CSED	2
Dr. Richard Wexelblat, CSED	1
IDA Control & Distribution Vault	3