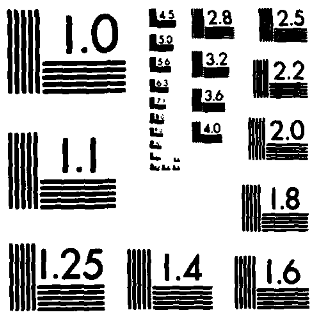
AD-A171 507 PROCEEDINGS OF THE IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION O..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C G ROBY DEC 85 IDA-M-146 1/5

UNCLASSIFIED IDA/HQ-85-30658 MDA903-84-C-0031 F/G 9/2 NL



'CROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Copy 119 of 127 copies

AD-A171 507

(2)

IDA MEMORANDUM REPORT M-146

PROCEEDINGS OF THE FIRST IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION OF Ada* 18-20 MARCH 1985

Clyde G. Roby, *Editor*

December 1985

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

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

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

86 8 20 007

IDA Log No. HQ 85-30658

SECURITY CLASSIFICATION OF THIS PAGE AD-A171 5 07

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) M-146	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses; 6b. OFFICE SYMBOL (If applicable) IDA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Ada Joint Program Office; 8b. OFFICE SYMBOL (If applicable) AJPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1211 Fern St., Room C-107 Arlington, VA 22202	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-4-263 / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada, 18-20 March 1985 (U)

12. PERSONAL AUTHOR(S) Clyde G. Roby, editor

13a. TYPE OF REPORT	13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day) 1985 December	15. PAGE COUNT 364

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada, verification, specification, secure systems, semantic, concurrency, computer security, software, support library, run-time support library

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The first IDA Workshop identified current issues in Ada verification: the uses of formal verification; what verification techniques and verification systems are available; what practical experience is there in the use of these approaches and who has this experience; what impact does Ada have on verification (both before and during coding activities); what are the major problems in the verification field; and what needs to be done to overcome these problems. Slides presented at the two-and-half day workshop are included.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

IDA MEMORANDUM REPORT M-146

PROCEEDINGS OF THE FIRST IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION OF Ada* 18–20 MARCH 1985

Clyde G. Roby, *Editor*

December 1985

IDA
INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-4-263

Foreword

These Proceedings of the First Workshop on Formal Specification and Verification of Ada, held at the Institute for Defense Analyses, are composed in part of papers and slides supplied by the speakers, and in part of summaries of the talks and discussions edited from recordings made of the Workshop.

The purpose of this initial two-and-a-half day Workshop was to identify current issues in Ada verification and to decide what could be done to improve current understanding and practice of Ada software verification. Since verification impacts not only coding activities but all development activities, it is desirable that many groups be kept informed about the progress of these Workshops.

The chief issues raised in the introductory remarks by Jack Kramer of IDA and Paul Cohen of the AJPO were: what are the uses of formal verification; what verification techniques and verification systems are available; what practical experience is there in the use of these approaches and who has this experience; what impact does Ada have on verificaton (both before and during coding activities); what are the major problems in the verification field; and what needs to be done to overcome these problems.

Below is the list of presentations, which gives the scientific program of the first two days, followed by a summary of the discussions of the third day. Finally, a list of participants, with their postal, telephonic, and electronic addresses is given.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

The First Day

On Monday, March 18th, 1985, after opening addresses by Jack Kramer and Paul Cohen, four talks were given:

A. Richard Platek (Odyssey): Towards the Formal Verification of Ada Programs

B. David Guaspari (Odyssey): Towards Ada Verification

C. Richard Platek (Odyssey): Formal Specification

D. David Luckham (Stanford): ANNA, a Specification Language for Ada

The second day

On the morning of Tuesday, March 19th, 1985, there were two parallel sessions, each containing four talks:

Session on Secure Systems in Ada
Chair: Margie Zuk (MITRE)

E. Tony Brintzenhoff (SYSCON): Re-implementing ACCAT Guard in Ada

F. Eric Anderson (TRW): Army Secure Operating Systems

G. Jim Freeman (Ford Aerospace): Trust Domains

H. LCDR Philip Myers (NAVALEX): Navy Technology and Ada

Session on Advanced Verification
Chair: Krzysztof Apt (IBM Yorktown Heights)

K. Krzysztof Apt (IBM Yorktown Heights): Reconsidering Correctness of CSP Programs

L. Frank Oles (IBM Yorktown Heights): Thoughts on an Ada-based Design Language

M. Norman Cohen (SofTech): Axiomatic Semantics for Ada

N. David Gries (Cornell): Teaching Programmers about Proofs

P. Discussion of papers in the session

In the afternoon there were seven talks and a lively discussion:

Session on Near Term Verification Systems
Chair: Donn Milton (VERDIX)

Q. Jim Williams (MITRE): Practical Verification Systems

R. Ryan Stansifer (ORA): Near Term Ada Verification

S. Ray Hookway (Case Western): Adapting a Modula Verification System to Ada

T. John McHugh (RTI): Adapting the GYPSY Verification System to Ada

U. Discussion of papers in the session

Session on Verification and Software Engineering
Chair: Norman Cohen (SofTech)

V. Norman Cohen (SofTech): Uses of Formal Verification

W. Friedrich von Henke (SRI): ANNA

X. David Luckham (Stanford): ANNA Tools

The Third Morning: General Review

The final session, on the morning of Wednesday, March 20th, began with summaries of the parallel sessions and continued with a general discussion reviewing issues raised during the Workshop.

Richard Platek said that although the meeting had been about all levels of verification, the first need was to sort out the problem of specification. He emphasized the urgent need for a specification language tailored to Ada, and called for several proposals, in addition to ANNA, which might then be compared. The experience of Honeywell with the SCOMP project was discussed and attention drawn to the great difficulty of writing abstract specifications in GYPSY. Norman Cohen remarked that a good specification for GYPSY would not necessarily be a good one for Ada.

LCDR Philip Myers asked whether formal semantics and concurrency are issues that need to be resolved before large strides can be made in Ada verification. A. L. Brintzenhoff mentioned his work on evaluating the role of Ada as a communications programming language. Jack Kramer led a discussion of the role of verification in the life cycle of software development systems.

The special application of verification to secure systems was then discussed. LCDR Myers said that those developing them must show that the systems are robust enough to defend against the "414 hackers", and that a more secure development environment is needed. Richard Platek said that it was desirable for Ada verification systems to be exposed to public criticism during the early stages of their development, as that would help to test their soundness.

The structure and future of the body constituting the Ada Verification Workshop was discussed. David Luckham said that the Workshop had a valuable role to play as a forum for continued information exchange. Norman Cohen stressed the need for a committee to consider not only interpretations of the Ada Reference Manual but also revisions of the language. Fridrich von Henke pointed out the lack of activities on the more formal aspects of Ada specification and drew attention to the working groups of Ada-Europe on Formal Semantics and Formal Methods. It was suggested that the possibility of forming a Committee on Formal Methods within SIGAda be explored.

The success of the IBM Cleanroom project under Harlan Mills in training some 2000 programmers in practical but informal methods of program testing was mentioned. LCDR Myers warned that some success stories will be needed in the near term if Ada Verification tools are not to miss the Navy boat. Richard Platek suggested that a new use for verification might be found in the fact that concurrent programs are so complex that verification is needed even to write them.

The Third Morning: Preparation for the next Workshop

There was general agreement that the last several years of effort has yielded some useful techniques. The role of these Workshops will be to act as a mechanism for establishing a group of experts that can assess the current state-of-the-art, identify promising research areas, monitor ongoing verification work, promote the use of the evolving technology, and ensure that valuable outputs from one area are fed into another area. The chief output of this group will be recommendations to various bodies to coordinate and sponsor certain R&D activities. It was agreed that Working Subgroups on special topics should be established, as follows:

SECURE SYSTEMS chaired by Margie ZUK
MITRE Corporation, Burlington Road, Bedford MA 01730;
(617) 271-7590;
MMZ@MITRE-BEDFORD

NEAR TERM VERIFICATION chaired by John McHUGH
Research Triangle Institute, Box 12194, Research Triangle Park, NC 27709;
(919) 541-7327;
McHUGH@UTEXAS-20

FORMAL SEMANTICS AND CONCURRENCY chaired by Norman COHEN
SofTech, Inc., 705 Masons Mill Business Park, 1800 Byberry Road, Huntingdon Valley, PA 19006;
(215) 947-8880
NCOHEN@ECLB

SPECIFICATION LANGUAGES chaired by Friedrich VON HENKE
SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025;
(415) 859-2560;
VONHENKE@SRI-CSL

VERIFICATION IN LIFE CYCLES chaired by Ann MARMOR-SQUIRES
TRW, Defense Systems Group, 2751 Prosperity Avenue, Fairfax, VA 22031;
(703) 876-8170;
MARMOR@ISI

"OFFICIAL" CLUSTERS chaired by Richard PLATEK
Odyssey Research Associates, Inc., 408 E. State Street, Ithaca, NY 14850;
(607) 277-2020;
RPLATEK@ECLB

It was envisaged that these subgroups should prepare material for the next Workshop and, where appropriate, draft recommendations for forwarding to the relevant official bodies after discussion at that meeting. There was some discussion of the areas where such recommendations might be needed, such as near-term, mid-term, and long-term Ada verification; the lessons learned in verification; formal semantics for verification; and the future role of Ada Verification Workshops.

The account ADA-VERIFY was created on USC-ECLB and will be used as a central repository for Ada Verification announcements, files, etc. The following general mailing list was also established on USC-ECLB to encourage the exchange of ideas.

Ada-VERIFICATION-LIST

Ada Verification Workshops should be held on a regular basis. The scope of the Workshops should not be restricted to looking at verification of Ada code, but to address verification throughout the software life cycle. They will also serve the purpose of providing a focal point for the development of verification technology and a coherent set of activities that will address current verification issues. The Workshop can also promote increased practical use of verification techniques.

The next Ada Verification Workshop will be scheduled in the July/August timeframe for a period of two - and - one - half days (preferably from Tuesday through Thursday). The place of this meeting was not decided; offers of hospitality from RTI, MITRE, and TI were received.

It was agreed that attendance at the next meeting should be by invitation, and that it would be undesirable to lose the constructive character resulting from the small size of the present meeting. It was suggested that our existence be advertised and requests for invitations be solicited through ACM's Software Engineering Notes and AdaLetters and that particular thought should be given to inviting people from the Language Maintenance Committee, the AJPO, and the safety community.

Plans are under way to organize a session at the July/August SIGAda meeting in Minneapolis that apprise the technical community of the role and purpose of the Workshop series and what has been documented by that time.

The Workshop ended at noon on the third day.

ADA VERIFICATION WORKSHOP

MARCH 18-20, 1985

LIST OF PARTICIPANTS

Bernard Abrams — ABRAMS@ADA20
Grumman Aerospace Corporation
Mail Station 001-31T
Bethpage, NY 11714
(516) 575-9487

Dr. Thomas C. Antognini — TCA@MITRE-BEDFORD or TCVB@MITRE-BEDFORD
MITRE Corporation
Mailstop B330
Burlington Road
Bedford, MA 01730
(617) 271-7294

Krzystof Apt
Thomas J. Watson Research Center
P. O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2923

Alton L. Brintzenhoff — SCI-ADA@USC-ISI
SYSCON Corporation
3990 Sherman Street
San Diego, CA 92110
(619) 296-0085

Richard Chan — RCHAN@USC-ECL
Hughes Aircraft Co.
P. O. Box 33
FU-618/P115
Fullerton, CA 92634
(714) 732-7659

Norman Cohen — NCOHEN@ADA20
SofTech, Inc.
1 Sentry Parkway
Suite 6000
Blue Bell, PA 19422
(215) 825-3010

Paul M. Cohen PCOHEN@ADA20
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

Mark R. Cornwell CORNWELL@NRL-CSS
Code 7590
Naval Research Lab
Washington, D.C. 20375
(202) 767-3365

Jeff Facemire FACEMIRE%TI-EG@CSNET-RELAY
Texas Instruments
P.O. Box 801
M/S 8007
2501 West University
McKinney, TX 75069
(214) 952-2137

John C. Faust FAUST@RADC-MULTICS
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241

James W. Freeman
Ford Aerospace & Communications Corp.
Mailstop 15A
10440 State Highway 83
Colorado Springs, CO 80908
(303) 594-1536

Inara Gravitis GRAVITIS@ADA20
SAIC
1710 Goodridge Drive
McLean, VA 22202
(703) 734-4096 or (202) 697-3749

David Guaspari RPLATEK@ADA20
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

J. Daniel Halpern SYTEK@SRI-UNIX or MENLO70!SYTEK!DAN@BERKELEY
SYTEK Corp.
1225 Charleston Road
Mountain View, CA 94043
(415) 966-7300

Brian E. Holland BRIAN@TYCHO
DoDCSC, C3
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6968

Ray Hookway HOOKWAY%CASE@CSNET-RELAY
Department of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(216) 368-2800

Paul Hubbard HOOKWAY%CASE@CSNET-RELAY
Department of Computer Eng. & Science
Case Institute of Technology
Case Western Reserve University
Cleveland, OH 44106
(26) 368-2800

Larry A. Johnson LJOHNSON@MIT-MULTICS
GTE
77 "A" Street
Needham, MA 02194
(617) 449-2000 ext. 3248

Dr. Jack Kramer KRAMER@ADA20
Institute for Defense Analyses
Computer & Software Engineering Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2263

Eduardo Krell
3804 Boelter Hall
UCLA
Los Angeles, CA 90024

Randall E. Leonard
Army System Software Support Command
ATTN: ASB-QAA
Fort Belvoir, VA 22060

Steven Litvintchouk SDL@MITRE-BEDFORD
Mail Stop A180T
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7753

David Luckham LUCKHAM@SAIL
Stanford University
Computer Systems Lab, ERL 456
Stanford, CA 94305
(415) 497-1242

Ann Marmor-Squires MARMOR@ISI
TRW
Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031
(703) 876-7223

A. R. D. Mathias RPLATEK@ADA20
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

John McHugh MCHUGH@UTEXAS-20
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

Donn Milton VRDXHQ!DRM1@SEISMO
Verdix Corporation
14130-A Sulleyfield Circle
Chantilly, VA 22021
(703) 378-7600

LCDR Philip A. Myers MYERS@NRL-CSR
Space and Naval Warfare Systems Command 1st 2nd
SPAWAR 8141A
Washington, DC 20363-5001
(202) 692-8484

Karl Nyberg NYBERG@ADA20
Verdix Corporation
14130-A Sulleyfield Circle
Chantilly, VA 22021
(703) 378-7600

Myron Obaranec LAKSHMI@CECOM-1
U. S. Army, CECOM
Fort Monmouth, NJ 07703
ATTN: AMSEL-TCS-SIO
(201) 544-4962

Frank J. Oles
Thomas J. Watson Research Center
P.O. Box 218
88-K01 Route 134
Yorktown Heights, NY 10598
(914) 945-2012

Richard Platek RPLATEK@ADA20
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

David Preston DPRESTON@ADA20
IITRI
5100 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711

Edmund Robinson
Peterhouse, Cambridge CB2 1RD
United Kingdom
(0223) 350256

R. Max Robinson RROBINSON@ADA20
Institute for Defense Analyses
Computer & Software Engineering Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2097

W. A. Robison
30 Charles Street West
Apt. # 1811
Toronto, Ontario, CANADA
M4Y 1R5
(416) 925-0751

Clyde G. Roby CROBY@ADA20
Institute for Defense Analyses
Computer & Software Engineering Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2541

Ryan Stansifer RPLATEK@ADA20
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 227-2020

David Sutherland RPLATEK@ADA20
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 227-2020

Friedrich von Henke VONHENKE@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-2560

Barry Watson WATSON@ADA20
Ada Information Clearinghouse
IITRI
Room 3D139 (1211 Fern St., C-107)
The Pentagon
Washington, DC 20301
(703) 685-1477

Jim Williams JGW@MITRE-BEDFORD
MITRE Corporation
Mailstop B332
Burlington Road
Bedford, MA 01730
(617) 271-2647

Larry Yelowitz KLY@FORD-WDL1
Ford Aerospace and Communication Corp.
Western Development Laboratory Division
Mailstop X-20
3939 Fabian Way
Palo Alto, CA 94303
(415) 852-4198

Christine Youngblut CYOUNGBLUT@ADA20
Advanced Software Methods, Inc.
17021 Sioux Lane
Gaithersburg, MD 20878
(301) 948-1989

Francoise Youssefi FYOUSSEFI@ADA20
Institute for Defense Analyses
Computer & Software Engineering Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-3605

Margie Zuk MMZ@MITRE-BEDFORD
Mailstop B321, Bldg B
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7590

Appendix A

Towards the Formal Verification of Ada Program

Richard Platek
Odyssey Research Associates, Inc.

Towards the Formal Verification of Ada Programs

Richard Platek

Odyssey Research Associates, Inc.

Table of Contents

Towards the Formal Verification of Ada Programs

Richard Platek
Odyssey Research Associates, Inc.

March 18 - 20, 1985

1 Introduction

The organization of the present meeting is part of a contractural effort between Odyssey Research Associates (ORA) and IDA. One of the goals of this effort, and the purpose of this meeting, is to begin to develop a community of interest centered around the formal specification and verification of Ada programs. Some of the issues to be discussed are: the use of Ada in secure systems; verification needs as perceived by the general Ada community; near term solutions to Ada verification; the issue of standards as they apply to formal specification/verification; advanced theoretical verification results applied to Ada; the interface between verification and other software engineering concerns.

2 The Verification Worlds

I would like to begin by looking at the verification world as it presently exists. I am aware of two essentially disjoint groups focusing on formal verification. The first is the academic community generally funded through NSF or DARPA; the second consists of industrial contractors involved with the building of secure systems and generally funded through the DoD Computer Security Center. The academic verification world is centered at Cornell, MIT, Stanford, and CMU and reports at the annual ACM POPL (Principles of Programming Languages), the annual IEEE FOCS (Foundations of Computer Science), and the biennial Logic of Programs Symposium and Conference on Automated Deduction (CADE). The Industrial Verification world is centered at SRI, SDC, UTexas, and hopefully someday ORA, and reports at the annual DoD/NBS Computer Security Conference, the annual IEEE Symposium on Security and Privacy, and the Verkshops, the most recent of which (the third) was held exactly one month ago in February, 1985. Neither of these tribes, the academic and industrial, have rallied to the banner of Ada verification and I think we should begin by examining why.

In the academic world the mathematical paradigm of abstraction and idealization is the principal method of investigation. This method studies the logic of some programming language construct, such as iteration or concurrency, by building a simplified model of it and then formulating properties of the model which are sufficient to develop proof rules adequate to capture the semantics of the simplified model. What do I mean that the model is simplified? First, it is customary to restricc oneself to integer and boolean types with the integers considered to be the infinite set of mathematical integers. Second, single programming language features are studied in isolation and the thorny problem of the interaction of various language features is ignored. For example, while a logic of pointer types (Ada's access types) which covers allocation, deallocation, and assignment has been developed and a logic of simple concurrency including inter-task communication has also been developed (I'm thinking of Hoare's CSP) the academic literature does not address inter-task communication using access types and this is a non-trivial matter. In an actual language, like Ada, all these features are intertwined and there is a very rich type structure including task types and dynamic task creation. Furthermore, Ada's raising and handling of exceptions related to the finiteness of the integers complicates the simplified academic results which assume the mathematical integers. Of course, one can not object to simplification as an approach but it is odd that the academic community finds their theoretical results meaningful when in truth they have very little relationship to the real problem of verified software.

While the academic world is free to proceed in an imaginative fashion, replacing hard, real problems by simpler easier ones, the industrial world is faced (or will soon be faced) with the actual job of producing verified running code. The feeling there is that this problem is hard enough in simple languages like Pascal without introducing a super-rich language like Ada. Gypsy, for example, was designed to be an inherently verifiable language: it eliminates side-effects by forbidding global variables; pointers are replaced by a few dynamic type constructors (e.g., arbitrarily long sequences); concurrency is restricted to cobegin statements whose child processes can only share buffers (FIFO queues with history sequences to support specification/verification); etc. Although Gypsy has served as an excellent teaching tool in the area of verification, the bottom line is that although the Gypsy Verification Environment effort was begun in 1975 and has been stable for a number of years there is no fielded Gypsy software! Thus, we know nothing about the life-cycle of verified software -- we do not know how user bug reports relate to the original verification effort; we do not know to what extent modification of requirements and maintenance force reverification. Such information would be a useful input into the design of an Ada verification system. The Gypsy people feel that such information should be gathered before we plunge into Ada. On the other hand, Gypsy's severe restrictions make certain kinds of applications difficult if not impossible. For example, the absence of global variables make it difficult to implement transaction-oriented systems like data bases in the usual manner although it can be done using buffers.

One of the explanations of the industrial world's reluctance to tackle Ada head-on is the fact that they are scientifically about a decade behind the academic world. This decade-gap is an observation not a criticism; the transformation of science into technology is a formidable undertaking and there are not sufficiently knowledgeable personnel in the area of program verification to staff the industrial positions which are arising. Gypsy, designed in 1975, is a good example of the decade-gap; another is the current effort by I.P. Sharp to build a Euclid verification system. Euclid was also designed about ten years ago (by a blue ribbon committee) in an attempt to devise a language which is inherently verifiable. More flexible than Gypsy, Euclid is based on the academic results in program verification of that era. It attempts to control the interactions between language features in such a way that the simplified academic results still apply. Actually, in the course of their work, I.P. Sharp discovered that the Euclid definition and verification results contained errors. Furthermore, I.P. Sharp's approach to verification is through the generation of Verification Conditions. This approach, which separates programs from proofs, is considered inadequate by academic researchers and other verification paradigms, such as program transformations, are currently predominant.

In summary, it appears that while academics produce the basic results which will be embodied in verification systems their approach is to study verification in a simplified fashion ignoring interactions of constructs. The industrial verification people, on the other hand, feel that we should learn to walk before we try to run. By that they mean that we need basic knowledge, which we don't have, about the feasibility of formal verification and the role it plays in the software life-cycle. Such knowledge, it is felt, can best be gathered on the basis of inherently verifiable languages. In this way the problem of verifiability can be separated from the question of its utility and the embarassing fact of life is that we don't know the utility of formal verification. Essentially, is it worth it? We can't answer that because we don't know its cost and we don't know its benefits.

Which brings us to Ada. The need to have a common DoD HOL resulted in Ada which is an engineering compromise between contending requirements each of which have strong supporters in certain user communities (e.g., the real-time people, the compiler people, the systems people, etc.). While formal verifiability was one of the original goals it lost ground to other requirements which were felt to be more pressing. That is not really suprising since, as I mentioned above, the verification people have not yet proved their case.

Although the verification people have not yet proved their case the possibility of wide spread use of provable, non-toy software still retains its appeal after so many years of expectant waiting. The explanation of the appeal is of course the pay-off which would result from successful, industrial verification. After all, what are the alternative assurance criteria for secure systems, strategic systems, SDI, commercial life-critical systems? And what really are the alternatives to Ada? Assuming the compiler and tool building programs are a success it seems to me that the utility of a universal standard will dominate in all areas of computer usage. One has seen this with the IBM PC and is presently seeing it with Unix. It doesn't matter that both of these de facto standards leave much to be desired; they simplify the world. So I believe Ada will triumph and if the verification people wish to be relevant they ought to try to understand the way the world actually works.

One of the Tasks in ORA's effort is to determine whether users perceive a need for formal verification. I am not speaking about builders of secure systems who when "Beyond A1" becomes a reality will be forced to verify the security critical components of their code. Instead, I am speaking about system builders who are told about formal verification and then asked how they would use it if it were available.

3 Questionnaire Development

Our approach to determining user needs has been to develop and circulate a questionnaire among people with experience in programming large systems. This questionnaire is meant to find out two things:

- What sort of system functionality is the most critical? That is for which functions would formal verification most increase reliability?

- Which Ada language constructs are the most heavily used?

Formal program verification is basically unkown and barely used in the software development process. We did not simply ask, "do you have a need for formal program verification?", because that would probably have evoked no response at all. (In fact, in military software development, the phrase "formal verification", sometimes refers to the act of the military customer certifying that a battery of tests has been run on the software.) Instead, we tried to find those aspects of the software product the developer is willing to spend the greatest amount of money to test: this may be the surest indication of a need for formal verification.

Correct proof rules are not known for all of Ada, but the scope of Ada verifiablity can certainly be increased by research. Thus, we also set out to find out the directions in which future research would be most useful, i.e., which Ada language features not now known to be verifiable do software engineers feel are most critical?

Initially, a draft form of the questionnaire was prepared. This draft was sent to a number of people for comments. We chose roughly a dozen individuals whose opinion we respect, some from the verification community, some designers of large software systems. After collecting comments from these people, we modified the questionnaire, incorporating many suggested improvements. The modifications roughly doubled the length of the questionnaire.

In its final form, the questionnaire divides into two parts:

1. general systems development experience

2. Ada-specific experience.

The first part asks for information about a system the respondent has worked on, not necessarily involving Ada. The questions pin down the type of system developed, its size, languages and tools used, and a brief statement of its purpose. The questions then try to determine how much testing effort was or is expected to be devoted to the project, and in what specific areas is the greatest fraction of effort devoted. The point is: if a developer is going to spend dollars on assurance, what critical functions, modules, or features will be deemed the most important to verify?

The second part asks questions about the use of Ada. Which language constructs are currently used, which are never expected to be used, and which are avoided now because of lack of faith in the particular compiler used.

The final form of the questionnaire is not limited to a survey of Ada users; nevertheless, we decided to concentrate on the Ada community for the first mailing. Our main source of contacts was a list of current Ada contracts compiled by Ann Reedy and published in Ada Letters. We telephoned most of the organizations on that list, both to determine the most appropriate recipient of the questionnaire, and to ask knowlegible people in the Ada community for other potential contacts.

4 Survey Results

Questionnaires were sent to key people in the following organizations. 18 individuals in 15 organizations have responded to date. The last questionnaires were sent out Feb 22, and responses are still arriving.

Organization	Project	# responses
Singer/Librascope	front-end for TACFIRE	X
Singer/Librascope	message communication terminal	X
Dalmo Victor Operations	tank sensor integration	XX
Veda	generic message editing	XX
Tasc	ASAP	
McDonnel Douglas	CAMP	
McDonnel Douglas	porting ICSC Ada	
McDonnel Douglas	convert AIS to Ada	X
Ford Aerospace	G-3 Maneuver Control	X
General Dynamics	TAG	
General Dynamics	decision support system	X
General Dynamics	IMF	
GTE	WIS	
Magnavox	AFATDS	X
Harris Corp.	ALPC	
Sonicraft	MEECM	
LTS	MEECM	
NAV AIR	F-18 operational flight program	X
NAV AIR	aircraft control & HUD	X
Syscon	ACCAT GUARD Ada reimplementation	X
RCA	MCF RTM O/S / ASOS	X
System Development Corp.	STARS	XX
TRW	STARS	
TRW	prototype advanced APSE	
TRW	ASOS	
TRW	TDBMS	
Computer Corp of America	Ada DBMS	
Intermetrics	hardware description lang. analyzer	X
Intermetrics	S/370 Ada compiler	X
Telesoft	WIS compiler	
SofTech	Ada/M UYK-44 retarget	
NYU	Ada/Ed	
Florida SU	Cyber 170 Ada compiler	
UC Irvine	Arcturus	

Several interesting results have emerged so far.

1. Many Ada systems have interfaces to other languages. The foreign languages were various assembly languages, FORTRAN, and in one case, PASCAL.

2. Correctness and precision of floating-point computations are not large problems for testing.

3. Denial-of-service problems receive relatively less testing effort than timing constraints.

4. A surprising number of respondents had encountered erroneous programs or programs with incorrect order dependence. One respondent called this question "academic nit-picking"; unfortunately, the fact that some Ada users encounter these situations implies that it is not "nit-picking", academic or otherwise.

5. Absolutely no respondent uses or claims to have an urge to use tasks passed as parameters to subprograms.

6. Few Ada users can make do without access types.

7. Many Ada users can make do without using functions with side-effects.

8. Two-thirds of the respondents make use of recursive subprogram calls.

A copy of the questionnaire, with total numbers of responses filled in, is given in the next section.

5 The Questionnaire

I. Please answer the questions below with reference to a specific software development project that you are or have been engaged in. If you cannot answer from experience about a project involving Ada, we are still interested in any experience with a medium- to large-scale software project.

a) Roughly, what is the size of the project, in bytes ?	40 KB - 60 MB
b) To which hardware is it targetted ?	
c) In what language(s) is it written ? What fraction for each ? (or give rough numbers for lines of code)	Ada:16 FORTRAN: 2 Pascal: 1 assembly languages: 7 9000 - 500,000 lines total
d) Was a program development language (PDL) used ?	NO: 3 YES: 7 Ada: 4
e) Is the project a commercial product development, DoD contract, IR&D, or other ?	DoD: 10 IR&D: 3
f) Describe briefly the goal of the project.	

II. We are interested in estimating the potential needs for formal verification in such a project. Because formal verification is not now a common phase of software development, we would like to gauge the most likely applications for formal verification by finding the areas to which the greatest fraction of testing now goes. For each area below, if it relates to the project you described above, please indicate the relative fraction of the testing effort devoted. Feel free to add any other areas which consume significant testing resources.

Level of testing effort:	none	very little	some	very much
a) timing constraints -- verification that real-time limits are not exceeded due to computational complexity	[7]	[3]	[3]	[5]
b) space limitations -- verification that space bounds are not exceeded due to dynamic memory allocation, or stack overflow as a result of nested procedure calls or interrupt handling, etc.	[2]	[5]	[4]	[7]
c) protection of sensitive data from unauthorized disclosure	[5]	[5]	[4]	[4]
d) protection of data integrity	[2]	[4]	[6]	[5]
e) resource management	[4]	[7]	[5]	[2]
f) denial of service	[5]	[9]	[3]	[0]
g) real-time external device control with feedback	[10]	[1]	[3]	[4]
h) fault tolerance	[6]	[4]	[5]	[2]
i) floating-point numerical computations: correctness and precision	[11]	[3]	[3]	[0]
j) fixed-point or integer numerical computations	[6]	[3]	[6]	[2]
k) machine-dependent interfaces, perhaps using low-level Ada	[4]	[3]	[5]	[6]

l) parallel processing (concurrency; tasking)	[6]	[1]	[3]	[8]
m) handling of external interrupts	[4]	[1]	[9]	[4]
n) graceful recovery from errors in external input	[1]	[3]	[8]	[5]
o) graceful recovery from internal program errors -- logical design problems, hardware failures, etc.	[2]	[4]	[8]	[2]
p) independent module testing	[0]	[0]	[10]	[8]
q) integration of system modules, each independently reliable	[0]	[0]	[5]	[13]
r) operations involving complicated (e.g. nested) data types	[2]	[4]	[2]	[9]
s) portability	[5]	[4]	[3]	[6]

t) other -- please explain

- inter-process communication in a shared bus architecture
- mutual exclusion of processes using shared resources (race conditions and deadlocks)
- generics - validation of the "correctness" of a generic definition

III. The following questions are Ada-specific. We realize that there are now compilers in use which implement only a portion of Ada, or which may not implement esoteric language features in an efficient manner. Which of these Ada language constructs do you find are heavily used, avoided because your compiler is not adequate, or not used at all ?

	heavily used now	don't trust compiler	not used & not likely to be used	(*)
a) low-level Ada:				
address clauses	[6]	[2]	[4]	2
unchecked storage deallocation	[4]	[3]	[4]	2
unchecked type conversion	[5]	[3]	[4]	1
b) interfaces to other languages	[7]	[1]	[6]	1
c) generics	[3]	[6]	[4]	2
d) recursive constructs:				
types	[9]	[0]	[5]	
subprogram calls	[10]	[0]	[5]	
e) exception handling	[11]	[3]	[0]	
f) tasking,	[3]	[4]	[4]	
including, in particular:				
shared variables	[2]	[4]	[8]	
tasks passed as parameters to subprograms	[0]	[6]	[8]	
task and entry attributes	[2]	[5]	[7]	
dynamic task creation	[2]	[3]	[9]	
g) functions with side effects	[2]	[0]	[13]	
h) global variables, except in packages	[9]	[0]	[5]	
i) limited private types	[5]	[1]	[7]	1
j) subtypes of predefined integer types	[11]	[0]	[2]	
k) subtypes of predefined real types	[7]	[1]	[6]	
l) access types	[10]	[1]	[1]	

m) renaming declarations	[6]	[1]	[6]

(*) is "not available"

IV. In your experience, how commonly used are the following SUPPRESS pragmas ?

	never	rarely	some	often
ACCESS_CHECK	[9]	[1]	[1]	[2]
DISCRIMINANT_CHECK	[9]	[0]	[2]	[2]
INDEX_CHECK	[8]	[0]	[2]	[3]
LENGTH_CHECK	[8]	[0]	[2]	[3]
RANGE_CHECK	[8]	[0]	[2]	[3]
DIVISION_CHECK	[9]	[1]	[1]	[2]
OVERFLOW_CHECK	[9]	[1]	[1]	[2]
ELABORATION_CHECK	[10]	[0]	[1]	[2]
STORAGE_CHECK	[9]	[1]	[1]	[2]

V. The Ada Language Reference Manual defines certain compiler-dependent situations in the following way:

- Erroneous program: Compilers are not required to detect violations of certain semantic rules of Ada, either at run-time or compile-time. For example, the results of procedure calls should not depend on the method of parameter passing, as it might if parameters are aliased. Programs which violate these rules are called erroneous.

- Incorrect order dependence: A rule of the Ada language under which different parts of a given construct are to be executed in some order that is not defined by the language (but not executed in parallel), and execution of these parts in a different order would have a different effect. The compiler is not required to provide either a compile-time or run-time detection of a violation. An example would be evaluation of the expression "f(x) + g(y)", where, due to side-effects, the sum would depend on the order in which f and g are evaluated.

	never	rarely	some	often
a) How often have you encountered erroneous programs ?	[6]	[2]	[4]	[2]
b) How often have you encountered programs with incorrect order dependence ?	[9]	[3].	[2]	[0]

6 Further Surveying of User Needs

In the future, we will expand the questionnaire with the following:

- Somewhat more information about the respondent's project, to discover correlations between the kind of project and the use of Ada.

- A request for the respondent to indicate whether erroneous programs or incorrect order dependence were left in the final code, and if so, whether that was because the user of the compiler knows the actual semantics of the compiler in these situations.

- The circumstances under which suppress checks were used.
- Any suggestions for improvements aired at the March workshop at IDA.

We expect to receive from USAF Systems Command more contacts in the Ada world in the near future. We also intend to extend the survey beyond the limits of the Ada user's community, and to developers of non-military systems.

7 The Predictability Problem

The primary objection to Ada from the verification community which I have heard is that the language definition does not determine actual program execution, i.e. the language is unpredictable and therefore not verifiable. The so-called unpredictability is a consequence of the requirement to maximum portability which is perhaps the main goal of a standard language. The language definition does not determine those aspects of execution which a given architecture can optimize. For example, the order of evaluation of an expression is not determined (just as it is not determined in Portable Standard Lisp as contrasted with almost all other Lisps). Since user defined functions can have side effects, different orders of expression evaluation can lead to different states and thus the same program can execute differently under different validated

Ada compilers. If this occurs it is called an "incorrect order dependence". A compiler is permitted to detect such an incorrect order dependence (if it can, in general the problem is unsolvable) at either compile time or run time; in the latter case a Program_Error exception can be raised and if there is an exception handler present further execution can ensue. Thus, a given piece of legal Ada text can give rise to a large variety of executions. The general term for such programs is "erroneous" but it is not clear if the types of "erroneous programs" enumerated in the language manual exhausts all the kinds of erroneous programs. Another example, of an erroneous program is one in which a variable is read before it is written. The language does not require Program_Error to be raised but permits it. Of course a perfect example of the use of verification is to show that a program is not erroneous.

Is non-predictability really an ultimate bar to formal verfication as many critics have maintained? I think an approach to an answer is suggested by the distinction usually made between design and code verification and the realization that Ada verification is actually a species of design verification rather than code verification!

In design verification one proves properties of a system from its formal specification. The formal specification, SPEC, is really an axiomatic description of a family of systems (namely all those which satisfy the axiomatic specification SPEC) and the proof of a property Q from the formal specification, namely

SPEC => Q

is really a proof that any system in the family has property Q. Code verification is a proof that a system S (given by its program P) meets the formal specification. In this way S is shown to have property Q. The usual understanding of this process is that the program P uniquely determines the system S. In actuality, an Ada program P, just like SPEC, determines a family of systems. This is due to to unpredictability. In summary, an Ada program is a specification of a family of actual object code systems and Ada verification is a very advanced species of design verification.

Of course, this glib description doesn't solve any problems; it only presents a framework within which to proceed. As it stands now a given Ada program P has too many possible object compilations (which we will call "execution models"). This makes it difficult to develop proof rules. Attention has been centered on finding verifiable subsets of Ada which will give rise to a tractable model space for a given program. What we have found at ORA is that such a search for a predictable subset of Ada must be coupled with a definition of a "predictable compiler" which places restrictions on the Ada compiler beyond those given in the language manual. Actually, we have been finding that the restrictions to be a "predictable compiler" aren't much different from being a "reasonable compiler" since the language manual permits behavior that no reasonable compiler would ever exhibit and the exotic permitted execution models thus allowed complicate the model space and make it difficult to devise proof rules.

8 Verifiable Subsets: The Cluster Approach

As mentioned above ORA is currently examining how far existing techniques of program verification can be adapted to the verification of Ada programs. We have surveyed the current literature in verification from the point of view of Ada and are preparing an extensive annotated bibliography

As the best understood and best worked-out methods are calculi of "Hoare triples" , {P}S{Q}, those are the techniques on which we have concentrated. These seem to force on us the following limitation:

executions which raise predefined exceptions be treated like executions which fail to terminate.

That is, the triple "{P}S{Q}" is taken to mean "If P is true when the execution of S is begun and execution of S terminates without raising a predefined exception, then Q will be true when S terminates." Original research remains to be done on the raising and handling of predefined exceptions. Those which are machine independent (e.g., when an array index is out of bounds) are relatively straightfoward; the machine dependent exceptions (e.g., overflow) require the verification environment to contain either explicit machine dependent constants (such as the actual range of INTEGER) or assumptions about these contants (such as the range of INTEGER includes the standard sixteen bit two's complement signed integer range).

Our ultimate goal is to identify not "a verifiable subset" but a number of overlapping subsets of the language, each of which, individually, is reasonably tractable. We call the "allowed subsets" clusters. The user would be required to write any package, subprogram, etc., wholly within the restrictions imposed by some cluster. This is our solution to the following problem: Imagine a language with the constructs R, S, and T. Suppose that one has in hand a usable proof system for programs in which only R and S occur. It's quite possible that incorporating T into the proof system would require not only the introduction of rules for T, but also the introduction of new complications into the rules for R and S -- so that even the proofs for programs involving only R and S become more difficult. If there were a domain of problems in which it seemed unlikely that construct T would be used it would pay to distinguish the subset {R,S} as a "verifiable" cluster with its own simple proof system. For example, introducing aggregate types like records and arrays complicates the logic of procedure calls; and, more generally, handling the logic of procedure calls requires a more detailed than usual analysis of the assignment statement.

Our use of clusters is driven by the fact that Ada was not designed as an inherently verifiable language (unlike Euclid and Lucid). It is extremely rich, allowing interactions between features which surprise even Ada's designers. This is not a criticism; a programming language should be a flexible tool in the hand of a creative system builder. The introduction of clusters is the recognition that Ada contains a large variety of inherently verifiable sublanguages and that a large program contains units from many clusters.

The cluster approach is not ad hoc since it is meant to mirror Ada's ability to hide information -- using it to hide awkward combinations of constructs from the sight of one another. This strategy is a first crude step toward recognizing that in actual programs constructs are not thrown together arbitrarily but occur in contexts, and the allowed clusters are meant to be abstract representations of "contexts."

Among the advantages of thus modularizing the proof system:

-- One concrete step is immediately suggested -- namely, to study the requirements of individual problem domains (numerical computation, communications, etc.) and look for useful tractable clusters. If certain combinations of constructs that naturally hang together are not well-handled in the existing literature such combinations are obvious candidates for research.

-- Questions of technique are not prejudged -- nothing requires that different clusters be attacked by the same methods, or by methods that can easily be integrated with one another.

-- The system can be improved piecemeal -- one can introduce new clusters as research makes them tractable, or alter one of the already carved-out clusters without having to alter any of the others. Notice that there's no reason to think that the proper strategy for improving the system will always be that of extending some one or more of the the allowed clusters. It might make sense to add to the collection an additional cluster which is a proper part of one of the allowed clusters -- if that part is useful and can be handled significantly more easily than its parent, or to merge clusters if new methods of analysis are developed.

-- More generally, cluster-building tools can be provided so that the verification environment is user-extensible. This will allow the user to formulate a cluster useful to him and prove the soundness of the cluster's proof rules in terms of an abstract mathematical model of Ada which the system would contain.

The word "construct" is used above as though one knew precisely what it meant, and it may further suggest that "constructs" are indivisible things whose combinations could only be all or nothing matters. Neither suggestion is intended. Consider the following, standard, example. It's meant to illustrate two things:

- as things are added to a language the logical rules tend to complexify more rapidly than the language grows

- different logics can be used for different fragments of the language.

Example -- In a language with only scalar types a simple and well known Hoare-style rule captures the assignment statement. Once arrays are added that rule must be made more complex. Not only is this well known, but it is also well known that the uses of assignment which necessitate the new complexities are unlikely to occur in practice.

If procedure calls are now added certain implicit assumptions of the assignment axiom must be brought to light: Namely, the assumption that variables which are syntactically distinct correspond to disjoint areas of memory (i.e., are not aliased). We can say this in another way: The naive rule for the assignment assumes that syntactically distinct variables are semantically unrelated.

The assumptions show themselves as follows: The obvious way to infer the effects of a procedure call is to calculate what effects execution of the procedure's body would have if carried out on the formal parameters and then infer that it would have corresponding effects on any actual parameters with which it was called. Unfortunately, the syntactically unrelated formal parameters of the procedure may be replaced by semantically related actual parameters, in which case the effects on the actuals need not correspond to the effects on the formals. If there were no procedure calls there would be no need to express this "deeper" analysis of the assignment rule in the formal axiomatics.

Our proposal for procedure calls is that the "ordinary" logic of them should forbid aliasing among the actual parameters -- and, consequently, a naive semantics can be used to calculate the effects of the body on the formal parameters. An "extraordinary" procedure is one for which certain instances of aliasing are explicitly allowed -- and in calculating the effects of the body it is then necessary to use a more complicated logic distinguishing locations from their contents, etc.

Here is another commonplace example: Access types can be used to build complicated data types such as lists, trees, etc. If such types were encapsulated as private types in packages which exported only the algebraic operations suitable for manipulating lists and trees, then the rest of the program needn't be cognizant even of the existence of access types.

Among the conclusions of our survey:

-- Procedure calls in full generality are intractible. The only practical remedy seems to be to restrict the possibilities for making aliased procedure calls (except in certain specified instances). The difficulties in Ada go beyond the "classical" difficulties with aliasing because the parameters in Ada procedure calls are specified functionally (as in, in out, or out) rather than operationally (as var or val), making them implementation-dependent. Further, the order of copy-in and copy-out is also implementation-dependent.

-- The logic of the predefined exceptions is in general too unstructured for us to deal with at the present time. To treat them as though every statement were implicitly decorated with conditional "goto's" (the "goto" branch being taken if the exception is raised) would be combinatorially overwhelming. There are just too many "goto's." Further, any single predefined exception can be raised in many places and one can't in general be sure when an exception is raised where it was raised. Therefore one can't in general know with any precision the state of the machine at the moment the exception was raised. It is possible, of course, that the exception handler will take charge and restore the machine to some determinate state, even though the state in which it was activated is in some sense unknown. User-defined exceptions, being fewer and more specific, are much easier to accomodate.

The last point outlines a very important area of research which we are proposing to undertake. Although at present applying existing techniques to the logic of predefined exceptions is intractable the question must be faced for real embedded systems. Presumably, the software must either be verified to recover gracefully from the raising of predefined exceptions or not to raise them. It might be tractable to state an entry condition for each exception handler which is then proved to hold before any statement which could raise that exception or to prove that before each such statement the state of the machine is such that the exception will not be raised.

Appendix B

Towards Ada Verification: Preliminary Report

David Guaspari
Odyssey Research Associates, Inc.

TOWARD ADA VERIFICATION
Preliminary Report

report revised March 25, 1985

ODYSSEY RESEARCH ASSOCIATES, INC.

Table of Contents

Odyssey Research Associates

1 Introduction

1.1 Limitations

This paper is the current draft of a continuing attempt to discover and to describe, as precisely as possible, restrictions on the use of Ada which will forbid the use of features or combinations of features which are clearly beyond the capacity of current methods of program verification. Its origins as a list of do's and don't's for programmers are still evident in its current incarnation. This draft incorporates the criticisms made at the workshop on Ada verification held at IDA on May 18 - May 20.

We would welcome any comments, which can be addressed to Odyssey Research Associates, 408 East State St., Ithaca, NY 14850 or, through the arpanet, to rplatek@eclb.

By a "verification" we will mean a correctness proof which is, if not fully automated, at least machine-checkable. We limit ourselves to considering proof techniques currently available in the literature, of which the commonest are the logical calculi of "Hoare-triples": assertions of the form "If condition A holds and program P is executed [and, perhaps, further hypotheses also hold] then condition B will result." Our first approximation added two hypotheses: that execution of P terminates (systems that add this hypothesis are called systems for "partial correctness"); that no predefined exceptions are raised during execution of P.

The hypothesis that no predefined exception be raised was criticized as being unnecessarily restrictive. The original opinion and the criticisms are further explained in the discussions of exception-handling.

The hypothesis that P terminate is the most common one to make in axiomatizing sequential programming languages -- or, to be more precise, in axiomatizing that part of programming which consists in the computation of functions. But there is an important distinction between constructs or modules which are intended to terminate and those which are not. If a module is intended to terminate then its effect is reasonably describable by Hoare triples as an input-output relationship, and failure to terminate is simply a mistake. A program unit which is not intended to terminate ordinarily provides a service -- for the moment we'll call them "services." The design of an appropriate language for speaking about services is a subject of active research. A Hoare-like strategy for specifying a service is to record an invariant which holds true either at every moment of the service's life, or at all moments except those explicitly bracketed off. If this strategy is adopted it may then seem unreasonable not to follow the strategy whole hog -- making the whole logic a logic of invariance. This is also a subject of active research. We mention the problems posed by non-terminating program units simply to take note of a difficulty with the approaches surveyed in this paper.

One final question, which might well have come first: Just what are we verifying -- the logic of the source text or the behavior of the compiled code? Here we're concerned not about tne possibility of bugs in the compiler but about the variations in behavior that can result from optimizing compilers acting quite legally. The discussion, in sections 2 through 4, of initialization and undefined variables presents an example of the difficulty.

1.2 One, Two, Many Systems

This notion of verification can be further refined, and that is why it makes sense to us to speak in the plural of verifiable subsets. Correctness is not the only goal of software engineering, and it might therefore be useful to carve out a variety of "verifiable" subsets corresponding to a variety of other goals (such as modifiability and portability).

One aim of Ada is to encourage the writing of software that can be easily modified. If a verified program were revised it would, in an ideal world, also be possible to modify a verification of that program into a verification of its revised version. Certain constructs (or certain uses of them) may well be "rigid," meaning that they would make this difficult: verification of the revised program would have to start from scratch. Experience will be the final judge of which constructs are "rigid," but there are obvious candidates (non-local constrpcts such as go to).

Another reason for choosing several "verifiable" subsets is well-illustrated in the literature. Imagine a language with the constructs R, S, and T. And suppose that one has in hand a usable proof system for programs in which only R and S occur. It's quite possible that incorporating T into the proof system would require us not only to add rules for T, but also to complicate the rules for R and S -- so that even the proofs for programs involving only R and S become more difficult. For example, introducing aggregate types like records and arrays complicates the logic of procedure calls. More generally, handling the logic of procedure calls requires a more detailed than usual analysis of the assignment statement.

If there were a domain of problems (numerical algorithms, communications systems, whatever) in which it seemed unlikely that construct T would be used it would pay to set aside the subset {R,S} as a "verifiable" subset with its own simple proof system. For example, time- or space-critical applications are unlikely to use recursive subprograms. It's at least thinkable that one could verify systems using large amounts of the language by restricting each program unit to some tractable combination of constructs (and thereby hiding the difficult combinations from one another). A fancy way to say this is to say that we're hoping to pursue a strategy which is not "context free."

Among the advantages of modularizing the proof system: It immediately suggests some concrete things to do -- namely, to study the requirements of individual problem domains and look for useful tractable subsets. It doesn't prejudge any questions of technique -- nothing requires that different subsets be attacked by the same methods, or by methods that can easily be integrated with one another. Finally, the system can be improved piecewise -- one can introduce new subsets at will, or incorporate an additional construct into an existing subset without having to incorporate that construct into any others.

1.3 Predictable compilers

There is not a sharp distinction between the work to be done by a verifier and the work to be done by a compiler. Compilers may use the results of verification to help optimize their performance (for example, by suppressing certain run-time checks that are known to be always satisfied). A user may wish to rely on the compiler to enforce the dictates of the verifier -- for example, to ignore the pragma suppress_checks if the verifier has not certified this to be safe. A verifier may verify a program relative to the assumption that certain aliased procedure calls don't occur, or are reported if they do -- and may thereby wish to rely on a pragma which compels the compiler to generate code which performs the necessary checks.

As the definition of Ada leaves many (semantically consequential) details to the discretion of its implementors, it might also be useful to verify certain programs relative to a (broad) class of compilers. We therefore begin to explore this possibility. A relative verification would contain the proviso: so long as the program has been compiled on a "predictable" compiler.

Predictability might be implemented by a set of pragmas which could be invoked to call for certain run-time checks or (selectively) to suppress others, etc.

1.4 Outline

Section 2, a laundry list of restrictions on Ada, is basically an account of how to incorporate into Ada the "classical" restrictions that are currently imposed on languages designed with verification in mind. It proceeds construct by construct and rules out those constructs for which no substantial principles of verification are known. "Known" means: discoverable by a survey of the current (and straightforwardly applicable) literature. It will be seen, for example, that our restricted subset essentially limits Ada tasking to the resources of CSP (see [Hoare, 1978].) Further, it describes the sorts of information that must be supplied to a would-be verifier by the writer of the program (even if writer and verifier are the same person). In some cases we just throw up our hands.

We do not claim to have a model of the "allowed" portion of the language; nor that there are proof rules for arbitrary combinations of the "allowed" constructs; nor, a fortiori to guarantee that any program written with the "allowed" constructs and accompanied by the appropriate sorts of comments can indeed be verified from the rules in the literature. All we can say is that programs which violate these restrictions lie comfortably within the large domain of current ignorance. Section 2 follows the order of ARM (the Ada Reference Manual ANSI/MIL-STD-1815A, 1983). By definition, it contains no surprises, although it does point out that certain classical problems, such as aliasing and side-effects, are especially awkward in Ada.

Section 3 is written mainly for the non-expert: It defines "aliasing" and "side-effects", reviews the terminology of access variables and access types, the text of the ARM's account of undefined variables, etc. It also justifies our assertion that aliasing is intractable in Ada. Most of the discussion of the peculiarities of Ada is deferred to Section 4.

Section 4 contains a discussion of program errors -- "erroneous programs" and "incorrect order dependences." Programs which are "erroneous" are sensitive to semantical decisions which have deliberately been left undetermined by Ada's designers. As different implementations will settle them in different ways the behavior of such programs will be implementation-dependent.

Section 5 is an annotated bibliography. It cites, in addition to the Ada literature, several of the standard papers in program verification which were found useful in preparing this survey.

The paper concludes with an index.

2 Rules

The section numbers and names come from ARM.

ARM Chapter 1, Introduction

No restrictions.

ARM Chapter 2, Lexical Elements

No restrictions.

ARM Chapter 3, Declarations and Types

ARM 3.1 - ARM 3.2

No restrictions.

ARM 3.2.1 Object Declarations

Languages designed for the sake of verification typically guarantee (by means of default values) that undefined variables cannot occur. The alternative is to prove as part of the verification that execution of a program will not result in attempts to evaluate undefined variables. This could be especially difficult to prove in Ada when declaration and initialization are distinct: because legal optimizations could allow an error to be raised between the declaration of a variable and its initialization even if no executable text occurs between them.

Ada does not allow most types to have default values, and does not allow variables of some types (limited private types) to be initialized by their declarations. We show, below, to what extent the rule "initialize all variables" could be enforced. Such a restriction would avoid a source of program errors: A program which attempts to evaluate a scalar variable whose value is undefined or attempts to apply a predefined operator to a variable any of whose scalar subcomponents is undefined is erroneous (ARM 3.2.1, 6.2). Further discussion is included in section 4.

Case 1: No exceptions are raised.

- Any requirements about initializing variables must also be applied, *mutatis mutandis*, to the formal *out* parameters of procedures.

- A declaration of a record type may, and therefore should, provide default values for variables of that type (ARM 3.7).

- Access variables automatically have the default value <u>null</u> and therefore need no explicit initialization.

- The execution of an allocator may and therefore should initialize the object designated by the access variable being allocated. The distinction between this case and the last case is as follows. If type POINTER is access T, and x is declared to be of type pointer, then an immediate attempt to evaluate any subcomponent of x, such as x.all, will result in a constraint error (unless the subcomponent appears as prefix to an attribute [ARM 4.1]). The value null is in effect an out-of-range index. After executing new(x) the result of evaluating x.all is unpredictable, as the value of x is now in-range, but the value of x.all is undefined. In particular an error need not be raised.

- Variables of limited private type which are declared outside the package creating the type can be initialized in only two ways: The type can be given a default value (which means that it is implemented either as an access type or as a record type with a default value). The package can provide an initialization procedure, which accordingly must be invoked immediately after a declarative part in which variables of the type are declared. In this case we must also insist that the variable not appear elsewhere in the declarative part in which it is declared -- otherwise it could be a parameter in an expression used to give an initial value to some other variable (of a type which is not limited private). The interval between declaration and initialization can present a problem -- errors might be raised.

- Variables of all other types can be, and therefore would be required to be, initialized upon declaration, with the following two exceptions:

- Variables can be attached (via address clauses) to addresses which are hardware controlled and these cannot be initialized by program declarations. Further, certain addresses may have special significance for the operating system and need not be initialized by program declarations. The compiler must know which addresses are "wired" -- so that it will not raise program_error on the grounds that variables assigned to those addresses are not undefined.

- If a variable is declared in the visible part of a package, initialized by the declaration, and altered by execution of the package body, uses of the package by other program units might be sensitive to the order in which program units were elaborated. (See the example in section 4, below.) A program which is sensitive to the order of compilation contains an (ARM 1.6, 10.5). This problem could be solved by using the ELABORATE pragma to determine the order of elaboration. An alternative to this solution is to demand that a variable declared in the visible part of a package be initialized in one and only one of the following two ways: (i) when it is declared; (ii) by the package body. This would also require use of the ELABORATE pragma, to make sure that there were no other program units which tried to make use of the variable before the package body was elaborated.

<u>Case 2</u>: Taking exceptions into account.

Exceptions raised during declarative parts will cause control to be transferred out of the scope of the variables declared in that declarative part (ARM 11.4.2). None of the variables declared in that declarative part will linger as undefined entities because all of them will cease to exist.

If initialization does not occur at declaration there is in general no simple syntactical way to guarantee that it ever occurs -- an error may intervene between declaration and initialization even if no executable text occurs between declaration and initialization. (Reason: optimizations may reorder computations.) See examples in section 4.

<u>ARM 3.2.2</u>

No restrictions.

<u>ARM 3.3 Types and Subtypes</u>

Task types will be used only as templates; access types to task types will be forbidden. (See restrictions to 3.8 and 9.)

<u>ARM 3.4 - ARM 3.5.6</u>

No restrictions.

<u>ARM 3.5.7, 3.5.8 Floating Point Types, Operations</u>

The difficulties in verifying floating operations, beginning with the difficulty of stating what one means by correctness, are well known and aren't the sorts of problem to which "restrictions" are the appropriate response. We take as a beginning [Sutherland, 1984] which formalizes the following notion of the <u>logical</u> correctness of an algorithm: A program is a logically correct representation of a mathematical function if the values which it computes converge to the correct values of the function as the accuracy of the machine on which it is run increases. If a (finite) polynomial is used to compute some transcendental function such as cos, the correct logical specification of the algorithm would be that it computes that polynomial, not that it computes cos. This proposal, equally applicable (or inapplicable) to any programming language, was criticized on the grounds that the model numbers of Ada are quite carefully specified and might therefore make quantitative analysis of Ada programs tractable.

3.5.9 - 3.8

No restrictions.

ARM 3.8 Access Types

No access types to task types.

ARM 3.9

No restrictions.

ARM Chapter 4, Names and Expressions

ARM 4.1 - 4.5.1

No restrictions.

ARM 4.5.2, Relational Operators and Membership Tests

Warning: If A and B are array variables it is possible that the value of "A = B" could be true and the value of "A(2) = B(2)" at the same time false. If the indices of A range from 1 to 5 and those of B from 2 to 6, the "=" operator asks only whether the first value of A equals the first of B, etc. See also ARM 5.2.1.

ARM 4.5.3 - 4.7

No restrictions.

ARM 4.8 Allocators

If insisting on initializing variables: The execution of an allocator may and therefore should initialize the object designated by the access variable being allocated.

ARM 4.9 - 4.10

No restrictions.

ARM Chapter 5 Statements

ARM 5.1

No restrictions

ARM 5.2.1

Warning: If A and B are array variables, then after the assignment "A := B" it could still happen that, say, A(2) /= B(2). If the indices of A range from 1 to 5 and those of B from 2 to 6 the assignment will replace the first value of A by the first value of B, etc. See ARM 4.5.2.

ARM 5.3 - 5.4

No restrictions.

ARM 5.5 Loop Statments

Loops immediately raise the problem, mentioned in the introduction, of intentionally non-terminating program units. Loops which are intended to terminate are well-understood in terms of Hoare-triples: While-loops and indexed loops must be annotated by "loop invariants" -- conditions true every time an iteration of the loop begins. General loops of the form "loop S; exit when B; T" must be annotated with two invariants: one which is true whenever control reaches the beginning of S, and another which is true not only whenever control reaches the end of S but also whenever it reaches the beginning of T. Loops with more than one exit are handled by an easy generalization. Loops can also be left by executing a return statement or by the raising of an exception. These in principle present no special difficulties and [Luckham and Polak, 1980] asserts that such use of errors in "normal" circumstances has not been found to be especially burdensome.

We simply note that we know no generally satisfactory strategy for specifying what it is that one wants to prove about loops which are intended to be non-terminating. Nor is it obvious that from a collection of pieces each of which is specified by Hoare triples one can expect to assemble and verify a loop specified in some other way (e.g., by an invariant true throughout the loop's lifetime).

ARM 5.6 - 5.8

No restrictions.

ARM 5.9 Goto Statements

As Ada has powerful control structures, including "return" and "exit" statements it seems not unreasonable to say: no go to's. [Ledgard and Singer, 1982] argues that the construct is "redundant" and [Good, Young, Tripathi, 1980] that it is an "anachronism." Notice that goto's are also likely to lead to "rigid" programs (programs whose verifications cannot be easily modified to apply to modifications of the program). The standard discussion of this matter is [Knuth, 1977].

ARM Chapter 6 Subprograms

ARM 6.1 Subprogram Declarations

The body of a subprogram may not contain a declaration for another subprogram of the same name and parameter type profile. ("Parameter type profile" is defined in ARM 6.6).

Tasks or objects with tasks as subcomponents may not be passed as parameters to subprograms. This restriction is imposed not because this is known to be intractable or to present additional difficulties beyond those already posed by understanding tasking, but rather because no work has been done on the question.

Subprogram specifications must be accompanied by comments which: list the global variables occuring in the subprogram; list the allowed exceptions to the "no aliasing" rule; and describe its effects, including side effects. (This requirement is set out in more detail in the discussions of ARM 6.4 and ARM 6.5. Section 3 contains the definition of "alias.")

ARM 6.2 Formal Parameter Modes

If insisting on initializations: The formal out parameter of a procedure must be initialized at the beginning of the procedure body. (See section 4.)

ARM 6.3 Subprogram Bodies

See 6.1, 6.2.

ARM 6.4 Subprogram Calls

These restrictions concern both the suggested way to annotate subprograms with comments and the appropriate use of subprograms based on their annotations.

1. Recall the comments about ARM 6.1 - ARM 6.3.

2. No forbidden aliasing. The term "alias" is elaborately defined and discussed in section 3. Experts should beware of a subtlety: The usual proof rules for Pascal-like languages permit a val parameter to be aliased against a var parameter. The Ada parameter modes in and in out, however, do not quite correspond to val and var. Aliasing

between <u>in</u> and <u>in out</u> parameters can result in erroneous programs (see section 4).

3. This restriction could be enforced syntactically, by ruling out all <u>potential</u> instances of aliasing. Doing so would rule out many calls in which aliasing does not actually occur. Alternately, the verifier could be called on to show that aliasing does not in fact occur. The verifier would reject any program for which he could not make such a demonstration.

4. If syntactical enforcement is chosen, there are two ways in which the restrictions might be relaxed:

   - The compiler could recognize a pragma ALIAS_CHECKING which would cause it to generate code that would at run-time raise the error ALIASING_ERROR if forbidden aliasing were to occur.

   - Subprograms which had been certified by the verifier to meet their specifications for all calls, aliased or not, would be exempt. This certification could, of course, be fed back to a compiler called on to do ALIAS-CHECKING.

5. In any case, procedure specifications must be accompanied by comments which do the following:

   - list the global variables of that procedure;

   - describe the intended result of the procedure, including the side-effects of a call on it ("side-effects" are defined in this section, in the discussion of ARM 6.5);

   - describe changes or potential changes in the objects designated by <u>in</u> parameters which are access variables ("designated" is defined in section 4 in the discussion of access variables);

   - indicate which instances, if any, of otherwise forbidden aliasing are to be permitted (premissions one would expect to find mainly in already-verified library routines).

<u>ARM 6.5 Function Subprograms</u>

1. We urge that unless it is prohibitively expensive, a function body contain no global variables at all -- that it should import as parameters any non-local variables which it uses. The ideal is that all functions should act like "mathematical" functions (i.e., like the predefined

operations) -- fully describable in terms of input and output. If global variables there must be, then the function specifications must be accompanied by a comment listing them and giving an account of their role.

2. No side-effects. This means that a function body may not contain:

   - assignments to global variables or calls to procedures which can change global variables;

   - I/O operations;

   - allocators -- statements of the form "x := new T", where x is an access variable designating objects of type T;

   - occurrences of "run-time" attributes -- attributes whose values can change during execution (this seconds the general restrictions which will be imposed below on the use of attributes);

   - assignments to subcomponents of access variables, or calls to procedures which make such assignments.

These matters are elaborately discussed in the next section.

<u>ARM Chapter 7, Packages</u>

Whether or not one is insisting on initializations: If a variable is initialized in the visible part of a package neither it nor any of its subcomponents must be altered by execution of the package body. See the discussion, above, of ARM 3.2.1. Those comments also discuss the treatment of limited private types.

Except for its effects on the variables declared in the package specification, execution of the package body should have no side-effects on entities visible outside the package body. For the meaning of "side-effects" see the discussion, above, of ARM 6.5. If a main program uses a package which violates this restriction and the package is a library unit (in particular, if it is one of the parts of a program that is separately compiled) the effect of the main program could depend on when in the sequence of elaborations of library units that package body is elaborated. This would be both a logical difficulty and a program error. See the discussions of ARM 10.5 in this section and the discussion of incorrect order dependences in section 4.2, below.

Packages don't provide a problem so much as they provide an opportunity -- to modularize systems into coherent parts. The problem is the problem of not wasting the opportunity, which means finding good ways to specify packages. Any restrictions on writing packages will be for the sake of accomodating those techniques.

<u>ARM Chapter 8, Visibility Rules</u>

No restrictions

<u>ARM Chapter 9, Tasks</u>

What's known about tasking is rather limited, and the rules below are no more than an indication of the kinds of limitations that have so far been imposed to isolate tractable fragments of the language. We have not attempted to extend or to synthesize these systems. The bibliography (section 5) sets out in some detail the fragment of Ada tasking treated in each of the included papers. These papers have, of course, deliberately stripped down the language for ease of exposition and in some cases extensions (of the sequential part of the fragment) seem routine.

<u>Typical restrictions on tasking</u>

1. Parameters passed at a rendezvous are scalars. Liberalizing this seems routine.

2. The collection of tasks must be fixed, the tasks must begin together, and the tasks must themselves be sequential -- i.e., they must not create further tasks by declaration or allocation. Accordingly: a task may not be declared within a task and there can be no access types to task types.

3. Tasks may not share memory. Accordingly: tasks may not have global variables in common and may not pass access variables in a rendezvous. A warning to experts: It might seem safe if tasks altered shared variables only by passing as <u>in out</u> parameters to a third task. Such programs can still be erroneous. (See section 4.)

4. Entry calls must obey all the restrictions (against aliasing) imposed on procedure calls.

5. The attributes COUNT, CALLABLE, or TERMINATED may not be used. This restriction effectively prevents the programmer from writing his own scheduler.

6. <u>delay</u> statements may not be used. In general, the logic of the real-time features is not understood.

7. Certain formalisms also prohibit conditional entry calls selective waits which contain an <u>else</u> clause.

ARM Chapter 10, Program Structure and Compilation Issues

No restrictions. Section 4.2, below, discusses incorrect order dependences that could arise among separately compiled program units. The ELABORATE pragma could be routinely used to eliminate any indeterminacy resulting from variant orders of elaboration.

ARM Chapter 11, Exceptions

1. Predefined exceptions: Our original position was to treat programs which raised predefined exceptions as "failures" analogous to (accidentally) non-terminating programs. The reason was not theoretical, but practical -- there seemed to be too many potential occurrences of them. The criticism of this was also practical: that experience with the exception-handling mechanism of Gypsy suggested that our position was mistakenly pessimistic. No one claims to know how to deal with tasking exceptions. The standard reference, [Luckham and Polak, 1980], omits tasking exceptions altogether.

Notice that the (non-tasking) exceptions are of two kinds: storage_error and numeric_error, which are strongly implementation dependent; and program_error and constraint_error, which are not. The first kind are (sometimes regrettable but) "normal" occurrences and need not indicate that the program is logically "incorrect," while it seems reasonable to regard the raising of program and constraint errors as indicators of logical mistakes. A formal verification of a program might be expected to generate, in passing, a proof that program and constraint errors would not occur.

2. Exception-handling: There seem to be three levels of complexity in the way in which exceptions can be handled. In order of increasing complexity they are:

- exceptions are handled locally and not propagated;
- exceptions are propagated, but within their scopes;
- exceptions are propagated outside their scopes.

The most conservative simplifying restriction would be to insist that exceptions be handled locally. In any event the proof rules require the programmer to supply: an "assumption" about the state in which the handler begins to do its work; an "assertion" about the states in which the exception is raised. Actually, the rules are more complicated -- because the association of exceptions with exception-handlers is dynamic.

3. Exceptions raised during procedure calls: If execution of the body of a procedure call is broken off by the raising of an exception one can't in general know the values of the actual parameters at the moment the exception is raised -- because the parameter-passing mechanisms are not in general determined by the language. The rules for procedure calls (see [Luckham and Polak, 1980]) make some assumption about the parameter passing mechanism.

<u>ARM 11.6, Exceptions and Optimization</u>

[Cohen, 1985] observes that optimizing compilers which reorganize computations present difficult problems to verifiers. Some legal reorganizations, for example, can cause errors to be raised that would not otherwise be raised or to alter the place at which an error is raised. This presents a problem for verification which can't be solved by subsetting.

<u>ARM Chapter 12, Generic Units</u>

No restrictions are imposed on the use of a generic X beyond those imposed on the use of X. In a simple-minded sense generics present no new difficulties because one can simply attempt to verify particular instantiations. That, of course, contradicts the spirit of the enterprise, which is to create an off-the-shelf template all of whose instantiations come pre-verified as a result of one "generic" verification. Like packages, generics are not a burden on the verifier but, potentially, part of the solution.

A few remarks, in keeping with the spirit of the enterprise:

Consider a case that is easy to deal with -- a generic stack manipulator. What makes this easy is that the manipulation of stacks is a kind of algebra, with calculational rules and axioms. Verifying such a generic "algebra" requires a clear setting out of what the algebra in question is.

We require of the objects being stacked nothing other than that they be objects. We might require more -- that is, facts about the operations that have been defined on them, or about subprograms that are imported to help manipulate them. Consider a generic sorting algorithm, which sorts with respect to some given (imported) relation. One intends that the relation that's imported should be some kind of ordering. The algorithm probably won't do anything coherent unless the relation is transitive, etc. The generic specification must therefore by accompanied by comments indicating what properties the imported subprograms are intended to have.

Accordingly, it seems that the important issue is not how to prove things about generics -- one needs no new rules -- but how to specify the data structures a generic is intended to act on and what the instantiation is or does.

ARM Chapter 13 Representation Clauses and Implementation-Dependent Features

We have little to offer here beyond ignorance.

Warning: The ways in which information can be coded into interrogations of the low-level attributes can be surprising and obscure. (For example, by using the ADDRESS attribute a program could discover whether a parameter had been passed by value or by reference, and its behavior could be affected by that (ARM 13.7.2, paragraph 15).) For more discussion of this see section 4.

ARM 13.10 Unchecked Programming

It is at least possible to generate an implementation-independent proof that some application of unchecked deallocation will not result in attempts to access dangling pointers. One could impose the requirement that unchecked_deallocation not be used in the absence of such a proof. Unchecked_conversion, of course, is completely implementation-dependent.

ARM Chapter 14 Input-Output

No I/O operations in function or package bodies. (See rules for ARM 6.5 and ARM 7. See also sections 3 and 4, below.)

ARM Appendix A, Predefined Language attributes

Notice that both the task attributes -- COUNT, CALLABLE, and TERMINATED -- and the low-level attributes of chapter 13 have been disallowed.

3 General matters

This section rehearses definitions and terminology about access types, undefined variables, aliasing, side-effects, and subprograms -- and attempts to justify our assertion that aliased procedure calls are intractable in Ada. As before, section numbers of the Ada Reference Manual are prefixed by "ARM."

3.1 access types

The point of this paragraph is to rehearse some terminology from ARM and make a few fine distinctions which will be useful later. Consider:

```
type T is array(1..2) of INTEGER;
type POINTER is access T;
x,y : POINTER;     -- x and y have default access value null;
                   -- attempts to evaluate x.all raise
                   -- constraint_error
x := new T'(0,0); -- x.all is initialized to (0,0);
y := new T'(1,1);
```

The terminology of ARM is: The allocator

"x := new T'(0,0)"

creates an object, and yields, for x, an access value that designates that object. The "'(0,0)" and "'(1,1)" are initializations of x.all, and y.all, the objects designated by x and y. The default initialization of x is a special access value null, which does not designate an object. The simple-minded model of allocation is that x is assigned an address (the access value), which is the location at which the new object of type T (the object designated by x) resides. The terminology goes on: So long as x contains the same access value it is said to designate the same object, even though the object itself may change. This is reflected in the two kinds of assignment statements. Given the declarations above, the result of:

```
x := y
```

is that x and y designate the same object, because each thereafter will contain the same access value (the one originally contained by y). The result of

```
x.all := y.all
```

is that x and y designate different objects (contain different addresses) whose components happen to be the same: they're identical twins. The result of

```
x(1) := 2
```

is that x designates the same object as before -- an object that is now changed (as in, "That's the same man, but now he has a beard").

3.2 initialization

ARM 3.2.1:

- The result of an attempt to evaluate an undefined scalar variable, or to apply a predefined operator to a variable that has an undefined scalar subcomponent will be unpredictable, but need not raise an error.

- The value of a scalar variable is undefined after elaboration of the corresponding object declaration unless an initial value is assigned to the variable by an initialization (explicitly or implicitly).

ARM 6.2:

- The value of a variable is said to be updated when an assignment is performed to the variable, and also (indirectly) when the variable is used as an actual parameter of a subprogram call or entry call statement that updates its value; it is also said to be updated when one of its subcomponents is updated.

- The value of a scalar [out] parameter that is not updated [by a procedure call] is undefined upon return; the same holds for the value of a scalar subcomponent other than a discriminant.

ARM 9.10:

- If the abnormal completion of a task takes place while the task updates a variable, then the value of this variable is undefined.

The reference manual carefully avoids talk about "defined" or "undefined" or "partially defined" aggregates. No explicit definition is given of what it means for a scalar variable to be defined, other than to say that a scalar variable initialized upon declaration is defined.

3.3 aliasing

Our principal difficulties are difficulties with the parameter-passing mechanisms of Ada -- with the fact that they are often implementation-dependent. This makes aliased procedure calls even more awkward than usual, for the semantics of aliased procedure calls will as a rule become correspondingly implementation-dependent. We think it will be necessary to rule out certain aliased procedure calls, as we are becoming persuaded that, as a rule, the peculiarities of Ada make them intractable.

The rule stated below is not peculiar to Ada, except for the caution, mentioned in section 2, that _in_ and _in out_ parameters should not be aliased. The effects of Ada's underdetermined semantics are deferred to the account of erroneous progrmas in section 4.

3.3.1 definition of alias

Two distinct occurences of variables are aliases 'if they refer to common areas of storage. In particular, distinct occurences of the same variable are, trivially, aliases. (The term "alias" is often restricted to aliases which are distinct, but it will be convenient here to speak more broadly.)

We first consider the case of records and arrays. Let A be an identifier which is an array, and suppose the following subcomponent is well-formed: A(t).NEXT(j). That is, A is an array of records, and the objects occupying the record field NEXT are themselves arrays. Combining the terminology of [Cartwright and Oppen, 1981] and [Gries and Levin, 1980] we'll say that the _abstract address_ of the variable 'A(t).NEXT(j)' in some machine state S is an ordered pair whose first co-ordinate is the identifier'A' and whose second co-ordinate is the _selector sequence_ <value of t, NEXT, value of j> -- where the values of t and j are computed in state S and we may as well think of the field-name 'NEXT' as being its own value.

Let x and y be variables and let their abstract addresses be (I1,s1) and (I2,s2), respectively. Then,

x and y are _aliases_ in state S

if and only if

- I1 and I2 are the same identifier;
- one of the selector sequences s1, s2 is an initial segment of the other (when both are evaluated in state S).

Further,

x and y are _potential aliases_

if and only if

- there exists some state in which they are aliases.

Finally,

- a variable is aliased or potentially aliased with an expression if and only if it is aliased or potentially aliased with any variable which occurs in the expression.

Notice that the selector sequence of the variable A (where A is an identifier) is the empty sequence, which is an initial segment of any other sequence.

Examples:

- a(i).NEXT and a(i).NEXT(j) are guaranteed to be aliases in all states, as are a and a(i);
- a(i) and b(i) cannot be aliases (if 'a' and 'b' are distinct identifiers);
- a(j) and a(t) are aliases if and only if j = t;
- a(i) and a(i+1) cannot be aliases, nor can a(i).NEXT and a(i).LEFT.

Access variables are only seemingly more awkward, because the customary notation disguises their complete analogy with the case of records and arrays. We adopt the terminology and adapt the notation of [Luckham and Suzuki, 1979]. Suppose that type POINTER is declared as an access type to T and that x is a variable of type POINTER. We introduce a new entity, T*, of a new type (type reference class), with the following meaning: T* is a variable-length array, whose allowed indices are the values of the variables of type POINTER, and whose components are of type T. In this new notation the object designated by x, denoted in Ada by x.all, is instead denoted by T*(x). If all Ada variables are rewritten in this new notation, then the definition of aliasing used above carries through. Here is another example:

```
type T;

type POINTER is access T;

type T is
  record
    VALUE : integer;
    LEFT  : POINTER;
    RIGHT : POINTER;
  end record;

x,y : POINTER;
```

In the revised notation,

- x.all becomes T*(x);

- x.LEFT becomes T*(x).LEFT;
- x.LEFT.all becomes T*(T*(x).LEFT);
- x.LEFT.RIGHT.LEFT becomes T*(T*(T*(x).LEFT).RIGHT).LEFT.

Notice that in the last example the selector sequence is not the sequence <LEFT, RIGHT, LEFT> but a sequence of length two: <T*(T*(x).LEFT).RIGHT, LEFT>. T* is, essentially, an array of records, and its selectors can have length at most two -- the first selector being an access value and the second a field-name.

Accordingly,

- x and y are neither aliases nor potential aliases;
- x.all and y.LEFT are potential aliases, and will be aliases whenever x = y -- for this translates to the assertion that T*(x) and T*(y).LEFT are potential aliases and aliases when x = y.
- x.RIGHT and y.LEFT.all are potential aliases, and will be aliases whenever y.LEFT = x -- for this translates to the assertion that T*(x).RIGHT and T*(T*(y).LEFT) are aliased just in case x = T*(y).LEFT.

One apparent anomaly remains: Suppose that x = y. Although an assignment to x affects neither the value of y nor the value of any of the subcomponents of y, an assignment to x.all alters the object designated by y. Yet the definition above says that x.all and y are not aliased -- T*(x) is not aliased with y. The anomaly is psychological: we tend to give x no status of its own, thinking of it as another name for T*(x). If we thought of an integer variable i as another name for A(i) the same seeming anomaly would result. What this does show is that calling 'x.all' a component of 'x' can in some circumstances be misleading -- as misleading as calling 'A(i)' a component of 'i'.

3.3.2 the no aliasing rule

Unless explicitly permitted by a procedure's annotations, procedure calls with aliasing of types (i) - (iii), below, must not occur. All parameters referred to in (i) - (iii) are actual parameters. All expressions involving access variables are to be understood as written out in the notation of reference classes given above; and, using the notation of the examples above, if type POINTER is declared globally to a procedure body in which T* occurs, then the logical entity T* will be a global variable of the procedure.

(i) aliasing between parameters to a procedure call and variables which are global to the procedure;

(ii) aliasing between <u>in out</u> parameters, between <u>out</u> parameters, or between any <u>in out</u> and any <u>out</u> parameter;

(iii) aliasing between any <u>in out</u> or <u>out</u> parameter and an expression which is an <u>in</u> parameter.

Notice that aliasing between <u>in</u> parameters is acceptable. Once again we warn experts that aliasing between <u>in</u> and <u>in out</u> parameters has been ruled out.

One can determine from the text of a procedure call whether it potentially possesses aliasing of types (i) - (iii). There is therefore a simple rule which is sufficient (but not necessary) to ensure that no aliased procedure call occurs: forbid procedure calls which potentially violate the rule. Note that doing so rules out some calls in which no dangerous aliasing actually occur.

The other possibility is to attempt to prove that aliased calls do not occur -- in particular, that no potential aliasing becomes actual. A verifier would reject any program for which this attempt failed.

To flatly rule out potentially aliased calls is awkward but possible, since it is always possible to "preprocess" procedure calls in a way that guarantees that forbidden aliasing will not occur. It would then be up to the programmer to find a preprocessing that results in a program that has the desired effect. For example, if a procedure Q has two <u>out</u> parameters, representing conceptually distinct values, one must begin the preprocessing of Q(x,x) by asking why Q(x,x) is supposed to result in anything meaningful.

The simplest kind of preprocessing is a reassignment: If Q(x,...) is aliased or potentially aliased because of links between the <u>in out</u> actual parameter x and other variables, then one can declare a brand new variable y of the appropriate type and replace the code "Q(x,...)" by "y := x; Q(y,...); x := y." If several variables are so treated, then, of course, the order in which the assignments are made will matter. It is up to the programmer to decide which of these, if any, achieves the desired effect. One of the effects of this trick is to guarantee that the result of the call will be equivalent to the result of a call by copy-in/copy-out

One can sometimes guard against potentially aliased variables' becoming actual by guarding a procedure call with a conditional:

```
if x /= y then Q(a(y),a(x))
else ... something suitable
```

It's up to the programmer to decide what is suitable.

Notice that guarding a call with a conditional may be insufficient if the actual parameter a(x) is potentially aliased with a global variable (say, a(z)) of the procedure. Whether x = z at the point of call is irrelevant. What will matter is whether the value of x at the point of call equals the value of z current at certain crucial moments during the execution of the subprogram body.

3.4 subprograms

The suggested restrictions can be justified not only because they simplify the logic of subprogram calls, but because they make it possible for subprograms to be used as Ada intends that they be used -- as modules one can pull off the shelf and insert into a program without any need to know the details of their inner workings.

3.4.1 functions

The logic of function calls is simplified if the functions produce no visible external effect other than their output -- that is, if they have no side-effects. The implementation-supplied functions "+" and "*" have no side-effects, and we rely on that: not only on evaluation of "x+y" returning a correct value, but also on that act of evaluation's leaving unchanged the values of all the variables in the program. (Note: A serious practical and logical problem corresponds to the big difference between "f(x)" and "x+y" -- namely, that "f(x)" may not return a value.)

The question arises: What is a side-effect? When a function call is made: the program counter moves, the machine's clock ticks, storage fills up, the universe expands. Not everything in the world remains the same. The notion of side-effect is relative. A change is a side-effect only if there is a way in which information about that change is in some way available to the program and can therefore affect its execution. These considerations justify the restrictions on functions.

Changes in the values of variables are visible effects: Therefore, global variables may not be changed, either by assignment by calling on subprograms which change them.

An I/O operation, although not changing the values of any explicit program variables, nonetheless produces a detectable change (movement of a file pointer) -- a change detectable by the next call to that operation.

The rule which forbids updating any (or all) of the components of an access variable closes a loophole: Such updates are not, strictly speaking, updates of the variable, which continues to have the same value and designate the same object. This forbids the following sort of trick:

```
type POINTER is access T;
y : POINTER;
function F(...) ... is
   x : POINTER;
begin
   x := y;
   alter the object designated by x;
       ...
end F;
```

This, if allowed, would produce what we'd have to count as a side-effect, since the object designated by the global variable y would be changed. But there is no way to alter that object except by assignments to the dereferenced versions of x or y (including x.all) -- i.e., assignments to subcomponents of access variables.

Notice that the "no side-effects" rule means that the objects designated by access parameters passed to a function will not be changed by the call. On the other hand, objects designated by parameters passed to a procedure, even those passed as in parameters, may be changed by the call.

Allocators are forbidden in function bodies because (as ARM says) an access type implicitly brings into a being a global variable which stands for the totality of allocated objects, and a new statement updates that variable, "incrementing" it by the addition of another object. If x is a local variable of the function, then any object allocated to x is inaccessible after any execution of a call to the function is completed. Nonetheless, such an allocation may leave tracks behind: It may not be cleaned up, and could lead to a STORAGE_ERROR (ARM 11.1, paragraph 8).

Notice that merely to read a non-local variable in a function body is to allow external influence on the behavior of that function. Calls on run-time attributes also allow outside influences on the behavior of functions, and in ways that can be much harder to keep track of.

3.4.2 procedures

We provide, below, a standard illustration of the sort of awkwardness that arises as the result of aliased procedure calls and note the standard remedy. The discussion of erroneous programs in section 4 will show why the standard remedy is not necessarily helpful in Ada. Consider:

```
procedure P(x: in out INTEGER;
            y: in out INTEGER) is
begin
   x := y+1;
end P;
```

We would like to be able to reason about P by enunciating a general principle like this: If x and y are passed to P then, after the call to P, x = y+1. Unfortunately, after the call P(a,a) -- a syntactically legitimate, but aliased, call -- it would seem to "follow" that a = a+1.

The logical mistake is that the demonstration of the original principle implicitly assumed that the parameters were not aliased against one another. The standard way to correct the mistake is to verify two separate facts about P, one under the assumption that its actual parameters will be unaliased and another and another under the assumption that they're aliased. The number of cases goes up rapidly with the number of parameters and the analysis of any one case requires that one know: (a) the method of parameter passing; (b) the order of copy-out, should parameters be passed out by copy.

Further discussion of aliased procedure calls is contained in the next section (erroneous programs).

4 Erroneous Programs, Incorrect Order Dependences, Predictable Compilers

4.1 Erroneous Programs

The term "erroneous" is defined in ARM 1.6 as follows:

> The language rules specify certain rules to be obeyed by Ada programs, although there is no requirement on Ada compilers to provide either a compilation-time or a run-time detection of the violation of such rules. The errors of this category

are indicated by the use of the word erroneous to qualify the execution of the corresponding constructs. The effect of erroneous execution is unpredictable.

In effect, the compiler is allowed to make certain assumptions about the execution of the program as a basis for generating code, doing optimizations, etc. Presumably, the more ingenious a compiler is at exploiting the assumptions the more peculiar will be its possible behavior if they are false.

The rules in question occur in sections 3.2.1, 5.2, 6.2, 9.11, 10.5 11.7, 13.5, 13.10.1, and 13.10.2 of ARM.

ARM 1.6 goes on to say that:

> If a compiler is able to recognize at compilation time that a construct is erroneous or contains an incorrect order dependence, then the compiler is allowed to generate, in place of the code otherwise generated for the construct, code that raises the predefined exception PROGRAM_ERROR. Similarly, compilers are allowed to generate code that checks at run time for erroneous constructs, for incorrect order dependences, or for both. The predefine exception PROGRAM_ERROR is raised if such a check fails.

ARM 3.2.1 Object Declarations

An attempt to evaluate a scalar variable which is undefined or to apply a predefined operator to a variable that has an undefined scalar subcomponent is erroneous.

Example(i)

```
x, y : integer;
x := y;    -- erroneous, as execution of this statement
           -- requires evaluation of y
x := 0;
```

A compiler could reject this program. It could generate code that detects the erroneous step during execution and raises program_error at that point. Strictly speaking, the language manual does not even permit the inference that if the program is run without the raising of program_error the program will terminate with the value of x equal to zero.

Example(ii)

```
type BA is array (1..10) of Boolean;
x,y : BA
x := y;   -- erroneous?
```

It is not obvious whether this is erroneous or not. The question comes down to the following: Does "evaluation of y" necessarily imply evaluation of its components?

Example(iii)

The point of this example, taken from ARM 11.6, is that an error can be raised between the declaration of n and its initialization, even though no executable statement appears between them.

```
declare
   n  : integer;
begin
   n := 0;
   for J in 1 .. 10 loop
      n := n + J**A(k);  -- A and k are global variables
   end loop;
exception
   when others => PUT(n);
end;
```

ARM says that an implementation may evaluate A(k) before the assignment to n, but not before the begin (as that would associate an error in the evaluation of A(k) with a different handler). If this evaluation raises an exception the handler will attempt to PUT the value of an undefined variable.

ARM 5.2 Assignment Statement

An assignment to a variable which is a subcomponent and which depends (as a subcomponent) on the discriminants of an unconstrained record variable is erroneous if any of the discriminants of that unconstrained object is changed by the assignment. (A similar warning is issued in ARM 6.2 about producing such an effect by means of a subprogram call. See the discussion of 6.2, below.) The definition of "depending on a discriminant" can be found in ARM 3.7.1. It's illustrated in the next example.

```
type ANSWER(LENGTH: INTEGER: =  3) is
   record
      OK: STRING(1..LENGTH);
   end record;

y : ANSWER;   -- y is an unconstrained record variable
c : ANSWER(2) := (OK => "no");
```

```
   function f return STRING(1 .. 3) is
  begin
      y := c;
      return "yes";
   end function f;

   y.OK := f;     -- erroneous: y.OK is a subcomponent of y
                  -- which depends (as a subcomponent) on the
                  -- discriminants of the unconstrained record
                  -- variable y and the assignment changes the
                  -- discriminant of y
```

Here the (one and only) component OK depends on the discriminant LENGTH. The assignment of c to y is legal and changes the discriminant of y in the only legal way -- by a complete assignment to all the components of y.

<u>ARM 6.2 Formal Parameter Modes</u>

1. ARM 6.2 (paragraphs 5 and 13) says that a procedure call is erroneous if any of its <u>out</u> parameters is not updated by the call. And the updating must be done by updating the formal <u>out</u> parameter. It does not suffice to update some alias of the actual parameter. The only parameters which can be given default values are <u>in</u> parameters.

2. Paragraph 10 of ARM 6.2 says:

> If the actual parameter of a subprogram call is a subcomponent that depends on discriminants of an unconstrained record variable, then the execution of the call is erroneous if the value of any of the discriminants of the variable is changeable by this execution.

This sounds like the warning of 5.2, and will also, but for a different reason, be superfluous -- if aliasing is entirely disallowed.

Procedure calls are allowed to have side-effects, but if x.A is a parameter to a procedure call which affects x, that means that the call has either called on a function that produces side-effects or has violated the rules against aliasing.

3. Scalar and access variables must be passed by copy-in/copy-out. The method of parameter passing for parameters of array, record, or task type is up to the compiler (and need not even be the same for successive calls to the same subprogram). In neither case is the order of copy-in or copy-out specified.

> The execution of a program is erroneous if its effect depends on which mechanism is selected by the implementation. (ARM 6.2)

The word "mechanism" is to be understood broadly, so as to encompass such details as the order in which parameters are copied in or out, etc.

ARM notes a condition sufficient to rule out such erroneous programs -- namely, that

> no actual parameter of such a type is accessible by more than one path

-- i.e., that there is no aliasing. So, ARM disapproves aliasing certain parameters, and we extend that, on logical grounds, to all parameters.

Here are some examples of erroneous programs that result from aliasing. Another example is given in the discussion of shared variables (ARM 9.11).

Example (i)

Let the body of P be:

```
procedure P(x: inout INTEGER; y: inout INTEGER) is
begin
                y:= x+1;
end P;
```

The result of the call P(u,u), which violates the rule against having linked in out parameters, is unpredictable: The initial value of u (call it u0) is copied into both x and y. Executing "y:= x+1" leaves u0 in x and u0+1 in y. The result of copying both x and y back into u will depend on the order in which the copying is done.

Example (ii)

Let the body of Q be:

```
procedure Q(x: in out ARRAY(1..N) of boolean) is
begin
     x: = not x
     "Search for an i such that x(i) = b(u).  If one is found,
          x: = not x; otherwise, skip."
     end if;
end Q;
```

The procedure call Q(b) is aliased since b(u) is a free variable of Q and is linked to the actual parameter b. If b is called by copy, then b is changed by the execution of the call. If b is called by reference it's unchanged. Accordingly, the program is erroneous.

Example (iii) Let R be like Q, but with the global parameter made into an explicit in parameter:

```
procedure R(c: in boolean)
                x: in out ARRAY(1..N) of boolean) is
begin
     x: = not x
     "Search for an i such that x(i) = c.  If one is found,
          x: = not x; otherwise, skip."
     end if;
end Q;
```

In the call R(b(u),b) an in paremeter is aliased against an in out parameter. Just as in example (ii), the call is erroneous.

ARM 9.11 Shared Variables

A shared variable is one which occurs in more than one task. A program which violates either of the following restrictions is erroneous:

- If between two synchronization points of a task, this task reads a shared variable whose type is a scalar or access type, then the variable must not be updated by any other task at any time between these two points.

- If between two synchronization points of a task, this task updates a shared variable whose type is a scalar or access type, then the variable must not be either read or updated by any other task at any time between these two points.

Synchronization is defined as follows:

- Two tasks are synchronized at the start and at the end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. a task that has completed its execution is synchronized with any other task.

This series of definitions is poorly worded: taken literally they seem to imply that every point in a task is between two synchronization points (the one at the beginning of activation and the one at completion). What is presumably intended is to define something like a matched pair of synchronization points and to require exclusion during the innermost matched pair surrounding a read or update.

The point is as follows: During a rendezvous (for example) an implementation may keep a local copy of a shared variable and read and write to it rather than reading or writing the shared variable itself.

It seems worth pointing out another, perhaps surprising, way in which shared variables can lead to erroneous programs. The example below is boiled down from an example in [Welsh and Lister, 1980].

```
type boolean_array is array(1..1) of boolean;
x: boolean_array := (1 => true);

task resource is
   entry request(u: in out boolean_array);
end;

task type caller;

task body resource is
begin
   loop
      accept request(u: in out boolean_array) do
         u := not u;
      end request;
   end loop;
end resource;

task body caller is
   request(x);
end caller;

caller1, caller2 : caller;
```

Suppose caller1 and caller2 make their calls on resource at roughly the same time, so that one -- let's say it's caller1 -- gets accepted and the other is queued. The crucial point is that the execution of an entry call is begun by "any evaluations required for actual parameters in the same manner as for a subprogram call" -- (ARM 9.5) -- and only after that is the call suspended to await a corresponding accept. Suppose that the parameters are passed by copy-in/copy-out. Then the value which caller2 is waiting to pass to resource is the value (true) and the fact that the value of x will have been changed before caller2's call is accepted is irrelevant. When tasks caller1 and caller2 terminate the value of x will be (false). If the parameters are passed by reference, then caller2 passes to resource the address of x, and when caller2's call is accepted that address contains the value (false). Accordingly the value of x will be (true) when the caling tasks terminate. This dependence on the parameter passing mechanism means that the program is erroneous.

ARM 11.7 Suppressing Checks

If checks on constraints, overflows, etc., are suppressed and the constraints, etc., violated by an execution of the program, then that execution is erroneous. As indicated in the discussion of exceptions, a verification is likely to accumulate in passing enough information to show that constraint checks and checks for program errors can be safely suppressed.

ARM 13.5 Address Clauses

An address clause resulting in overlaying an object or program unit, or linking an interrupt to more than one entry is erroneous. Whether an address clause results in overlaying an object is entirely implementation dependent. Verifications of programs with address clauses are non-portable. One might verify such programs under the assumption that this error did not occur.

ARM 13.10.1 Unchecked Programming

Use of unchecked deallocation can lead to dangling pointers. An attempt to access the objects which such pointers designate is erroneous. It is easy to show that there is no algorithm for deciding whether a program is free of dangling pointers.

4.2 Incorrect Order Dependences

ARM says of certain steps in execution (or elaboration, or evaluation) that they occur "in some order that is not defined by the language" and that constructs which depend on the order in which those steps are executed are incorrect. This is principally an instruction to the writers of compilers, but the programmer can, with no outside help, produce incorrect order dependences, by producing side-effects, either in functions or in the bodies of packages. This is therefore another reason to restrict constructs which cause side-effects.

[The sections of ARM that discuss and define the incorrect order dependences: 1.6, 3.2.1, 3.5, 3.6, 4.1.2, 4.3.1, 4.3.2, 4.5, 5.2, 6.4, 10.5.]

Consider the following sequence of declarations:

```
package A is
    I: integer := 1;
end A;
```

```
package body A is
    I := 0;
end A;

with A;
package B is
    J: integer := A.I;
end B;
```

The rules for elaboration require that the specification of A be elaborated before either the body of A or the specification of B -- but require nothing further of the order of elaboration. Should the specification of B be elaborated before the body of A the value of B.J will be 1, and otherwise it will be 0.

All further points about incorrect order dependences can be made fully by looking at one further example, (ARM 3.5): When elaborating a range constraint the simple expressions specifying the bounds are evaluated "in some order not specified by the language." Let the range in question be f(m)..g(n). Here are two cases in which it will matter whether f(m) is evaluated first or second:

(a) if a call to the function g alters the value of m;

(b) if the result of a call to f can be affected by the fact of a previous call on g.

In each case the call on g has a side-effect. The kind of side effect seen in example (a) has already been ruled out by the restrictions placed on the definitions of functions. The example in (b) is probably ruled out in the same way -- that is, it's ruled out if one fleshes (b) out to an actual example in the obvious way: Let the value returned by f depend on some global variable i and let each call of g increment i by 1. But it's possible to record the fact that g has been called without storing anything in a variable. Here is an example:

Let T be a task with the single entry ENTER, whose sole action consists of the following: Accept ENTER and then terminate. Let F and G be functions with the same body:

```
x: INTEGER
begin
   if T'TERMINATED then x := 1
   else ENTER;
      x := 2;
   end if;
   return x;
end;
```

The value of F(1) - G(1) will be +1 or -1 according to whether F(1) or G(1) is evalluated first.

4.3 "Predictable" compilers

Here are some preliminary suggestions for pragmas that would a compiler keep company with a verifier. They are mainly gathered together from the preceding sections.

1. Pragmas that will identify a compilation unit as one that has been verified, allowing the compiler to suppress certain checks; and pragmas that will warn of occurrences of "dangerous" constructs which have not been certified as verified.

2. A pragma alias_check that would generate code, where appropriate, that would check at run-time for improperly aliased procedure calls and raise alias_error if such a call occurred.

3. Pragmas that would restrict the compiler's ability to raise program_error clairvoyantly. Consider the following example:

```
procedure P is
   E1, E2: execption;
   x:INTEGER

   function F(u:INTEGER) return INTEGER is
   begin
      raise E1;
   end F;

   function G(u:INTEGER) return INTEGER is
   begin
      raise E2;
   end G;

begin
   x := F(2) + G(2);
exception
   when E1 => x := 0;
   when E2 => x := 1;
end P;
```

This program terminates with the value of x equal either to 0 or to 1, depending on the order of evaluation of the terms in the expression F(2) + G(2). Since the language definition does not specify the order of evaluation (ARM 4.5, paragraph 5) "the construct is incorrect" (ARM 1.9, paragraph 9). It's not clear whether the fact that different errors can be raised is already erroneous, or whether the program is erroneousness only because the errors are handled differently. Furthermore, should the compiler detect this fact, it may generate code which does nothing but raise the predefined exception PROGRAM_ERROR. Notice that this kind of incorrect construct is possible whenever different errors can be raiseraised by different terms occurring in the same expression. It seems likely that one still might get intelligible predictions from the verifier about program behavior, in which case we would like to forbid the compiler from, in effect, rejecting the program.

5 Bibliography

Andrews, G.R. and Schneider, F.B., "Concepts and notations for concurrent programming", ACM Computing Surveys, vol. 15, no. 1, March 1983, 3-43

A useful survey, from the earliest proposals for concurrency constructs (such as fork and join) to the most recent. These constructs are organized into three general classes: procedure-oriented, in which shared variables are used for communication and the principal problems are mutual exclusion and synchronization of conditions (by semaphores, for example); message-oriented, in which communication consists of the sending and receiving of messages and the problems are synchronizing communications and designating the sender and receiver; and operation-oriented (the "remote procedure call" such as Ada's rendezvous), seen as a coming together of the other two. An extensive bibliography is included.

Apt, K.R. "Ten years of Hoare's logic: a survey -- part 1", ACM TOPLAS, vol.3, no.4, October 1981, 431-483

A systematic account relating Hoare-style axiomatics to a precise formal semantics for languages with: while, procedures (recursive procedures are included, and a variety of parameter mechanism), arrays, and 'procedures as parameters.' It is self-contained, there are no known errors, and the bibliography is extensive.

Apt, K.R., Francez, N., and deRoever W.P., "A Proof System for Communicating Sequential Processes", TOPLAS, vol.2, no.3, July 1980, 359-385

This paper contains a proof system for CSP (see [Hoare, 1979]). It is a Hoare-style logic for partial correctness which can also be used to prove programs deadlock-free. Proofs of soundness and completeness are mentioned but not provided. The separate concurrent processes can first be reasoned about individually, input-ouput Hoare-triples being proved for them from explicit assumptions about the inputs supplied by other processes at the "rendezvous"'s. Then the processes must be shown to co-operate -- to be consistent with the assumptions made about them in the first stage.

Barringer, H. and Mearns, I., "Axioms and proof rules for Ada tasks", IEE Proceedings, vol. 29 part E, no. 2, March 1982, 38-48

An adaptation of [Apt, Francez, and de Roever, 1980] to some of the tasking features of Ada. Shared variables are not allowed, nor are subprograms with side effects. Proposals are also made for extending this beyond the CSP-like features of Ada tasking: to nested accepts, delay statements, conditional entry calls, timed entry calls, selective waits with else-parts, and tasking errors. Neither formal semantics nor a soudness proof are presented or claimed.

Cartwright, R. and Oppen, D., "The logic of aliasing", Acta Informatica 15 (1981) 365-384

A formal semantics and proof system are proposed for procedure calls in a modified Pascal-like language allowing array types and aliasing, and obeying the following restrictions: 1. No functions may be passed as parameters. 2. Every global variable accessed in a procedure must be accessible at the point of every call. 3. No procedure named p may be declared within the scope of a procedure p. The paper is very difficult to read. Someone wishing to attempt it should first read [Gries and Levin, 1980]

Cohen, N., "Axiomatic semantics for Ada", talk given at the Ada verfication workshop, March 18-20, 1985, at IDA

Cook, S.A. "Soundness and completeness of an axiom system for program verification", SIAM J. Computing vol. 7, no. 1, 1978, 70-90

This paper sets out the now-standard theoretical definition of the meaning of "completeness" for Hoare-like axiom systems -- namely, that a system is complete if it is complete relative to the complete theory of the underlying domain of data types (and assuming also that the language of the underlying domain meets a certain technical condition called "sufficient expressiveness").

deBakker, J.W., Mathematical Theory of Program Correctness, Prentice-Hall, 1980

A microscopic account, all details provided, of the denotational semantics of a variety of combinations of sequential programming constructs (including: while, recursive procedure calls, blocks, go to).

Gehani, N. Ada: concurrent programming, Prentice-Hall, 1984

A lucid exposition of Ada tasking, with lots of examples.

Gerth, R. "A sound and complete Hoare axiomatization of the Ada rendesvous", Proc. 9th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 140, Springer Verlag, 1982, 252-264

An adaptation of [Apt, Francez, and de Roever, 1980] to a partial correctness logic for a fragment of Ada tasking. The fragment in question is defined precisely: the types are boolean and integer; the sequential statements are while, if, and assignment; the set of tasks is fixed and all are activated simultaneously; tasks may not have shared variables; the allowed communications statements are calls (not conditional calls), accepts, and selective waits (without else-parts or delays); there are strong "no-aliasing" restrictions on the parameters of an entry call (and therefore an assumption of call-by-copy semantics). Ada semantics are modified: there are no entry queues (a partial correctness logic is unable to distinguish between this and fairness); calling on a terminated task leads to deadlock, not an error. Soundness and completeness proofs are sketched: they proceed by translating programs from the Ada fragment into CSP programs and using the completeness result of [Apt, Francez, and de Roever, 1980].

Gerth, R. and deRoever, W.P., "A proof system for concurrent Ada programs", RUU-CS-83-2, Rijksuniversiteit Utrecht, January 1983

An extension of [Gerth, 1982] to deal with proofs of safety properties, deadlock freedom, and termination. Calling on a terminated task is now treated, as in Ada, as an error. The authors observe that they can trivially extend their system to incorporate delays, conditional and timed entries, and conditional and timed accepts for the trivial reason that the effects of such calls aren't expressible in the assertion language. The authors remark that their approach depends essentially on the assumption of a fixed number of tasks, activated simultaneously, and on forbidding queue attributes and access variables to task types.

Good, D.I. and Young, W.D., "Generics and verification in Ada", Proceedings of the ACM Sigplan Symposium on the Ada Programming Language, 1980, 123-127

The principal observation of this paper is that one can't expect to verify the behavior of generics with respect to completely arbitrary instantiations, so that there must be some mechanism for restricting the parameters of the instantiation and specifying those restrictions. One of the shortcomings the authors note has been attended to: In the final version of Ada it is possible, for example, to restrict the instantiation of a generic type solely to integer types, or solely to discrete types, etc.

Good, D.I., Young, W.D., and Tripathi, A.R., "A preliminary evaluation of verifiability in Ada", Proceedings of the 1980 Annual Conference of the ACM, 218-224

What the title suggests. The authors are commenting on preliminary Ada. The final version of Ada is, from the point of view of their criticisms, an advance on some fronts and a retreat on others.

Gries, David and Levin, Gary, "Assignment and Procedure Call Proof Rules", ACM TOPLAS, vol. 2, no. 4, 1980, 564-579

Proof rules are proposed for procedure calls in languages containing array and record types and in which: formal parameters are specified as var, result, or val; global variables are allowed in procedure bodies. Aliasing is not allowed, but specific instances of aliased calls can be accomodated by rewriting the given aliased call as an unaliased call to a related procedure. No formal semantics is provided (and therefore no soundness proof), but the intuition behind the rule is clearly presented.

Hoare, C.A.R., "An axiomatic basis for computer programming", ACM Communications, vol. 21, no. 8, 1969, 578-580, 583

The original paper on "Hoare logic."

Hoare, C.A.R., "Procedures and parameters: An axiomatic approach", in Symposium on semantics of algorithmic languages, edited by Engeler, Lecture Notes in Mathematics, volume 188, Springer-Verlag, Berlin 1971, 102-116

This paper extends [Hoare, 1969] by adding a rule for procedure calls (including recursion). The problems with aliasing are illustrated and it is observed that if the proof rule is taken as the definition of the semantics of procedure calls, then programs using only unaliased calls can be correctly implemented by any of the standard mechanisms for parameter passing. No formal semantics is provided.

Hoare, C.A.R., "Communicating sequential processes", ACM Communications, vol. 21, no. 8, 1978, 666-677

This paper proposes a construct which is the ancestor of the

Ada rendezvous. CSP is a toy language having as sequential constructs assignment, iteration, guarded alternatives. A cobegin statement may activate a fixed set of (non-nested) parallel processes simultaneously, and control may not pass beyond the cobegin until all processes have terminated. The processes may communicate only through paired input-output statements that have the effect of an Ada rendezvous in which parameters are passed but the accept body is empty.

Ichbiah, J. _et al_, "Rationale for the design of the Ada programming language", Sigplan Notices, vol. 14, no. 6. June 1979, part A

Note that many of the features of Ada discussed in this report have since been changed.

Knuth, D.E., "Structured programming with goto statements", in _Current Trends in Programming Methodology_, _vol_. _1_, R. T. Yeh, ed., Prentice-Hall, 1977, 140-194

From the paper's introduction: "This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs efficiently without go to statements; and (b) a methodology of program design, beginning with readable and correct, but possibly inefficient, programs that are systematically transferred if necessary into efficient and correct but possibly less readable code."

Ledgard, H.F. and Singer, A., "Scaling down Ada", Communications of the ACM, vol. 25, no. 2, Feb. 1982, 121-125

Luckham, D.C. and Polak, W., "Ada exception handling: an axiomatic approach", ACM TOPLAS, vol. 2, no.2, April 1980, 225-233

Proof rules are presented for Ada exception-handling which are adaptations of the standard rules for _go to_. The authors note that applying their method to the predefined exceptions requires in effect the insertion of several implicit _go to_'s after every program step -- one for each exception which could be raised by its execution -- and that this may well increase the computational costs to impractical heights. tTo deal with exceptions raised during the execution of procedures it is in general necessary to know the methods used for parameter passing. Tasking exceptions are not covered.

Luckham, D.C. and Suzuki, N., "Verification of array, record, and pointer operations in Pascal, ACM TOPLAS, vol.1 no. 2, October 1979, 226-244

Proof rules are provided for the operations of assignment, selection, dereferencing, and allocation. Extensions are proposed to the standard rules for a fragment of Pascal which would incorporate procedure calls with pointer variables as actual parameters, etc. Pointer operations are modelled on arrays, which are already well-understood. Pointer variables are though of as indices to an "array" and dereferenced pointers as the components of the "array." Allocation adds to the range of the "array"'s allowed indices. The authors refer to, but do not provide, proofs of the soundness and completeness of their rules, but it is not made clear with respect to what assertion languages. A correction to their allocation rule is noted in [Gries and Levin, 1980]. The authors note that reasoning about complex (especially: recursive) data structures requires additional notions, such as "reachability." A list of 20 axioms is provided for the notions of "reachability" and "betweenness." Sample proofs -- both proofs by hand and automated proofes -- are provided.

Luckham, D.C., von Henke, F.W., Kireg-Bruecken, B., and Owe, O., "Anna, a language for annotating Ada programs, preliminary reference manual" Stanford Computer Systems Lab technical report 84-261

A report on the most substantial effort known to us for producing a specification and verification system for Ada. Annotations are by and large generalizations of the Ada notion of "constraint." An annotation may, for example, constrain all variables of some integer type to have even values (a constraint which can't be made in Ada). The annotator can control the scopes in which annotations are to hold -- and, in particular, what is usually called an "embedded assertion" is an annotation whose scope is a single point in the text. An annotated program (possibly containing specially marked off auxiliary code, called "virtual text") is to be translated into an "Anna kernel," a new Ada program in which, for the most part, the annotations have been rewritten as embedded assertions. If, for example, an integer type is annotated as having only even values, every place in the program at which a variable of the given numeric type could be altered would be followed by an embedded assertion saying "the value of the variable is even." Such a kernel could be executed -- with the truth of the embedded assertions being tested wherever they arose -- or run through a verifier which would generate verification conditions, etc. There is no formal semantics for Anna. Instead, its semantics is defined by the reduction of Anna programs to the Anna kernel -- in effect, to Ada semantics.

Manna, Z. and Pneuli, A., "Temporal verification of concurrent programs: the temporal framework for concurrent programs", in _The Correctness Problem in Computer Science_, ed. R.S. Boyer

and J.S. Moore, Academic Press, 1981

This paper sets out a system of temporal logic, a modal logic suitable for expressing and reasoning about certain properties of ordinal (non-quantitative) discrete time. A formal semantics for temporal logic is provided and a variety of assertions are shown to be semantically valid. Execution of concurrent sequential processes is modelled by interleaving the steps in their execution, and it is then shown that many interesting properties of concurrent computations are expressible in the notation of temporal logic: invariance properties (stating that some condition always holds true), eventuality properties (stating that if condition A occurs then condition B must eventually become true), and precedence properties (stating that one event must precede another.

O'Donnell, M.J. "A critique of the foundations of Hoare-style programming logic", Communications of the ACM, Dec. 1982, vol. 25, no. 12, 927-935

This paper shows that the failure to demand a correct definition of "correctness" has filled the literature with "proof systems" which are inconsistent outright, or are unsound in the sense that the addition of true axioms can make them inconsistent. On the way to true conclusions these systems in effect indulge in a kind of trick -- intermediate inferences which are illegitimate, and lead to trouble as soon as there are enough truths available in the system to exploit their weaknesses. The correct definition of "correctness" is that every inference (and not merely every theorem) lead from truths to truths.

Olderog, E.R., "Sound and complete Hoare-like calculi based on copy rules", Acta Informatica 16 (1981), 161-197

A systematic treatment of procedure calls is given for a variety of Algol-like languages, with various scope rules, which allow procedures as parameters. The author is primarily interested in characterizing the languages for which his calculi will be complete. Although this is in some sense the fullest treatment of procedure calls it does not help solve the problems encountered in treating procedure calls in Ada.

Olderog, E.R., "Hoare's logic for programs with procedures -- what has been achieved?", in <u>Logics of Programs, 1983</u>, Lecture Notes in Computer Science no. 164, ed. E. Clarke and D. Kozen, Springer-Verlag, 1984

A survey which is emphatically not an introduction.

Pneuli, A. and deRoever, W.P., "Rendesvous with Ada -- a proof theoretical view", Proc. AdaTEC Conference on Ada, Arlington, Va., October 1982, 129-137

An operational semantics is defined for an informally described fragment of Ada, using interleaved execution to model concurrent execution. It is then shown that for any program written in this fragment and not using the queue attribute COUNT partial correctness semantics cannot distinguish between: (a) putting entry calls into a fifo queue, and (b) always selecting non-deterministically but "fairly" from the waiting calls. A system of temporal logic is defined for making assertions about programs over this semantics and various proof rules are shown sound. A program in the fragment is a block containing a fixed number of tasks. Within tasks: there may occur no subprograms or nexted blocks; there may be no delay statements; selective-wait alternatives may only be accept-alternatives or terminate.

Stanford Verification Group, "Stanford Pascal verifier user manual", STAN-CS-79-731, March, 1979

This report describes the use of the PASCAL verifier. Practically all of PASCAL is handled. "Only some of the theory [of data structures] is implemented by the simplifier and it is up to the user to include in his rulefile rules ... to express any required data structure axioms."

Sutherland, D., "Formal verification of mathematical software," NASA contract report 172407, Odyssey Research Associates, 1984

This paper presents a definition of logical correctness for floating point computation -- the "asymptotic paradigm." It says, intuitively, that a logically correct program (which computes a mathematical function) is one whose outputs converge more and more closely to the mathematically correct value if it is run on more and more accurate machines. This is formalized using non-standard models of the real line.

Wegner, P. and Smolka, S.A., "Processes, tasks, and monitors: a comparitive study of concurrent programming primitives", IEEE Transactions on Software Engineering, vol. SE-9, no. 4, July 1983, 446-462

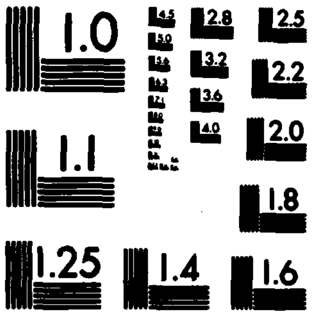
As the title indicates, CSP, Ada, and monitors are compared at work on several standard concurrent applications.

Welsh, J. and Lister, A., "A comparative study of task communication in Ada," Software Practice and Experience, vol. 11, 1980, 257-290

Ada is compared to CSP and Distributed Processes.

AD-A171 507 PROCEEDINGS OF THE IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION O..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C G ROBY DEC 85 IDA-M-146 2/5

UNCLASSIFIED IDA/HQ-85-30658 MDA903-84-C-0031 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Index

Appendix C

Formal Specification

Richard Platek
Odyssey Research Associates, Inc.

Richard Platek: Formal Specification

As an introduction to the first of three talks on ANNA, the creation of David Luckham and his collaborators and to date the only specification language for ADA, let me say a few words on formal specification.

Aside from informal comments ADA provides no way to specify a program unit's functionality. There is obviously a need to develop an ADA specification language so that one can actually say what that program unit means, so it could be used by other units. For example, a package may have two procedures and a named exception in the visible portion and there is no way of telling which of the two procedures might raise that exception. A formal specification language would be used to present the semantic interfaces between ADA program units. The interface would communicate to users the total effects of calling such units. Such effects would include the results of normal execution, the conditions on which named exceptions are propagated out of the unit, the conditions under which predefined ADA exceptions are handled within the unit and the effects of such handling, the specification of a unit's concurrency features (for example, the conditions under which rendezvous occur and what the effects are), and the effects of elaboration of package initialisation.

Another use of specification is to encourage the use of generics by providing a way of semantically restricting generic parameters. For example, the generic parameters to a generic sort package might include a user-suppled type and user-supplied linear order over that type. At present only the type-signature of the latter function can be specified. One would like to add to this a semantic specification which states that a linear order is needed. This is necessary for generic, re-usable proofs. This would communicate that the specified effects of the package's subprograms can only be expected if the user-supplied function is indeed a linear order.

Another use of a good specification language is to encourage the use of ADA as a program development language. Formal specification would play the role of informal pseudocode.

A formal specification language should support formal design verification, that is, proofs that a design entails certain system properties, and should support program verification.

There are other uses of a good specification language, e.g. for the generation of runtime monitors in the absence of formal verification, and, if essentially executable, for rapid prototyping of a system before it is actually fully encoded.

There is no way to build a library of packages without a formal specification language, because you have no way of knowing what packages mean. My vision is of a large library of formally specified and verified programs and means to retrieve what is needed from that library using the formal specification of the packages. Such a library retrieval system must have a great deal of knowledge built into it. If I am looking for a package which has property A and some package which has been proved correct has property B, the retrieval mechanism should allow me to prove that B implies A, so that I can get the verified package out to be able to use it.

Obviously such a language should be fully compatible with ADA, it should use the ADA type philosophy, and ADA structuring. It is my belief that taking existing specification languages like INA JO or revised SPECIAL or anything like them and trying to retrofit them to ADA is not the way to go. One should actually develop a formal specification language that has the ADA philosophy embedded in it.

I feel a standard should be chosen. Just as a standard was chosen for an ADA programming language, a standard should be chosen for an ADA specification language; and I would personally recommend going through a cycle just like the choice of the programming language.

I view ANNA as a very good first beginning in that direction. I don't think it goes far enough. It has certain restrictions to a low level that I think a specification language should not have.

Perhaps I should introduce rather than undermine the next speaker. I think we all owe David Luckham a great deal for beginning this work and giving us something to chew on.

Appendix D

ANNA: A Specification Language

David Luckham
Stanford University

David Luckham: ANNA, a Specification Language for ADA

ANNA is a proposal for a specification language, or rather a language in which one might experiment with specification languages. The work was begun by Bernd Krieg-Brueckner and myself, and subsequent collaborators have been O. Owe from Oslo, who worked on the axiomatic semantics, and Friedrich von Henke who worked on the language reference manual and redesigned some of the finer points of the language. S. Sankar, D. Rosenblum, R. Neff, and D. Bryan are currently implementing various prototype tools for experimentation.

ANNA is an syntactic extension of ADA: it takes a subset of ADA productions and adds more. The ANNA specifications appear as formal ADA comments. This means ANNA comments can be processed by a standard ADA tool, which will simply ignore them, and also by special ANNA tools.

All proposed ANNA tools use an extension of DIANA, and therefore can be interfaced easily with other tools in an ADA environment.

ANNA can be used for comparative testing. Comparative testing means comparing the ADA code against its formal specifications for consistency. Self-checking programs are ones which leave the runtime checks compiled from the formal specifications in the program permanently.

If you are going to design a specification language, there are at least two approaches. One is the fresh start; the other is the evolutionary approach. The fresh start has the advantage that you do not have to put up with the quirks of the programming language such as those discussed in D. Guaspari's lecture. The evolutionary approach is to start with an existing high level programming language and to extend it gradually to allow the program to supply information that cannot be expresssed in the programming language itself. The general philosophy has been one of cautious extension which is why critics will say that it does not go far enough.

There are two kinds of formal comments: virtual text and annotations. Webster's Dictionary defines the word "virtual" as "possessing all of the properties but not accepted or recognised".

The scoping rules of ADA are applied to formal comments so that the formal comments apply over regions of text; they are not just assertions which apply at a point.

There are different kinds of annotations for the different kinds of ADA constructs. Here is a reasonably complete list:

- objects,
- types and subtypes,
- statements,
- subprograms,
- packages,
- exceptions,
- context clauses, and
- generics.

I am going to concentrate on packages today. I'm going to tell you just enough about the others so that you can look at some package specifications.

Let us start very briefly with virtual text. Consider the time-honoured example, the standard stack package with two procedures PUSH and POP. For some reason the ADA implementers did not want to provide a function LENGTH, but it turns out to be a very natural concept to use in talking about the stack, so we introduce it as a virtual function declaration. This virtual function is visible in the normal ADA scope of visibility, but only in annotations.

Now the expressions in ANNA are somewhat richer than they are in ADA in that you have quantifiers, conditional expressions, a few new operators and a few new attributes. Each of the annotations is constructed from ANNA boolean expressions and some extra reserved words.

Here is an example of an object annotation:

```
m, n : INTEGER; --| n <= f(m)
```

What it means is that in every observable state during a computation in this scope n must be less than or equal to f(m). The function f is some previously defined virtual function. Now you need to be precise about what you mean by an observable state. During a simple statement a constraint does not have to hold. After a simple statement completes there is an observable state in which that constraint must then hold. Let me emphasize that this is not an assertion but an object constraint; it holds over the whole scope; every time m and n are updated the constraint must be checked.

There are two parallel definitions of formal semantics: one is checking semantics, which tells you what you have to do to check each of these annotations; and the other is axiomatic semantics, which tells you what the correct rules of proof are, to prove consistency between the text and the annotations.

Exception annotations are useful for describing the state of affairs after an execption has been raised. Consider using a table managing package which may raise the error TABLE_FULL. What has happened to the items that have been inserted previously? Can the package be used further? Other exceptions might be raised which reveal the implementation.

When you look at the structure of a package you can put annotations at various places in it. The annotations in the visible part are visible annotations and the ones in private or body parts are hidden. It turns out that a lot of packages can be specified using the previous kinds of annotations, e.g. type, subprogram, exceptions, and procedures.

Consider a string conversion package. [See slides 17-20.] The main problem is that SHORT_INTEGER is implementation-dependent so you do not know how many characters can be stuffed into a short integer. You might like to know what PACK and UNPACK actually did, because if you call them with parameters that have the wrong lengths what is going to happen? Will an exception be raised, will the packed string be truncated, will some of the characters be lost? How can we annotate this behaviour?

Probably the first thing is to write some comments in English. What do you do next? You might still be inept at writing a general specification, so you might just try writing a few test cases; you can express these cases on some inputs and output in ANNA. I claim that by the time you have the test cases formalised you are in a position to write the fully formal specification.

The fully formal specification says everything you need to know. You can check the implementation against the specification.

There are two other concepts that seem to be required in one form or another to specify all kinds of packages. The first is ate Packages can have a memory or value and their behaviour depends on that memory or value which we call state.

States conceptually are a new kind of value associated with the package, and you cannot see any of their structure from the outside. They behave just like a limited private type exported by the package itself.

There is a state type, a type of the states of the package, and that is an attribute of the package in ANNA. Then there are some important states, the initial state and the current state, and these are attributes of the package, since the current state is used so often we allow you to contract notation and just use the package name.

The other thing that you can do is declare the package axioms in the visible part of a package. [See slide 24.] Their semantics is a little different. The normal statement (without "axiom" there) is just a constraint that would have to be true at that position. If you add "axiom" this becomes a promise to the outside user that this is true when you use the package and it is a constraint on the implementation in the body. Some properties are more easily expressed in axioms and some in subprogram annotations.

I am going to end this talk today by showing you something that could be taken as a package specification in a reasonable state for negotiation, the DIRECT_IO package from Chapter 14 of the ADA Reference Manual.

Chapter 14 is an attempt to specify a standard I/O environment for ADA. It begins with a preamble to define what a direct access file is: really a linear sequence of elements, it is set-theoretical and has an index. External files are machine dependent things.

The English explanation of the DIRECT_IO facilities talks about the concept of a new file and hopes that you understand what a new file is. It talks about creating, severing and deleting files, and it assumes you know what that means.

Following this English explanation, there then come several paragraphs for each subprogram saying what sorts of exceptions it propagates under what kinds of conditions, all in English. At the end we get the ADA package specifications. So this is a tacit admission that ADA by itself does not specify a package for you but you have got to explain it in formal English somehow first.

I have gone from that formal English to a formal ANNA specification. [See slides 28-32.] We can negotiate about its correctness, and such negotiations are still going on in Language Standard committees.

One of the things I can do in using this as a medium of negotiation is to automate the drawing of consequences or conclusions. It is not hard, a PROLOG program can be written to do it. I claim that not only can we write the specifications in ANNA but we can automate the negotiation process of asking questions about the consequences of the specifications.

The notation of ANNA needs improvement, but the major omission in ANNA is tasking. It is just not practical to go into full temporal logic: what ideas we have are not mature enough to be presented.

ANNA

A Specification Language for Ada

David Luckham

Collaborators:	B. Krieg-Bruckner,	O. Owe,
	F. Von Henke	
Implementation:	S. Sankar,	D. Rosenblum,
	R. Neff,	D. Bryan

Computer Systems Lab
Stanford University

* project sponsored by DARPA

* ANNA is an extension of Ada

  ** Machine processable specifications together with underlying Ada text form an Anna program.

  ** Anna specifications appear as (formal) Ada comments.

     Anna programs can be processed by standard Ada tools.

     Can also be processed by special Anna tools.

* All proposed Anna tools use an extension of DIANA and can be interfaced easily with other tools in an Ada environment.

---

* MOTIVATION for ANNA:

  ** To permit precise machine-processable specifications and documentation to be supplied with an Ada program.

  ** To investigate the possible applications of formal specifications.

     Specification prior to full implementation:

        Checking of Designs
        Rapid Prototyping

     Annotation of complete Ada programs

        Comparative testing (debug)
        Self-Checking programs
        Instrumentation (e.g. Simulations)
        Formal Verification

DESIGN APPROACHES FOR SPECIFICATION LANGUAGES:

(1)	The fresh start	e.g., PROLOG.
(2)	Evolutionary	An existing high level programming language is extended.

ANNA design can be applied to other Languages:

MODULA-2
VHDL (VHSIC Hardware Design Language)

Design Philosophy: Cautious extension of Ada

---

* Two kinds of formal comments in Anna:

Virtual text	--:
Annotations	--|

* Formal comments apply over regions of the program

  They obey the standard Ada scope and visibility rules.

* Different kinds of annotations apply to the different kinds of Ada constructs:

  Object annotations
  Type or Subtype
  Statement
  Subprogram
  Package
  Exception
  Context
  Generic Units

* There is no assumption that Anna specifications are "complete" - the programmer can specify what he wants to.

EXAMPLE OF VIRTUAL TEXT

```
package STACK is

     --: function LENGTH return NATURAL;

         procedure PUSH (X : in ITEM);
         --|  where in STACK.LENGTH < MAX,
         --|      out (STACK.LENGTH = in STACK.LENGTH + 1);

         procedure POP (X : out ITEM);

end STACK;
```

LENGTH is used to specify PUSH. It is not an actual operation of STACK. It is a specification concept. LENGTH can be given a virtual body, and used to check the correctness of PUSH at runtime.

RESTRICTION: * VIRTUAL TEXT must not change the values of actual objects -- read only.

* VIRTUAL TEXT must not hide actual entities.

Anna Boolean expressions are a small extension of the Ada Boolean expressions.

Quantifiers
Conditional expressions
A few new operators, relations, and tests
A few new attributes

* Annotations are constructed from

  * Anna Boolean expressions

  * Reserved Words

EXAMPLES OF ANNOTATIONS

* Object annotation

```
    M, N : INTEGER;      --|  N < F(M);
```

The VALUE of M and N must satisfy the annotation in every observable computation state in the scope of the declaration.

* Type annotation

```
    type INTERVAL is
         record
              LEFT_END, RIGHT_END : REAL;
         end record;
    --|  where I:INTERVAL =>
    --|       I.LEFT_END <= I.RIGHT_END;
```

All INTERVALs must satisfy the annotation.

*   Statement annotation

```
     if A(X) > A(X+1) then
             Y := A(X);
          A(X) := A(X+1);
        A(X+1) := Y;
     end if;
     --|  A(X) < A (X+1);
```

Simple assertion. Statement annotations may also be given for invariants over compound statements.

*   Subprogram annotation

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

includes: an in-annotation on A,
an out-annotation on POSITION
a propagation annotation for exception NOT_FOUND.

*   Propagation Annotations   Two kinds: Strong
                                                          Weak

```
     package TABLE_MANAGER is

          type ITEM is ...

          ...

          procedure INSERT (NEW_ITEM : in ITEM);

          procedure RETRIEVE (FIRST_ITEM : out ITEM);

          TABLE_FULL : exception;
          -- raised by INSERT when table full (*)

     end;
```

(*) could have said "when FREE_LIST_EMPTY" !

```
     package TABLE_MANAGER is

          type ITEM is ...

          ...

     --: function FULL return BOOLEAN;

          TABLE_FULL : exception;

          procedure INSERT (NEW_ITEM : in ITEM);
     --|  where
     --|  in TABLE_MANAGER.FULL => raise TABLE_FULL,  (1)
     --|  raise TABLE_FULL =>
     --|       TABLE_MANAGER = in TABLE_MANAGER;      (2)

          procedure RETRIEVE (FIRST_ITEM : out ITEM);

     end;
```

(1) is a strong annotation, (2) is a weak annotation.

*   Context Annotations

**   Apply to program units (not just library units)

**   Used to declare dependency on global variables:

```
          A, B, C : T;

     --|  limited to A;
          package P is
               ...          -- A may occur here.
          end P;

          D : T;

          package body D is
               ...          -- A and D may occur here.
          end P;
```

*   Used to restrict context:

```
          with U, V, W;
          --|  limited to U.A, V;

          Package P is
               ...          -- any visible variable of V,
               ...          -- but only A from U, and
               ...          -- no variables of W.
          end P;
```

* Package Annotations

  ** Placed in the Package:

```
package P is
    ...          -- VISIBLE annotations
private
    ...          -- HIDDEN annotations
end P;

package body P is
    ...          -- HIDDEN annotations
end P;
```

  ** The private part and body together are the HIDDEN implementation.

  ** VISIBLE annotations specify the semantics of visible types and subprograms INDEPENDENTLY of (and prior to) any body.

  ** HIDDEN annotations specify the (implementation in the) body.

---

* Previous kinds of annotations (types, subprograms, exceptions) are sufficient for specifying many packages.

*   New concepts are required for specifying (the visible parts of) packages:

**   package has a memory (STATE) QUEUES, STACKS, SYMBOL TABLES.

**   package and private type define a data structure and operations on the data (algebraic specifications) - COMPLEX NUMBERS, LIST PROCESSING.

*   Package states

From the outside, a package is viewed as an object of some new Anna type having a state (or value).

**   Anna attributes of a package:

* STATE TYPE	P'TYPE
* INITIAL STATE	P'INITIAL
* CURRENT STATE	P'STATE, P

**   Successor states

New states of a package result from sequences of package operations.

Terms in Anna written as sequences

```
STACK [ PUSH (A); POP (Y) ] = STACK
```

*   Package axioms

**   Visible annotations,

```
axiom A;
```

**   promises to the package user

**   constraints on the package body

Example of a Symbol Table Package

```
generic
    type TOKEN is private;
    N : INTEGER;
package SYMTAB is

    OVERFLOW, UNDEFINED : exception;

    --: function SIZE return INTEGER range 0 .. N;
    --: function "=" (SS, TT : SYMTAB'TYPE) return BOOLEAN;

    function DEFINED (S : STRING) return BOOLEAN;

    procedure INSERT (S : STRING; I : TOKEN);
    --| raise OVERFLOW => in SYMTAB.SIZE = N;

    function LOOKUP (S : STRING) return TOKEN;
    --| raise UNDEFINED;

    procedure ENTERBLOCK;

    procedure LEAVEBLOCK;

--| axiom
--| for all SS : SYMTAB'TYPE; S, T : STRING; I : TOKEN =>
--| SYMTAB'INITIAL [LEAVEBLOCK]      = SYMTAB'INITIAL,
--| SYMTAB'INITIAL.DEFINED (S)       = FALSE,
--| SS [ENTERBLOCK; LEAVEBLOCK]      = SS,
--| SS [ENTERBLOCK].DEFINED (S)      = FALSE,
--| SS [ENTERBLOCK].LOOKUP (S)       = SS.LOOKUP(S),
--| SS [INSERT (S, I); LEAVEBLOCK]   = SS[LEAVEBLOCK],
--| SS [INSERT (S, I)].DEFINED (T)   =
--|     if S = T then TRUE
--|     else SS.DEFINED (T) end if;
--| SS [INSERT (S, I)].LOOKUP (T)    =
--|     if S = T then I else SS.LOOKUP (T) end if;

end SYMTAB;
```

Example: Ada specification of a string conversion package.

```
package STRING_CONVERSION is

    type PACKED_STRING is array(INTEGER range <>) of SHORT_INTEGER;

    procedure PACK_STRING(S : STRING; BUFFER :out PACKED_STRING);
    procedure UNPACK_STRING(BUFFER : PACKED_STRING; S : out STRING);

end STRING_CONVERSION;
```

Example: Ada specification of a string conversion package with comments.

```
package STRING_CONVERSION is

    type PACKED_STRING is array(INTEGER range <>) of SHORT_INTEGER;

    -- a short integer is represented as two bytes

    procedure PACK_STRING(S : STRING; BUFFER :out PACKED_STRING);
    -- packs two consecutive characters of S in each short integer of BUFFER

    procedure UNPACK_STRING(BUFFER : PACKED_STRING; S : out STRING);
    -- expands each short integer in BUFFER into two characters and puts them

    -- in S PARAMETER_LENGTH_ERROR exception is propagated if
    -- parameter lengths don't match.

end STRING_CONVERSION;
```

Example: String conversion package declaration with formal test cases.

```
package STRING_CONVERSION is

    -- A short integer is represented as two bytes.

    type PACKED_STRING is array (INTEGER range <>) of SHORT_INTEGER;

    procedure PACK_STRING(S : STRING; BUFFER :out PACKED_STRING);
    -- Assume S'FIRST = 1 and BUFFER'FIRST = 1 to simplify equations.
    -- Then the following are intended test cases:
--| where out (if S = "0123" then
--|                 BUFFER(1) = CHARACTER'POS('0') * 256
--|                                               + CHARACTER'POS('1') and
--|                 BUFFER(2) = CHARACTER'POS('2') * 256
--|                                                    + CHARACTER'POS('3')
--|           end if and
--|                 BUFFER'LENGTH = 2),
--|       out (if S = "abcde" then
--|                 BUFFER(1) = CHARACTER'POS('a') * 256
--|                                               + CHARACTER'POS('b') and
--|                 BUFFER(2) = CHARACTER'POS('c') * 256
--|                                               + CHARACTER'POS('d') and
--|                 BUFFER(3) = CHARACTER'POS('e') * 256
--|           end if and
--|                 BUFFER'LENGTH = 3);

   procedure UNPACK_STRING(BUFFER : PACKED_STRING; S : out STRING);
        . . .  -- Similar annotations of test cases.

   -- PARAMETER_LENGTH_ERROR exception is propagated if parameter
   -- lengths don't match.

end STRING_CONVERSION;
```

Example: Anna specification of a string conversion package.

```
package STRING_CONVERSION is

--| for all X : SHORT_INTEGER => X'SIZE = 16;

    type PACKED_STRING is array (INTEGER range <>) of SHORT_INTEGER;

    PARAMETER_LENGTH_ERROR : exception;

    procedure PACK_STRING(S : STRING; BUFFER :out PACKED_STRING);
--|     where
--|         in BUFFER'LENGTH = (S'LENGTH / 2) + (S'LENGTH mod 2),
--|         in S'FIRST = 1 and in BUFFER'FIRST = 1,
--|         out (if S'LENGTH mod 2 = 0 then
--|                 for all N : BUFFER'FIRST .. BUFFER'LAST  =>
--|                     BUFFER(N) = CHARACTER'POS(S(N * 2 - 1)) * 256
--|                                                 + CHARACTER'POS(S(N * 2))
--|             else
--|                 for all N : BUFFER'FIRST .. BUFFER'LAST - 1 =>
--|                     BUFFER(N) = CHARACTER'POS(S(N * 2 - 1)) * 256
--|                                                 + CHARACTER'POS(S(N * 2))
--|                 and
--|                 BUFFER(BUFFER'LAST) = CHARACTER'POS(S(S'LAST))*256
--|             end if),
--|         raise PARAMETER_LENGTH_ERROR;

    procedure UNPACK_STRING(BUFFER : PACKED_STRING; S : out STRING);
--|     where
--|         S'LENGTH = BUFFER'LENGTH * 2,
--|         in S'FIRST = 1 and in BUFFER'FIRST = 1,
--|         out (for all N : S'RANGE =>
--|             if N mod 2 = 1 then
--|                 CHARACTER'POS(S(N)) = (BUFFER(N/2 + N mod 2) -
--|                     BUFFER(N/2 + N mod 2) rem 256) / 256
--|             else
--|                 CHARACTER'POS(S(N)) = BUFFER(N/2+N mod 2) rem 256
--|             end if),
--|             raise PARAMETER_LENGTH_ERROR;

end STRING_CONVERSION;
```

Example: TABLE_MANAGER package from the Ada Rationale

```
package TABLE_MANAGER is

    type ITEM is ...
    procedure INSERT(NEW_ITEM     : in ITEM);
    procedure RETRIEVE(FIRST_ITEM : out ITEM);
    TABLE_FULL : exception;  -- Raised by INSERT when table is full.

end TABLE_MANAGER;
```

Example: Specification of the exception propagation in the TABLE_MANAGER package.

```
 package TABLE_MANAGER is

     type ITEM is ...
     TABLE_FULL : exception;

--: function FULL return BOOLEAN;

    procedure INSERT(NEW_ITEM : in ITEM);
--|    where
--|        in TABLE_MANAGER.FULL => raise TABLE_FULL,
--|        raise TABLE_FULL => TABLE_MANAGER = in TABLE_MANAGER;

    procedure RETRIEVE(FIRST_ITEM : out ITEM);

end TABLE_MANAGER;
```

Example: Specification of the TABLE_MANAGER package.

```
package TABLE_MANAGER is

    subtype DATA is STRING (1 .. 10);
    type PRIORITY is range 0 .. 255;

    type ITEM is
        record
            D : DATA;
            P : PRIORITY;
        end record;

    TABLE_FULL : exception;

--: function FULL return BOOLEAN;
--: function MEMBER (X : ITEM) return BOOLEAN;

    procedure INSERT(NEW_ITEM : in ITEM);
--|     where
--|         in TABLE_MANAGER.FULL => raise TABLE_FULL,
--|         raise TABLE_FULL => TABLE_MANAGER = in TABLE_MANAGER,
--|         in NEW_ITEM.PRIORITY'DEFINED,
--|         out ( not in TABLE_MANAGER.MEMBER (NEW_ITEM) ->
--|                         TABLE_MANAGER.MEMBER (NEW_ITEM)   ),
--|         out ( in TABLE_MANAGER.MEMBER (NEW_ITEM) ->
--|                   TABLE_MANAGER = in TABLE_MANAGER );

    procedure RETRIEVE(FIRST_ITEM : out ITEM);
--| where
--|         out( not TABLE_MANAGER.MEMBER (FIRST_ITEM) ),
--|         out( for all X : ITEM => TABLE_MANAGER.MEMBER (X) ->
--|                   X.PRIORITY >= FIRST_ITEM.PRIORITY ) ;

end TABLE_MANAGER;
```

Example: Axiomatic annotations for an integer package.

```
package INTEGERS is

    type INTEGER is  -- Implementation_defined;

    -- The predefined operators:
    function "=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    function "+" (LEFT, RIGHT : INTEGER) return INTEGER;
    . . .

    --| axiom
    --| for all A, B, N : INTEGER =>
    --|     A mod B = (A + N * B) mod B,
    --|     A = (A / B) * B + (A rem B),
    --|     (-A) / B = -(A / B),
    --|     A / (-B) = -(A / B),
    --|     A rem (-B) = A rem B,
    --|     (-A) rem B = -(A rem B),
    --|     ... ;

end INTEGERS;
```

PARNAS : A RATIONAL DESIGN PROCESS

HOW AND WHY TO FAKE IT

STEP C : DESIGN and DOCUMENT MODULE INTERFACES

* PRECISE INTERFACES MUST be specified for each module.

* It MUST be FORMAL and provide a black box picture of each module.

* It is written by a SENIOR DESIGNER and reviewed by both the future implementors and programmers who will use the module.

* An INTERFACE SPECIFICATION contains just enough information for another programmer to use the module, and NO MORE.

CONTENTS: (1) Ada Package visible part

(2) Externally-visible effects of subprograms

(3) Timing Constraints

(4) Definition of undesired events

FAKING IT :

* Produced by a process of negotiation

* The resulting documentation is not easy or relaxing reading

* Acts as an accurate reference manual

A PL/1 STRING MANIPULATION PACKAGE

Our first example is the specification of a package that provides operations on strings similar to the string manipulation facilites in PL/1. These operations can all be described using the standard Ada string operations and attributes. There are no special concepts outside the Ada domain of strings, so the package should be easily understood from a quick reading of its specification. Two virtual functions are used to name expressions occuring frequently in the annotations; the virtual definitions enable us to shorten the annotations. In fact, one of the virtual functions, SLENGTH, duplicates an actual function, LENGTH, and its only used is to make some annotations a little clearer. The package has a trivial state " it does not store any values. This example was suggested by Paul Reilly [REIL84A].

Example: A PL/1 string manipulation package.

```
package PL1_STRINGS is

-- Virtual functions naming commonly used expressions.

--: function SLENGTH(STR : STRING) return NATURAL;
--|     where return LEN : NATURAL =>
--|         (for all I : NATURAL => (I in STR'FIRST .. STR'FIRST + LEN-1
--|             -> STR(I) /= ASCII.NUL)) and STR(STR'FIRST + LEN) = ASCII.NUL;

--: function SLAST(STR : STRING) return NATURAL;
--|     where return I : NATURAL => I = SLENGTH(STR) + STR'FIRST -1);


-- Actual subprograms:

   procedure NUL(STR : out STRING);
      -- Create an empty string.
      --| where out (for all I : STR'RANGE -> STR(I) = ASCII.NUL);

   function IS_EMPTY(STR : in STRING) return BOOLEAN;
      -- Indicate whether or not a string is empty
      --| where return STR(STR'FIRST) = ASCII.NUL;

   function LENGTH(STR : in STRING) return NATURAL;
      -- Return the length of a string.
      --| where return SLENGTH(STR);

   procedure ASSIGN(TARGET : out STRING;
                    SOURCE :  in STRING);
      -- Assign Source to Target
      --| where
      --|     out (TARGET(TARGET'FIRST .. TARGET'FIRST + SLENGTH(SOURCE)-1) =
      --|                        in SOURCE(SOURCE'FIRST .. SLAST(SOURCE)),
      --|     out (TARGET(TARGET'FIRST + SLENGTH(SOURCE)) = ASCII.NUL),
      --|     SLENGTH(SOURCE) > TARGET'LENGTH => raise CONSTRAINT_ERROR;

   function CATENATE(LEFT, RIGHT : STRING) return STRING;
      -- Return the catenation of left followed by right
      --| where return STR : STRING =>
      --|     STR(1 .. SLENGTH(LEFT)) =
    --|         LEFT(LEFT'FIRST .. SLAST(LEFT)) and
    --|     STR(SLENGTH(LEFT) + 1 .. SLENGTH(LEFT) + SLENGTH(RIGHT)) =
    --|         RIGHT(RIGHT'FIRST .. SLAST(RIGHT));

function EQUAL(LEFT, RIGHT : STRING) return BOOLEAN;
    -- Indicate if string LEFT matches string RIGHT.
    --| where return SLENGTH(LEFT) = SLENGTH(RIGHT) and then
    --|     LEFT( LEFT'FIRST .. SLENGTH(LEFT)) = RIGHT( RIGHT'FIRST .. SLENGTH(RIGHT))

function INDEX(BASE_STRING, FRAGMENT : STRING) return NATURAL;
    -- Return the starting position in BASE_STRING where FRAGMENT is found;
    -- return 0 otherwise.
    --| where return I : NATURAL =>
    --|     I /= 0 and BASE_STRING(I .. I + SLENGTH(FRAGMENT) -1) =
    --|                         FRAGMENT(FRAGMENT'FIRST .. SLAST(FRAGMENT))
    --|     or
    --|     I = 0 and (for all J : BASE_STRING'FIRST ..
    --|                             SLAST(BASE_STRING)-SLENGTH(FRAGMENT) =>
    --|                     BASE_STRING(J .. J + SLENGTH(FRAGMENT) -1) /=
    --|                     FRAGMENT(FRAGMENT'FIRST .. SLAST(FRAGMENT)) );

end PL1_STRINGS;
```

Commentary
This specification depends only on Ada concepts ˜ and quantification over constrained ranges, which is really just a for loop test. So we should be able to analyse it to see if it fits our understanding of PL/1. One way to do this is to execute the specifications symbolically on small test cases. Consider,

```
        A : STRING(1 .. 3) := (1 => 'A', 2 => 'B', 3 => ASCII.NUL);
        B : STRING(1 .. 5) := (1 .. 3 => 'F', others => ASCII.NUL);
    begin
        PL1_STRINGS.ASSIGN(B, A);
        --| B(1 .. 2) = A(1 .. 2),
        --| B(3) = 'F',
        --| B(4) = ASCII.NUL;
    end;
```

We can deduce that SLENGTH(A) is 2 and SLENGTH(B) is 3. But take care. This should be done mechanically by trying each value of *LEN* in the specification of *SLENGTH*, starting at 1, 2, ... until one is found that satisfies the quantified expression when A or B is substituted for STR. This tells us that the call, ASSIGN(B,A), should not raise the exception. Then we can make substitutions in B, using A, so that the out specification of Assign is satisfied for this particular pair of parameters. The result should satisfy the assertions. However, EQUAL(A,B) is false in this outptut state. Is that correct PL/1?

Symbolic execution of package specifications on small tests is a powerful way to check the consequences of specifications. The results can be used to negotiate changes before implementation begins, or to see if a package provides features needed by some using program. Symbolic execution should be automated because it is easy to make mistakes in executions by hand ˜ indeed the same mistakes that were made in the specification.

As discussed in Chapter 5, the specifications of units in a package can depend on one another in ways that have unforeseen consequences. The dependencies in PL1_STRINGS are very simple. The only package functions that are used in other specifications are Slength (or Length) and Slast. There are no mutually dependent subprograms. Therefore, any implicit constraint on the package must a consequence of an individual subprogram specification.

Example: Specification of the Package Direct_IO

```
    with IO_EXCEPTIONS;
    generic
        type ELEMENT_TYPE is private;
    package DIRECT_IO is

        type FILE_TYPE is limited private;

        type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
        type COUNT is range 0 .. implementation defined;
        subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

        -- implementation concepts
--:     package EXTERNAL_SYSTEM is
--:
--:         type EXTERNAL_FILE is limited private;
--:         NO_FILE : constant EXTERNAL_FILE;
--:
--:         fuction PROPER(NAME : STRING) return BOOLEAN;
--:         function FILE_MAP(F : FILE_TYPE] return EXTERNAL_FILE;
--:         function NAME_MAP (NAME: STRING) return EXTERNAL_FILE;
--|             where PROPER(NAME);
--:
--:         function INACCESSIBLE(E : EXTERNAL_FILE) return BOOLEAN;
--|             where return (for all X : FILE_TYPE =>
--|                                  E /= FILE_MAP(X) ) and
--|                               (for all X : STRING => E /= NAME_MAP(X));
--:         function DISTINCT(F, F1 : FILE_TYPE) return BOOLEAN;
--|             where return FILE_MAP(F) /= FILE_MAP(F1);
--:
--:     end EXTERNAL_SYSTEM;

--:     use EXTERNAL_SYSTEM;
```

```
     -- Exceptions
     -- [Exceptions have been moved up so that they are visible in the anotations.]

     STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
     MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
     NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
     USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
     DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
     END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
     DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

     -- [The following function declarations have been moved up and reordered
     -- in order to make them properly visible in the annotations.]

     function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

     function MODE (FILE : in FILE_TYPE) return FILE_MODE;
--|      where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR;

     function NAME (FILE : in FILE_TYPE) return STRING;
--|      where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR;

     function FORM (FILE : in FILE_TYPE) return STRING;
--|      where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR;

     function INDEX (FILE : in FILE_TYPE) return POSTIVE_COUNT;
--|      where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR;

     function SIZE (FILE : in FILE_TYPE) return COUNT;
--|      where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR;
```

```
      -- File management

      procedure CREATE (FILE : in out FILE_TYPE;
                        MODE : in FILE_MODE := INOUT_FILE;
                        NAME : in STRING := "";
                        FORM : in STRING := "");
--|       where
--|           IS_OPEN(FILE) => raise STATUS_ERROR,
--|           not PROPER(NAME) => raise NAME_ERROR,
--|           raise USE_ERROR,
--|           out IS_OPEN(FILE),
--|           out (INDEX(FILE) = 1),
--|           out (MODE(FILE) = MODE),
--|           out in INACCESSIBLE(FILE_MAP(FILE)),
--|           out FILE_MAP(FILE) /= NO_FILE,
--|           out ( FILE_MAP(FILE) = NAME_MAP(NAME) );

      procedure OPEN(FILE : in out FILE_TYPE;
                     MODE : in FILE_MODE;
                     NAME : in STRING;
                     FORM : in STRING);
--|       where
--|           IS_OPEN(FILE) => raise STATUS_ERROR,
--|           not PROPER(NAME) or NAME_MAP(NAME) = NO_FILE =>
--|                                                  raise NAME_ERROR,
--|           raise USE_ERROR,
--|           out IS_OPEN(FILE),
--|           out (INDEX(FILE) = 1),
--|           out (MODE(FILE) = MODE),
--|           out (FILE_MAP(FILE) = in NAME_MAP(NAME));

      procedure CLOSE(FILE : in out FILE_TYPE);
--|       where
--|           not IS_OPEN(FILE) => raise STATUS_ERROR,
--|           out (not IS_OPEN(FILE)),
--|           out (FILE_MAP(FILE) = NO_FILE);
```

```
        procedure DELETE (FILE : in out FILE_TYPE);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             raise USE_ERROR,
--|             out (not IS_OPEN(FILE)),
--|             out (FILE_MAP(FILE) = NO_FILE),
--|             out INACCESSIBLE(in FILE_MAP(in FILE));

        procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             raise USE_ERROR,
--|             out (INDEX(FILE) = 1),
--|             out (MODE(FILE) = MODE);

        procedure RESET(FILE : in out FILE_TYPE);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             out (INDEX(FILE) = 1),
        -- Input and output operations

        procedure READ(FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE;
                       FROM : POSITIVE_COUNT);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             MODE(FILE) = OUT_FILE => raise MODE_ERROR,
--|             FROM > SIZE(FILE) => raise END_ERROR,
--|             raise DATA_ERROR,
--|             out (INDEX(FILE) = FROM + 1);

        procedure READ(FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             MODE(FILE) = OUT_FILE => raise MODE_ERROR,
--|             END_OF_FILE(FILE) => raise END_ERROR,
--|             raise DATA_ERROR,
--|             out (INDEX(FILE) = in INDEX(FILE) + 1);

        procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE;
                        TO : POSITIVE_COUNT);
--|         where
--|             raise USE_ERROR,
-- Raised if capacity of external file is exceeded.
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             MODE(FILE) = IN_FILE => raise MODE_ERROR,
--|             out (INDEX(FILE) = TO + 1);

        procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);
--|         where
--|             raise USE_ERROR,
    -- Raised if capacity of external file is exceeded.
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             MODE(FILE) = IN_FILE => raise MODE_ERROR,
--|             out (INDEX(FILE) = in INDEX(FILE) + 1),

        procedure SET_INDEX(FILE : in FILE_TYPE; TO : in POSTIVE_COUNT);
--|         where
--|             not IS_OPEN(FILE) => raise STATUS_ERROR,
--|             out (INDEX(FILE) = TO);
```

```
      function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
--|       where
--|          not IS_OPEN(FILE) => raise STATUS_ERROR,
--|          MODE(FILE) = OUT_FILE => raise MODE_ERROR,
--|          return (INDEX(FILE) > SIZE(FILE));

--|   axiom for all S : DIRECT_IO'TYPE;  F, F1 : FILE_TYPE;
--|        INDEX, I, J : POSITIVE_COUNT;  ITEM, X, Y : ELEMENT_TYPE;
--|        NAME, FORM : STRING; MODE : FILE_MODE
--|        =>
--|      S[READ (F, ITEM, INDEX)] = S[SET_INDEX(F, INDEX);
--|                                  READ(F, ITEM)],
--|      S[WRITE(F, ITEM, INDEX)] = S[SET_INDEX(F, INDEX);
--|                                  WRITE(F, ITEM)],
--|      S[WRITE(F1, X, INDEX)].READ'OUT(FILE =>F, ITEM =>Y, FROM =>
--|                         INDEX).ITEM =
--|                     if not DISTINCT(F, F1) then X
--|                     else S.READ'OUT(FILE => F, ITEM => Y, FROM =>
--|                         INDEX).INDEX
--|                     end if,
--|      S[WRITE(F, X, I); READ(F1, Y, J)] = if DISTINCT(F, F1) or else
--|                                          I /= J then
--|                                              S[READ(F1, Y, J);
--|                                                WRITE(F, X, I)]
--|                                          else
--|                                              S[WRITE(F, X, I);
--|                                                READ(F1, Y, J)]
--|                                          end if,
--|      S[WRITE(F, X, INDEX); WRITE(F, Y, INDEX)] =
--|         S[WRITE(F, Y, INDEX)],
--|      S[READ(F, X, I)].READ'OUT(FILE => F, ITEM => Y,
--|                         INDEX => I).ITEM =
--|      S.READ'OUT(FILE => F, ITEM => Y, INDEX => I).ITEM;

private
    -- Implementation dependent
end DIRECT_IO;
```

SYMBOLIC EXECUTION: "What Happens IF ...

```
CREATE (F, N)
          OUT : FILEMAP (F) = FILEMAP (N) = E,
                ISOPEN (F);

CREATE (F', N)
          OUT : FILEMAP (F) = E,
                FILEMAP (F') = NAMEMAP (N) = E',
                ISOPEN (F),
                ISOPEN (F'),
                E = E';

CLOSE (F)
          OUT : FILEMAP (F) = NO_FILE,
                FILEMAP (F') = E',
                NAMEMAP (N) = E',
                ISOPEN (F'),
                not ISOPEN (F),
                E = E';

OPEN (F, N)
          OUT : FILEMAP(F)=NAMEMAP(N)=FILEMAP(F')=E',
                ISOPEN (F'),
                ISOPEN (F),
                E = E';
```

Anna Near Term Tools

The primary goal is to encourage use of Formal Specifications in the development and maintenance of correct Ada programs.

*   Transformational Semantics

- ** More powerful annotations can be transformed into sets of simpler annotations (in most cases)
- ** The simplest annotations, assertions, can be transformed into runtime checking code.

EXAMPLE: Type annotation --› Object annotations

```
type EVEN is new INTEGER;
     --|  where X : EVEN => X mod 2 = 0;

subtype POS_EVEN is EVEN;
     --|  where X : POS_EVEN => X >= 0;

A : EVEN;
B : POS_EVEN;
...


type EVEN is new INTEGER;
subtype POS_EVEN is EVEN;

A : EVEN;             --|  A mod 2 = 0;
B : POS_EVEN;         --|  B mod 2 = 0    and
                      --|  ( B >= 0 );
...
```

*   Annotations can be transformed into executable runtime checks

```
 ANNA
Program
   |
   |     +-------------+              +----------+
   |     | Annotation  |  Ada program  |   Ada    |
   +---> |             |-------------->|          |----+
         | Transformer |  with checks  | Compiler |    |
         +-------------+              +----------+    |
                                                       V
                                                      Self
                                                    Checking
                                                     Program
```

*   Checking of consistency between the underlying Ada program and its formal specifications is performed automatically at runtime.

*   Exceptions and diagnostics are propagated in case of an inconsistency.

*   UPSHOT: Capability to run an Ada program against (in comparison with) its formal Anna specifications.

* APPLICATIONS

** Test and Debug

** Permanent checking of crucial specifications

Security in data bases

Error situations in control systems

** Comparative simulation (validation) of high level specifications of architectures

* ISSUES

** Implementation of Anna transformations

** Testing on significant examples

** OPTIMIZATION of runtime checks

** DESIGN-ANNA, a language for systems design

** Annotations of timing constraints

PROJECT STATUS

* Anna Language Manual -- available

* Anna Overview Paper -- available

* Introduction to Anna -- in progress

* Rationale for Anna design -- in progress

* Transformation for Runtime Testing

    Initial implementation for large Anna subset

    * Implementation specification -- completed

    * Support Tools:

```
    * DIANA extension                 -- completed
    * Anna parser                     -- completed
    * Validaton suite                 -- in progress
        first tests                   -- July 84
        first experimental version    -- April 85
```

* Anna Book -- in progress

SOME OTHER ANNA TOOLS

*	Optimizer	For Runtime checks, Uses rules associated with annotation concepts
*	Parallel Checker	Preprocessor compiles runtime checks for execution on parallel processors
*	Specification Analyzer	Consistency of package specifications, Symbolic execution of specification for question answering.
*	Structure Editor	Prompts for annotations Semantic analysis Context annotation checking Allows deferred decisions and tracks them.
*	Standard Packages	Timing Package

Appendix E

Re-Implementing AACAT Guard in Ada

Tony Brintnenhoff
SYSCON

Tony Brintzenhoff: Reimplementing ACCAT guard in ADA

Brintzenhoff was involved in an effort sponsored by DCA to evaluate Ada as a language for communications software. This effort involved reimplementing TCP/IP and ACCAT Guard in Ada. He talked about the reimplementation of ACCAT Guard, which also had as a goal the evaluation of Ada as a language for developing secure systems.

ACCAT Guard is a trusted application for the Kernelized Secure Operating System (KSOS) whose purpose is to connect two hosts with different security levels on a network (ostensibly the Arpanet) and allow controlled communication by upgrading and downgrading information moving between the hosts. Communication is through the ACCAT Guard, and reclassification is controlled by a security watch officer.

The project had three major objectives:

1. To evaluate Ada constructs for secure applications
2. To develop a list of restrictions on the use of Ada constructs which would enhance the verifiability of secure Ada systems
3. To develop a methodology for developing software in Ada

In order to provide an environment in which the ACCAT Guard could be executed, the project also designed emulations of the ACCAT Guard's interface to KSOS and to the Arpanet.

Brintzenhoff's group used a draft formal specification of ACCAT Guard written in the formal specification language SPECIAL. The aim was to translate the SPECIAL specification into an Ada program.

The Ada design methodology developed for the project used the concept of "virtual packages". It was discovered that developing a design at the package level offered too little granularity in constructing a design, while developing the design at the subprogram level provided too much granularity. Virtual packages are a graphical representation of a package which contains some detail of the units internal to the package. This offered a medium level of granularity, as well as facilitating stepwise refinement.

The designers attempted to translate the OFUNs of the SPECIAL spec into Ada subprograms on a one-to-one basis. This aim was not completely realized, in part due to inadequacies of the SPECIAL specs. In certain places, the SPECIAL specs seemed to have been written procedurally rather than non-procedurally, as SPECIAL is meant to be written.

The designers felt that Ada provided an adequate degree of separation between the trusted and untrusted software. Generics and variant records were felt to be particularly useful in minimizing the amount of trusted code. It was estimated that the use of generics and variant records would decrease the amount of trusted code by roughly 4000 lines (generics were not used in the design because the compiler used did not support them).

Problems were encountered arising from erroneous programs.

SYSCON CORPORATION

EVALUATION OF ADA* AS A COMMUNICATIONS PROGRAMMING LANGUAGE PHASE II

FOR: DEFENSE COMMUNICATIONS AGENCY, DEFENSE COMMUNICATIONS ENGINEERING CENTER

BY: SYSCON CORPORATION SAN DIEGO DIVISION

16 JANUARY 1985

*ADA IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT, ADA JOINT PROGRAM OFFICE

SYSCON CORPORATION

THE EVALUATION OF ADA TO SUPPORT THE OBJECTIVES OF THE DOD COMPUTER SECURITY INITIATIVE PROGRAM

DCA OBJECTIVES

INVESTIGATE ABILITY OF ADA TO SUPPORT TRUSTED SOFTWARE APPLICATIONS

ADA ASSESSMENT

- EVALUATION OF ADA LANGUAGE CONSTRUCTS
- IDENTIFICATION OF ADA LANGUAGE RESTRICTIONS
- IDENTIFICATION OF ENFORCEMENT MECHANISMS FOR ADA LANGUAGE RESTRICTIONS
- RECOMMENDATION OF ADA DEVELOPMENT METHODOLOGY

SYSCON CORPORATION

THE EVALUATION OF ADA TO SUPPORT THE OBJECTIVES OF THE DOD COMPUTER SECURITY INITIATIVE PROGRAM

APPROACH

DEFINE SOFTWARE APPLICATIONS

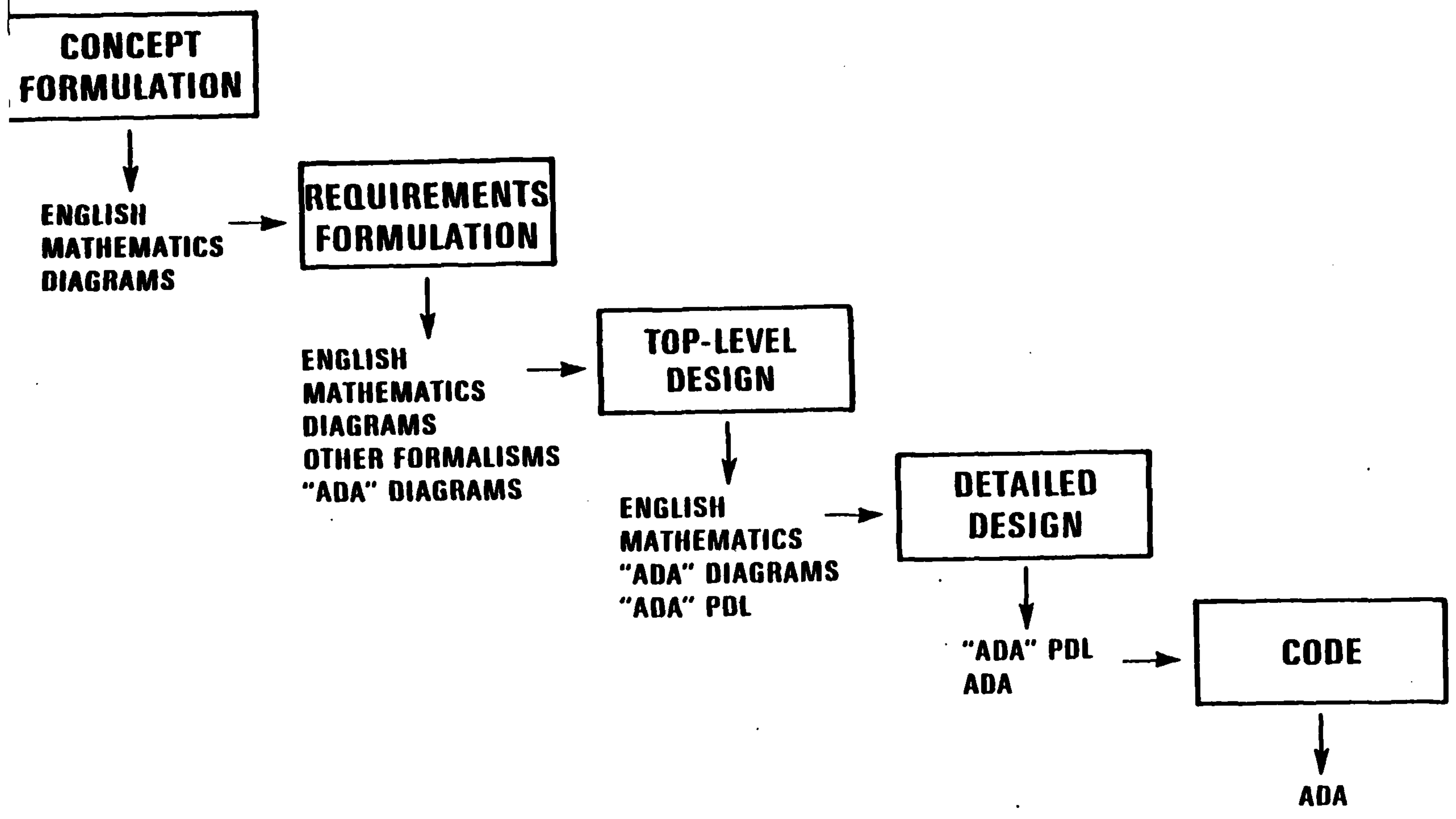
- ACCAT GUARD SUBSET
- EMULATION OF KSOS INTERFACE
- EMULATION OF GUARD INTERSYSTEM INTERFACES

DEVELOP SOFTWARE

- USE CONTEMPORARY SOFTWARE ENGINEERING PRINCIPLES
- DEFINE PROTOTYPE METHODOLOGY
- TRANSCRIBE TRUSTED SOFTWARE SPECIAL SPECIFICATIONS
- DESIGN SOFTWARE COMPONENTS
- IMPLEMENT SOFTWARE COMPONENTS
- EVALUATE RESULTS

SYSCON CORPORATION

SOFTWARE-DEVELOPMENT-PHASE NOTATIONS



SYSCON CORPORATION

DESIGN METHODOLOGY

OBJECTIVES

ESTABLISH EARLY ADA ORIENTATION
PROVIDE EARLY, CONTINUAL DESIGN VISIBILITY
PROVIDE DESIGN CONTINUITY ACROSS DEVELOPMENT PHASES
PROVIDE FOR LATE COMMITMENT TO DETAIL
PROVIDE BASIS FOR CONFIGURATION MANAGEMENT
USE SOFTWARE ENGINEERING PRINCIPLES

ABSTRACTION	LOCALIZATION
INFORMATION HIDING	UNIFORMITY
MODULARITY	

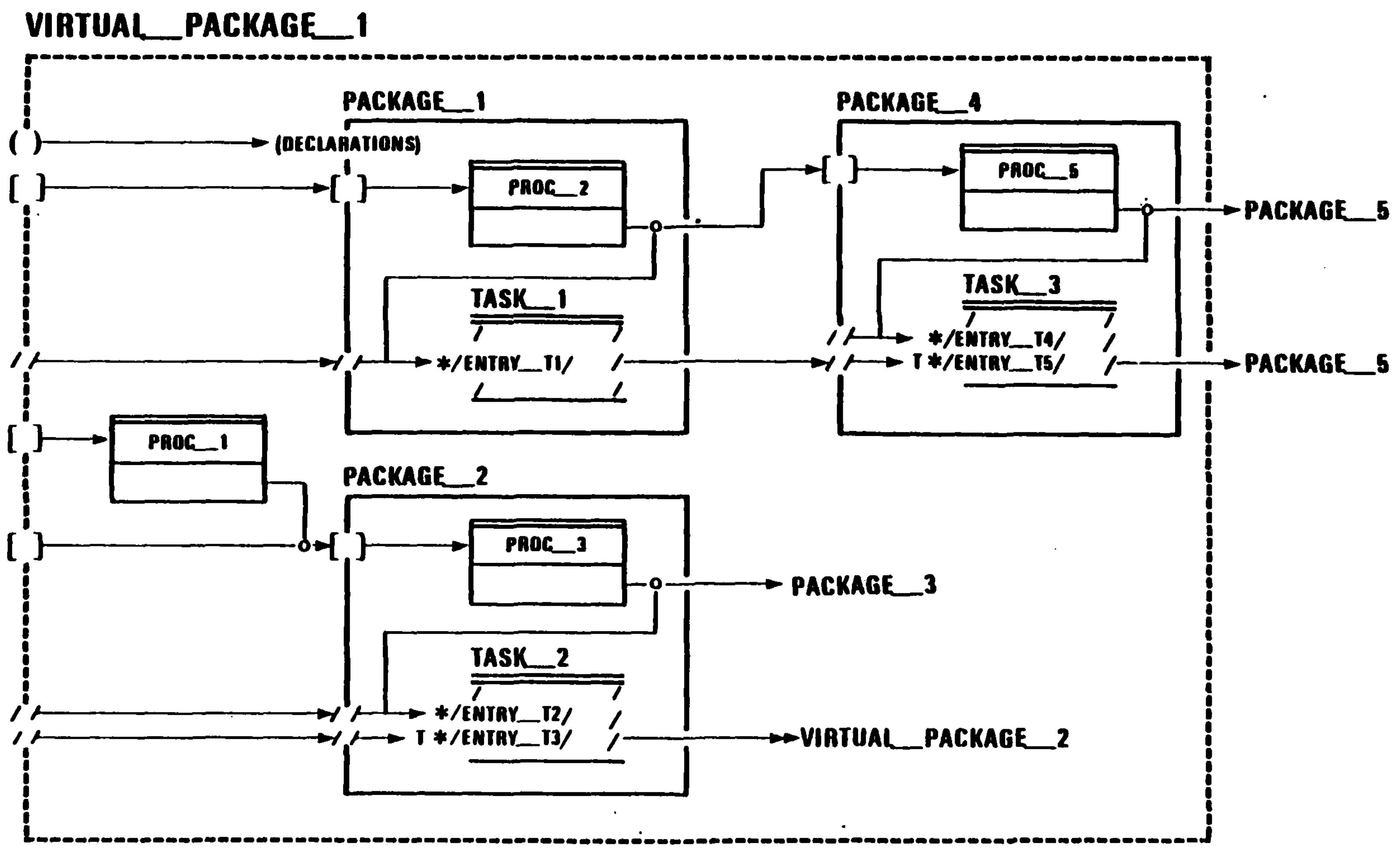
INCORPORATE EXISTING MODELS, ARCHITECTURAL PRINCIPLES

IMPLICATIONS

EVALUATE PACKAGE - PROGRAM LIBRARY GRANULARITY
    ESTABLISH VIRTUAL PACKAGE CONCEPT
USE OBJECT ORIENTED DESIGN DIAGRAMS
USE ADA AS A PDL
DEFINE MACROSCOPIC/MICROSCOPIC LEVELS

SYSCON CORPORATION

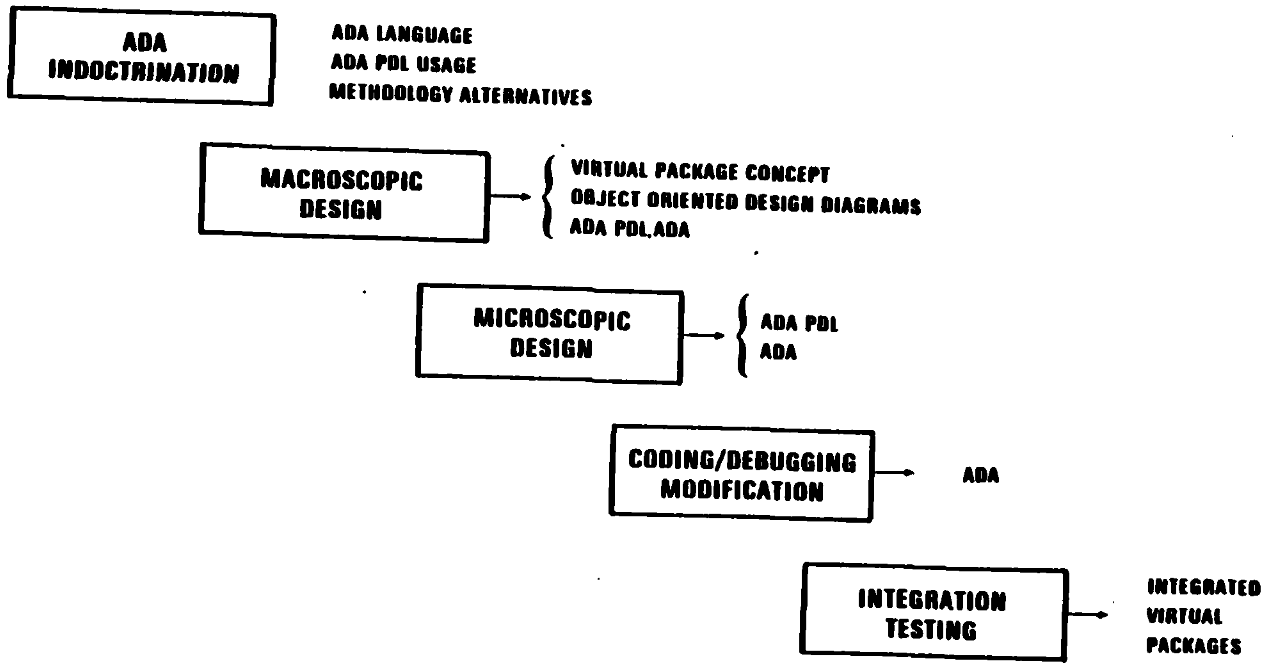
VIRTUAL PACKAGE REPRESENTATION CONVENTIONS



SYSCON CORPORATION

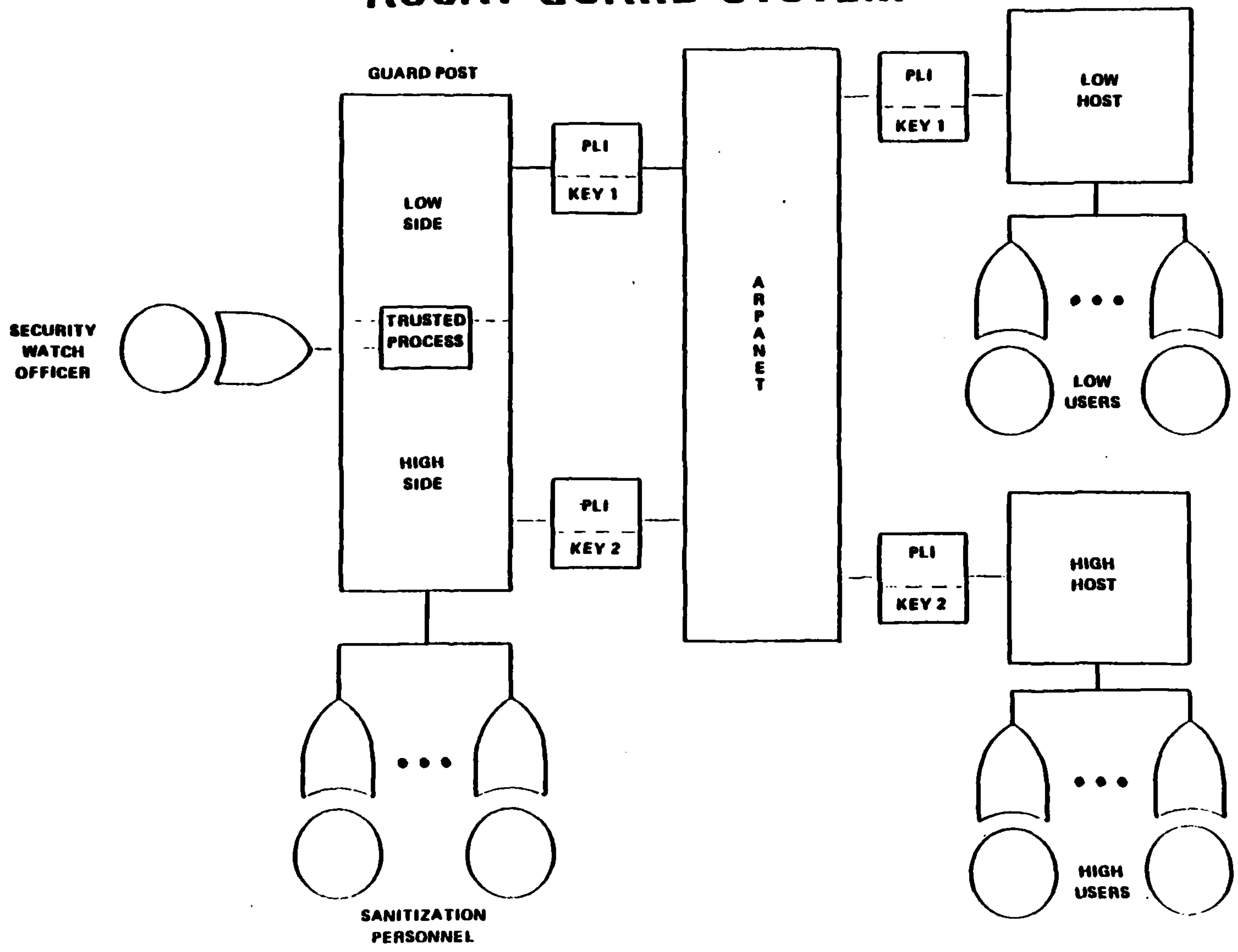
SOFTWARE DEVELOPMENT APPROACH

EMULATE MAJOR SOFTWARE DEVELOPMENT PROJECT
FORM MINI SOFTWARE DEVELOPMENT PROJECT
USE ADA AS PROGRAM DESIGN LANGUAGE



SYSCON CORPORATION

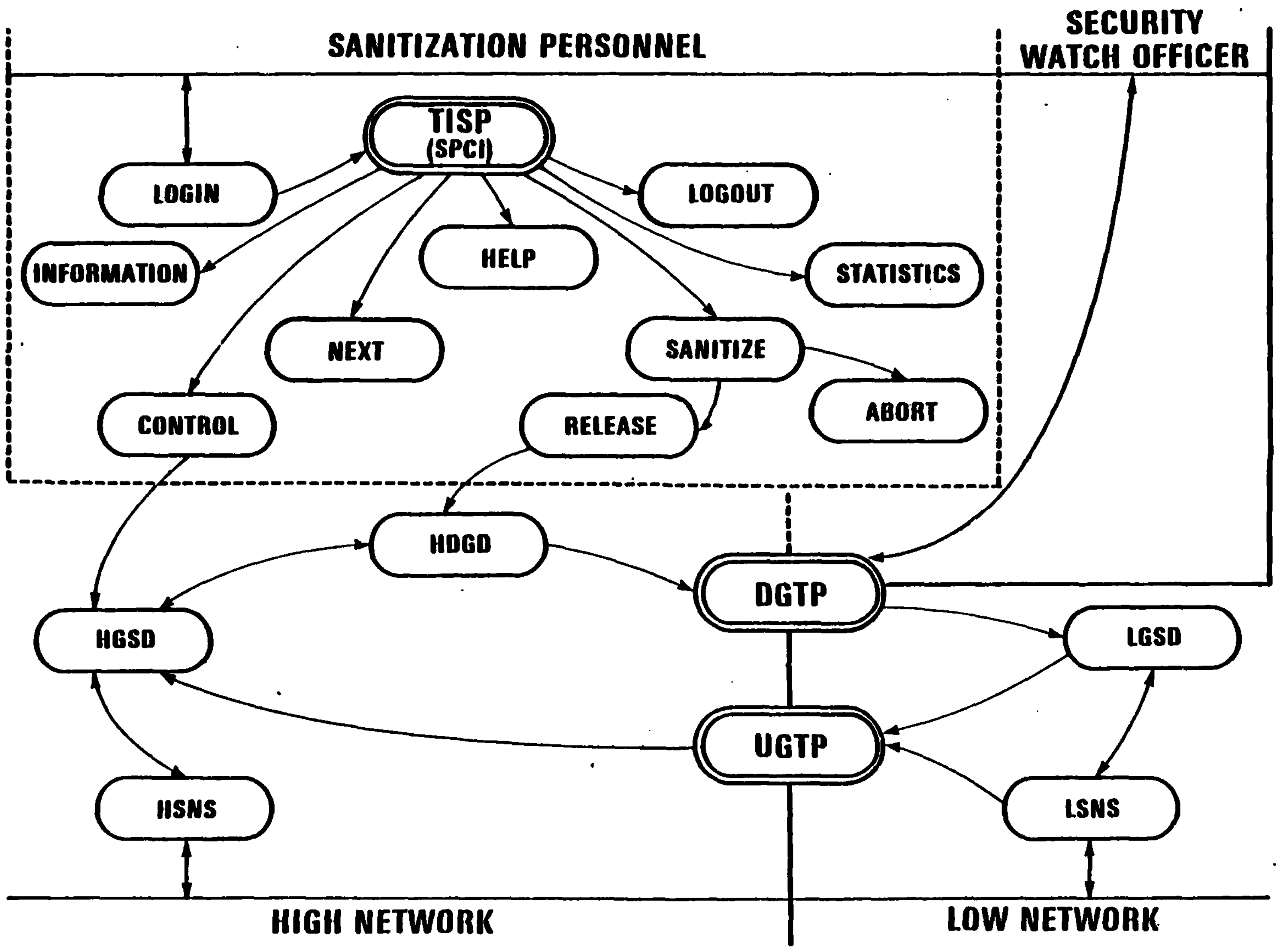
TRUSTED SOFTWARE APPLICATION ACCAT GUARD SYSTEM



59490

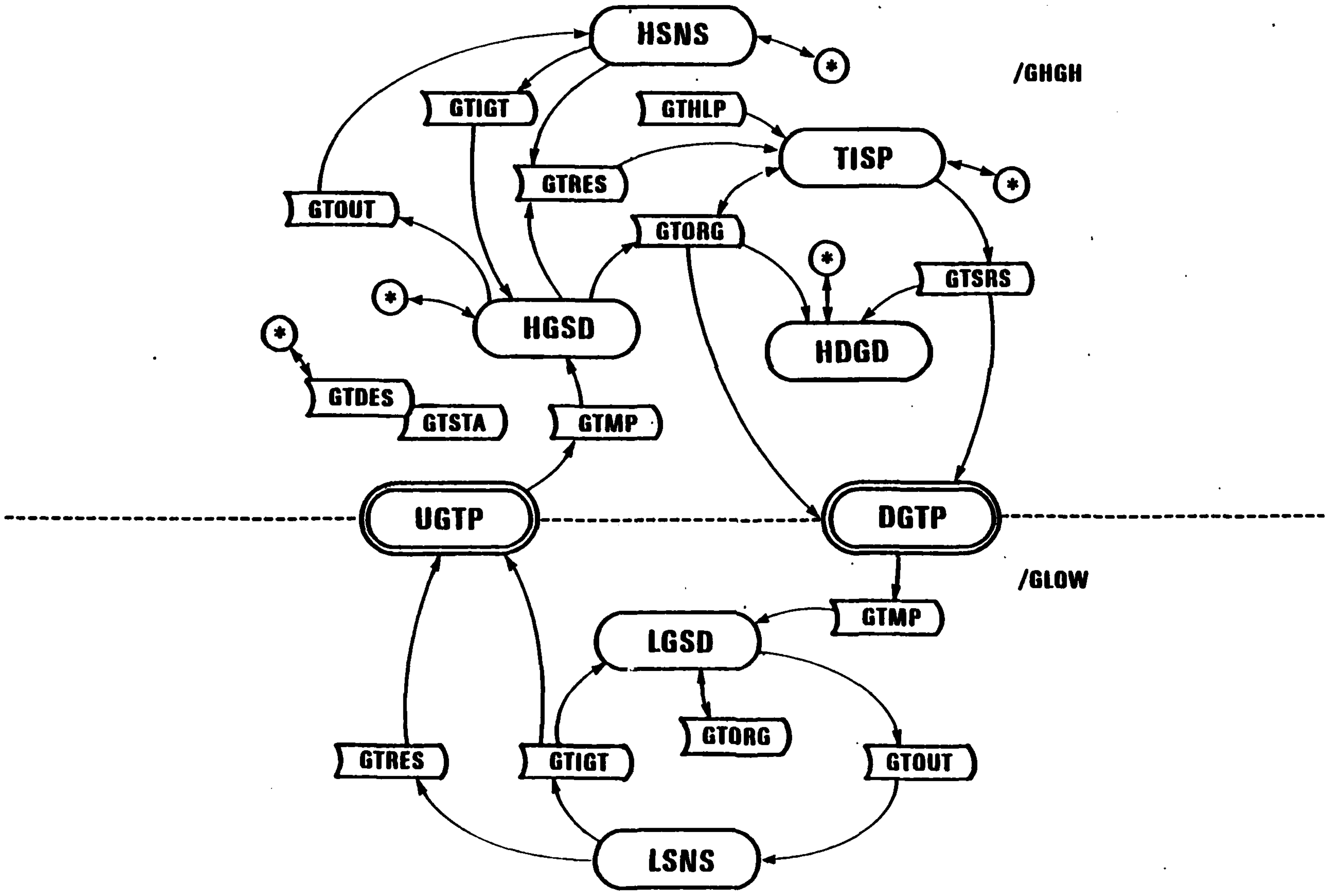
SYSCON CORPORATION

MODIFIED GUARD CONFIGURATION



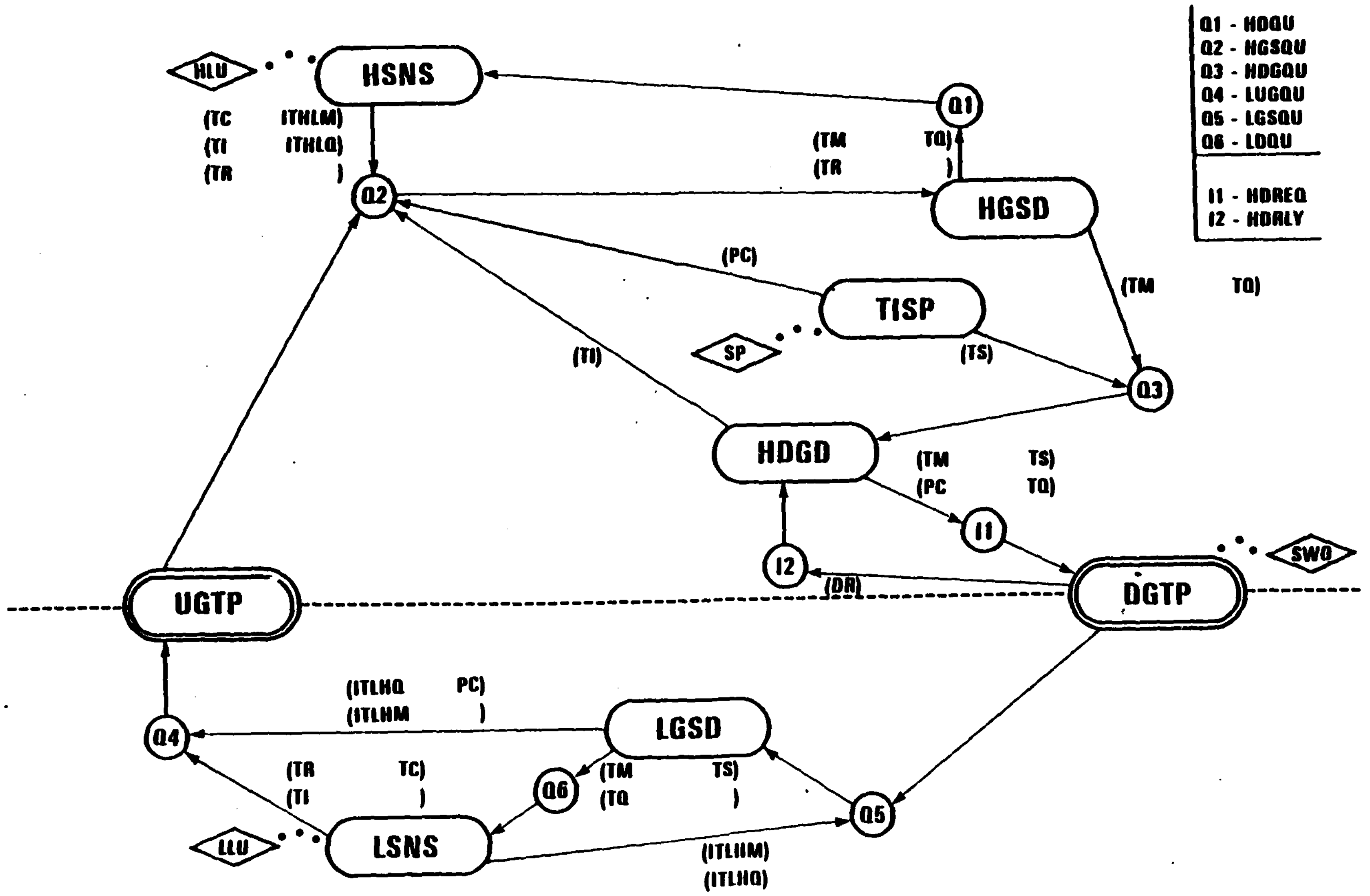
SYSCON CORPORATION

GUARD MESSAGE FLOW



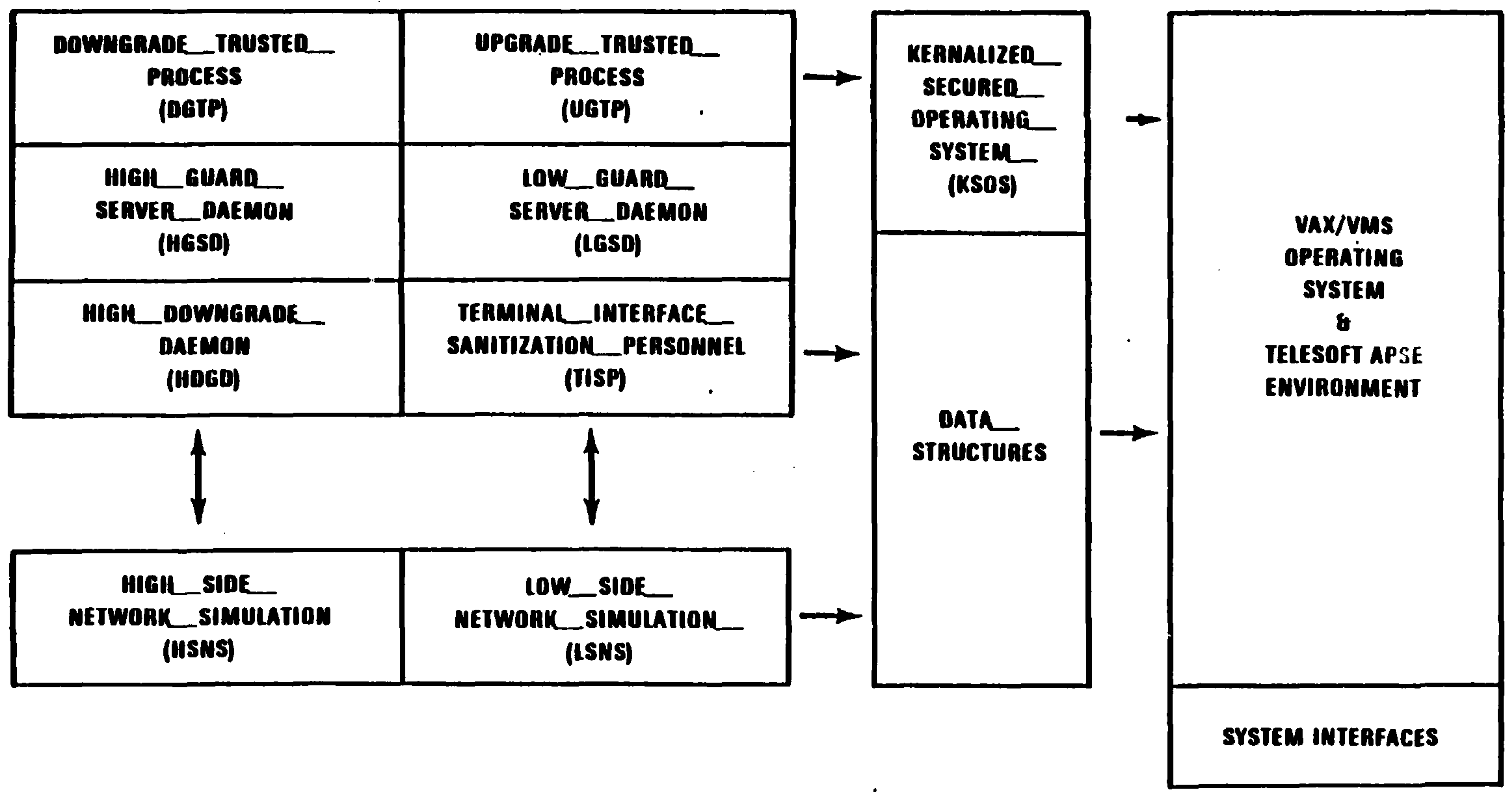
SYSCON CORPORATION

GUARD TRANSACTION FLOW



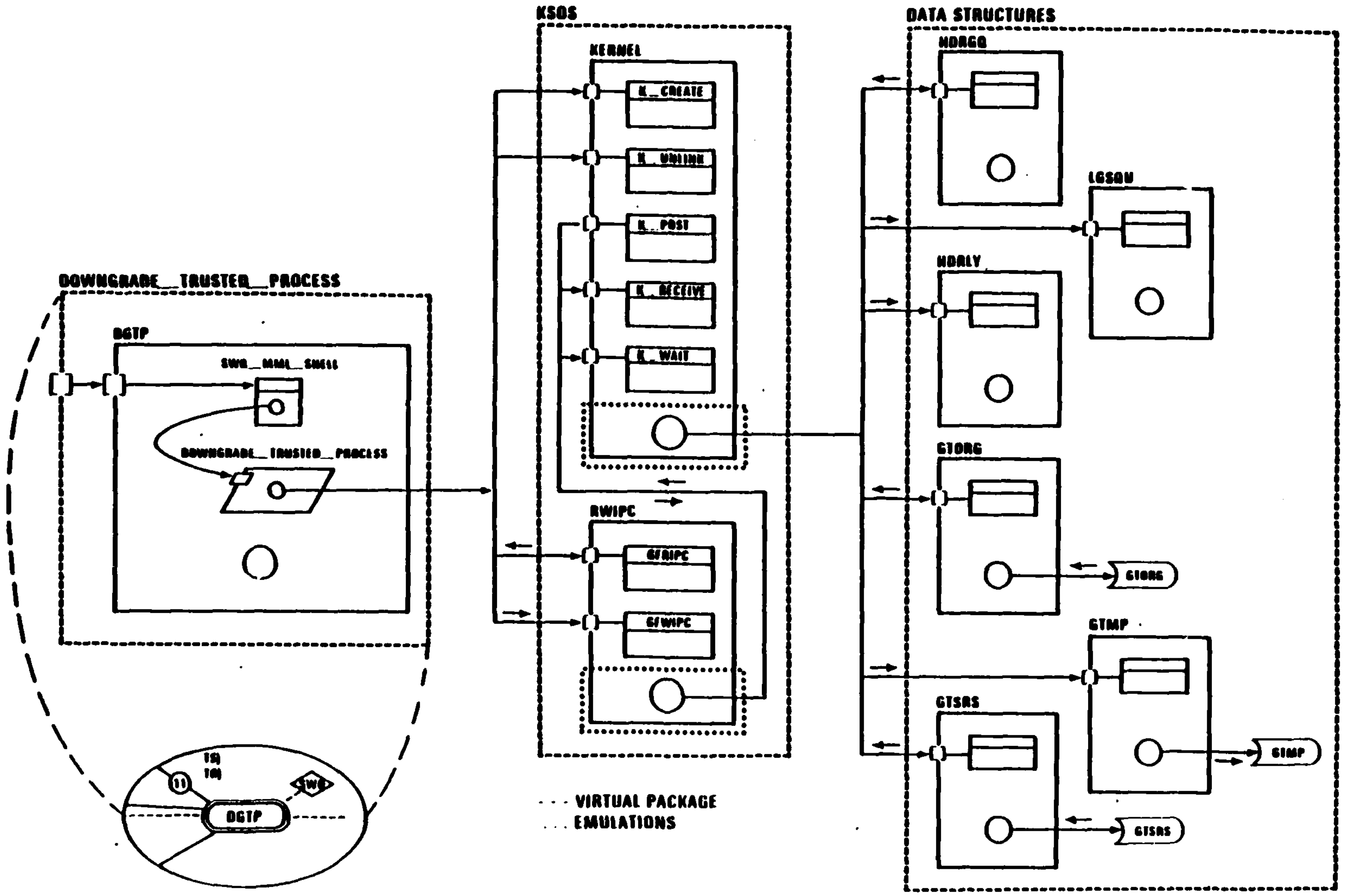
SYSCON CORPORATION

ACCAT GUARD DETAILED ARCHITECTURE


DOWNGRADE_TRUSTED_ PROCESS (DGTP)
UPGRADE_TRUSTED_ PROCESS (UGTP)
HIGH_GUARD_ SERVER_DAEMON (HGSD)
LOW_GUARD_ SERVER_DAEMON (LGSD)
HIGH_DOWNGRADE_ DAEMON (HDGD)
TERMINAL_INTERFACE_ SANITIZATION_PERSONNEL (TISP)
HIGH_SIDE_ NETWORK_SIMULATION (HSNS)
LOW_SIDE_ NETWORK_SIMULATION_ (LSNS)
KERNALIZED_ SECURED_ OPERATING_ SYSTEM_ (KSOS)
DATA_ STRUCTURES
VAX/VMS OPERATING SYSTEM & TELESOFT APSE ENVIRONMENT
SYSTEM INTERFACES

SYSCON CORPORATION

DOWNGRADE TRUSTED PROCESS INTERACTIONS



SYSCON CORPORATION

MACROSCOPIC DESIGN

```
--
--  Virtual Package DOWNGRADE_TRUSTED_PROCESS      --  **  DGTP  **
--

--  MODULE:         dgtp.specs (Version 1.1)
--  TYPE:           SPECIAL specifications
--  CURRENT PROBLEMS:
--      DGBUFFER interface in display_next_block
package DGTP is

  procedure SWO_MMI_SHELL ;

end DGTP ;

pragma PAGE ;
with RT_UTILITY ;
with EXTREFS ;
with DGTP_PARAMETERS ;
package body DGTP is
  use EXTREFS ;
  use KERNEL, RWIPC ;
  use DGTP_PARAMETERS ;

  --* response_type: (Accept,  Reject,  Continue,  Yes,
  --*                  No,      Logout,  Invalid) ;

  --* tty_resp: ONE_OF(CHAR ; NULL) ;

        •  •  •
```

MACROSCOPIC DESIGN

```
--* OFUN DownGrade_Trusted_Process() ;
--*                          $( Downgrade files from High to Low)
--*       EFFECTS
--*                EFFECTS_OF get_next_request() ;
--*

task body DOWN_GRADE_TRUSTED_PROCESS is
begin
   [ Open TTY file reference ]
   [ Set up control loop to monitor and transfer GUARD transactions ]
   [ Inform High Downgrade Daemon of down grade activity ]
   GET_NEXT_REQUEST [ Formatted request ] ;
   [ Terminate control loop when GUARD system terminates ]
   [ Close TTY file reference ]
end DOWN_GRADE_TRUSTED_PROCESS;

procedure SWO_MMI_SHELL is
   --
   --   This procedure is the package entry point to initiate the
   --   SWO terminal interface.
   --
begin
   [ Activate Downgrade Trusted Process Operations ]
end SWO_MMI_SHELL ;

end DGTP ;
```

SYSCON CORPORATION

MICROSCOPIC DESIGN

```
--* OFUN DownGrade_Trusted_Process() ; .
--*                          $( Downgrade files from High to Low)
--*       EFFECTS
--*               EFFECTS_OF get_next_request() ;
--*

task body DOWNGRADE_TRUSTED_PROCESS is
  --
  --  This is the parallel processor driving the DGTP.
  --
begin
  [ Open TTY connection with the Security Watch Officer ]
  loop
    [ Send 'READY' message to High Downgrade Daemon ]
    loop
      GET_NEXT_REQUEST [ Formatted request ] ;
      [ Terminate inner loop at log-off ]
    end loop ;
    [ Terminate outer loop at GUARD system shut-down ]
  end loop ;
  [ Close TTY connection with the Security Watch Officer ]
end DOWNGRADE_TRUSTED_PROCESS ;

procedure SWO_MMI_SHELL is
  --
  --  This procedure is the package entry point to initiate the
  --  SWO terminal interface.
  --
begin
  [ Activate Downgrade Trusted Process Operations ]
end SWO_MMI_SHELL ;

end DGTP ;
```

ADA SOURCE CODE

```
--* OFUN DownGrade_Trusted_Process() ;
--*                        $( Downgrade files from High to Low)
--*       EFFECTS
--*               EFFECTS_OF get_next_request() ;
--*
task body DOWNGRADE_TRUSTED_PROCESS is
  --
  --  This is the parallel processor driving the DGTP.
  --
  PROCESSING : Boolean := False ;
begin
  accept ACTIVATE ;   --  Controlled activation of the task
  TTYOD := K_OPEN (TTY_SEID, OM_EXCLUSIVE, STCAP, True) ; -- Open TTY
  while  not GUARD_SYSTEM_DOWN  loop
    GFWIPC (HDGDSEID, READY, HIGHNSPSEID) ; -- Send 'READY' notice
    SWO_LOGOUT_FLAG  := False ;
    while  not SWO_LOGOUT_FLAG  loop
      if  not PROCESSING  then
        PROCESSING := True ;
        GET_NEXT_REQUEST (GMSG1) ;
        PROCESSING := False ;
      end if ;
    end loop;
  end loop;
  K_CLOSE (TTYOD) ;  --  Close TTY
end DOWNGRADE_TRUSTED_PROCESS ;

procedure SWO_MMI_SHELL is
  --
begin
  DOWNGRADE_TRUSTED_PROCESS.ACTIVATE ;  -- Activate Downgrade task
end SWO_MMI_SHELL ;

end DGTP ;
```

SYSCON CORPORATION

ACCAT GUARD SOFTWARE STATEMENT ANALYSIS

DOWNGRADE_TRUSTED_PROCESS VIRTUAL PACKAGE

Compilation Unit Name	Unit Type	Source Statements	Comment Statements	Total Lines
DGTP_PARAMETERS	Library Unit	24	37	82
DGTP	Library Unit	6	8	23
	Secondary Unit	554	1558	2532
TOTAL		584	1603	2637

SYSTEM SOFTWARE STATEMENT ANALYSIS SUMMARY

Virtual Package Name	Number of CU's	Source Statements	Comment Statements	Total Lines
GUARD_GLOBAL_TYPE	1	54	99	170
GUARD_MASTER	1	19	17	34
HIGH_DOWNGRADE_ DAEMON	1	140	91	251
HIGH_GUARD_SERVER_ DAEMON	1	220	206	488
HIGH_SIDE_NETWORK_ SIMULATOR	3	593	812	1790
LOW_GUARD_SERVER_ DAEMON	1	120	111	278
LOW_SIDE_NETWORK_ SIMULATOR	3	599	828	1807
TERMINAL_INTERFACE_ SANITIZATION_PERSONNEL	3	924	809	2306
KERNELIZED_SECURED_ OPERATING_SYSTEM	2	750	1106	2328
DOWNGRADE_TRUSTED_PROCESS	2	584	1603	2637
UPGRADE_TRUSTED_PROCESS	2	99	173	354
DATA_STRUCTURES	28	2673	3674	8862
TOTAL	48	6775	9529	21305

SYSCON CORPORATION

SOFTWARE QUALITY CRITERIA

SOFTWARE QUALITY FACTORS / SOFTWARE QUALITY CRITERIA		ACCURACY	COMMUNICATION COMMONALITY	COMMUNICATIVENESS	COMPLETENESS	CONCISENESS	CONSISTENCY	DATA COMMONALITY	ERROR MANAGEMENT	GENERALITY	HARDWARE ARCHITECTURE	HARDWARE INDEPENDENCE	INSTRUMENTATION	LANGUAGE CONSTRUCTS	LANGUAGE IMPLEMENTATION	MODULARITY	OPERABILITY	OPERATING SYSTEM ARCHITECTURE	OPERATING SYSTEM INDEPENDENCE	SELF-DESCRIPTIVENESS	SIMPLICITY	TRACEABILITY
SOFTWARE DEVELOPMENT	EFFICIENCY I					●								●								
	FLEXIBILITY									●		●				●				●		
	INTEROPERABILITY		●													●						
	MAINTAINABILITY					●	●	●								●		●		●	●	●
	REUSABILITY									●		●				●			●	●		
	TESTABILITY					●							●			●				●	●	
	TRANSPORTABILITY											●				●			●	●		
SOFTWARE PERFORMANCE	CORRECTNESS	●			●		●															●
	EFFICIENCY II										●				●			●				
	INTEGRITY	●				●											●			●	●	
	RELIABILITY	●	●	●			●	●	●								●				●	
	ROBUSTNESS						●		●	●												
	USABILITY			●													●					

SC1 2234U

SC169660

SYSCON CORPORATION

SOFTWARE ARCHITECTURE EVALUATION
SYSTEM ARCHITECTURE - TRUSTED SOFTWARE

DEVELOPMENT

- EFFICIENCY-II
  - ADA HAS ROBUST LOGICAL, LEXICAL, PHYSICAL REPRESENTATION CAPABILITIES
  - TRUSTED SOFTWARE RESTRICTIONS REQUIRED TO
    - LIMIT NEEDLESS COMPLEXITY
    - MAKE FORMAL VERIFICATION FEASIBLE
- FLEXIBILITY
  - FACILITATED SIGNIFICANTLY VIA GENERICS, VARIANT RECORDS
  - ORIGINAL MODULARITY PRESERVED
  - SEPARATION OF TRUSTED, NONTRUSTED SOFTWARE ACHIEVED
- MAINTAINABILITY
  - GOOD TRACEABILITY VIA MACRO/MICRO DESIGN METHODOLOGY
  - FACILITATED VIA GENERICS, COULD ELIMINATE 4000 STATEMENTS
  - SEPARATION OF TRUSTED, NONTRUSTED SOFTWARE
- REUSABILITY
  - NO SPECIFIC REQUIREMENTS OR ASSESSMENT
- TRANSPORTABILITY
  - NO SPECIFIC REQUIREMENTS OR ASSESSMENT

3Y_ _C. .
CORPORATION

SOFTWARE ARCHITECTURE EVALUATION
SYSTEM ARCHITECTURE - TRUSTED SOFTWARE

- PERFORMANCE
  - CORRECTNESS
    - INITIAL DESIGN ERRORS RESULTING IN ERRONEOUS PROGRAMS
  - EFFICIENCY-II
    - LIMITED ASSESSMENT DUE TO SYNCHRONOUS TEXT__IO
    - EFFECTS OF TASK ENCAPSULATION/NONENCAPSULATION NOT ASSESSED
  - INTEGRITY
    - LIMITED ASSESSMENT OF A1-LEVEL TRUSTED SOFTWARE CRITERIA
  - RELIABILITY
    - NO SPURIOUS ERRORS UNDER NORMAL OPERATIONS
    - STRESS TESTING WITH ORIGINAL DESIGNS NOT PERFORMED
    - LIMITED STRESS TESTING ON MODIFIED DESIGNS
    - OCCASIONAL FAILURES DUE TO TASK STACK SIZE LIMITATIONS
  - ROBUSTNESS
    - MAN-MACHINE INTERFACES ARE CORRECT, COMPLETE, CONSISTENT
    - EMULATED KSOS ERROR CONDITIONS
      - NOT TESTED
      - MAY IMPACT CORRECTNESS, INTEGRITY, RELIABILITY

SYSCON CORPORATION

SOFTWARE ARCHITECTURE EVALUATION
SYSTEM ARCHITECTURE - TRUSTED SOFTWARE

APPROPRIATE USAGE

TYPE, OBJECT NAMING CONVENTIONS
VARIANT RECORDS
DATA INITIALIZATION
ATTRIBUTES
BLOCKS FOR EXCEPTION HANDLING LOCALIZATION
GENERIC PACKAGES
MONITOR AND TRANSPORTER TASKS

INAPPROPRIATE USAGE

ENCAPSULATION OF TASKS VIA PROCEDURES
IF STATEMENTS VS. CASE STATEMENTS
DIFFICULTY IN INSTRUMENTATION WITH TEXT__IO

SYSCON CORPORATION

ADA LANGUAGE EVALUATION

DISTINGUISHED FEATURES

- DECLARATIONS, TYPES
  - TYPE, SUBTYPE DECLARATIONS
  - ENUMERATION TYPES
  - RECORD, VARIANT RECORD TYPES
  - ACCESS TYPES
- NAMES, EXPRESSIONS
  - ATTRIBUTES
- PACKAGES
  - PRIVATE, LIMITED PRIVATE TYPES
- TASKS
  - RENDEZVOUS
  - TASK TYPES
- GENERICS

MISUSED/MISUNDERSTOOD FEATURES

- GLOBAL DATA DECLARATION, USAGES
- STATEMENTS
  - CASE VS. IF
  - BLOCK
- EXCEPTIONS
  - HANDLING, PROPAGATION
- DECLARATIONS, TYPES
  - PRIVATE, LIMITED PRIVATE TYPES

SYSCON CORPORATION

ADA LANGUAGE EVALUATION (CONT.)

PROBLEMATIC FEATURES

- DECLARATIONS, TYPES
  - VARIANT RECORD STRUCTURES
    - MAY REDUCE MAINTAINABILITY
    - MAY BE EXCESSIVELY COMPLEX
- STATEMENTS
  - EVALUATION: IF - DYNAMIC VS. CASE - STATIC
- PACKAGES
  - PACKAGE SIZE LIMITATIONS
- TASKS
  - DEPENDENCY ON RTS FOR TASKING MODEL
  - EFFECTS OF TASK STACK SIZE LIMITATIONS
- PROGRAM STRUCTURE, COMPILATION
  - LARGE DEPENDENCY CHAINS IN LARGE SYSTEMS
  - PROPAGATION EFFECTS OF SPECIFICATION RECOMPILATION
  - PARTIAL ELABORATION ORDER, ACCESS BEFORE ELABORATION ERRORS
  - IS SEPARATE: LEXICAL VS. LOGICAL VS. OBJECT PLACEMENT
- EXCEPTIONS
  - WHEN, WHERE, HOW TO USE
  - EFFECTS OF EXTRANEOUS EXCEPTION PROPAGATION
  - WEAK ASSOCIATION OF EXCEPTIONS WITH SOURCE
    - PROLIFERATION OF EXCEPTION HANDLERS

SYSCON CORPORATION

ADA LANGUAGE EVALUATION

IMPLEMENTATION DEPENDENT FEATURES

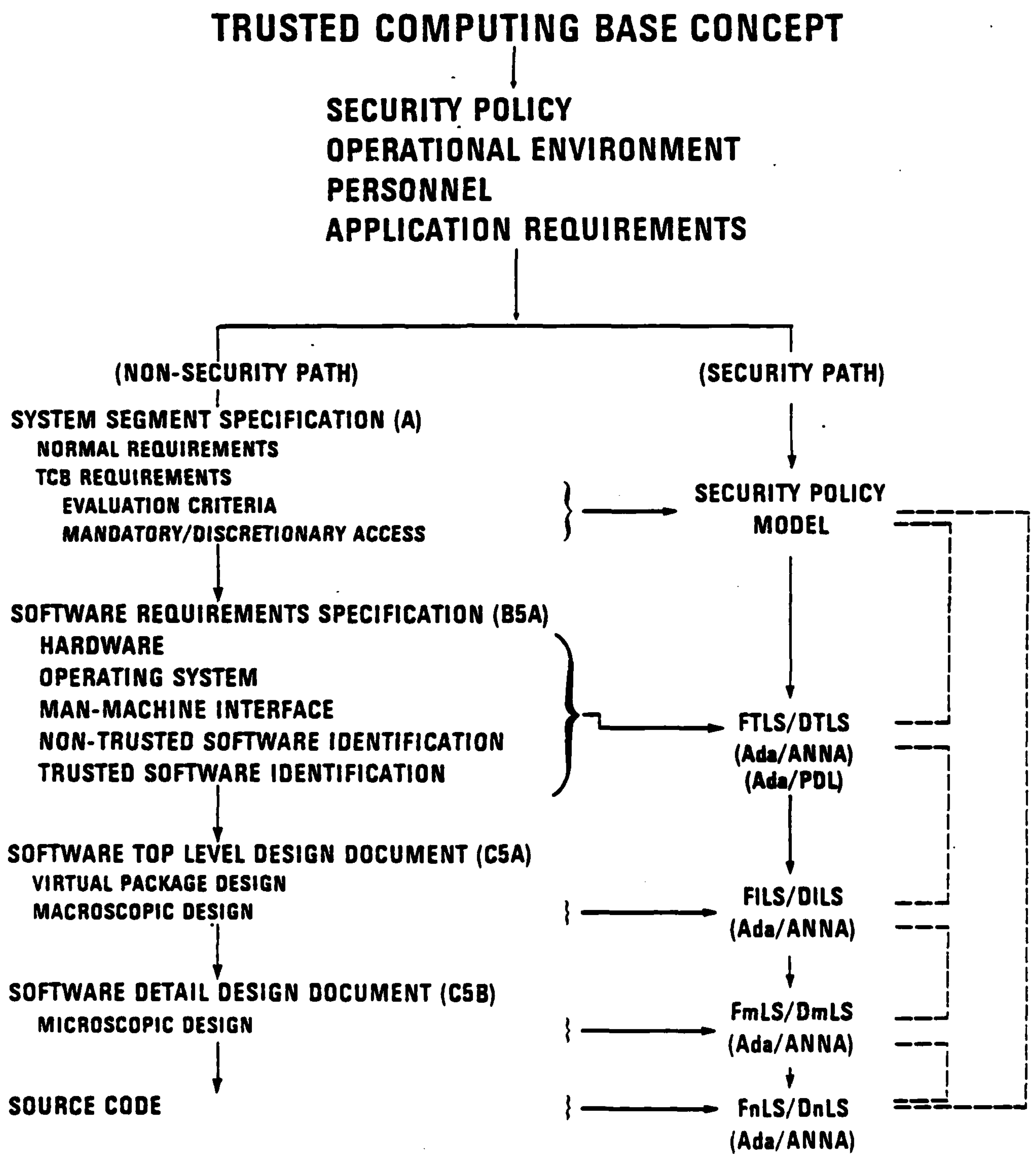
- STATED
    - APPENDIX F, PACKAGE SYSTEM
- UNSTATED
    - PARAMETER PASSING MECHANIZATION
    - COMPILER PRAGMATICS
    - TASK TERMINATION IN LIBRARY UNITS
    - SELECTIVE WAIT MODELS
    - TASK SCHEDULING ALGORITHM
        - PRIORITIZED TIME SLICE VS. RUN-UNTIL-BLOCK
        - SELECTIVE WAIT MECHANIZATION
    - DYNAMIC MEMORY MANAGEMENT DEPENDENCIES
    - TEXT__IO MECHANIZATION: SYNCHRONOUS VS. ASYNCHRONOUS

ADA LANGUAGE EDUCATION

- REQUIRES SOFTWARE ENGINEERING BASIS
- REQUIRES SOFTWARE METHODOLOGY CONTEXT
- REQUIRES ABRIDGED LANGUAGE REFERENCE MANUAL
- SHOULD INCLUDE USE OF ADA RATIONALE
- ADDITIONAL REQUIREMENTS
    - INCORPORATION OF GUIDELINES FOR TRANSPORTABILITY
    - INCORPORATION OF GUIDELINES FOR REUSABILITY

SYSCON CORPORATION

TRUSTED SOFTWARE DESIGN METHODOLOGY



SYSCON CORPORATION

ADA PROGRAMMING GUIDELINES
TRUSTED SOFTWARE

OBJECTIVES

- MAINTAINABILITY, TESTABILITY, CORRECTNESS, INTEGRITY, RELIABILITY, ROBUSTNESS
- DEFINITION OF COMPATIBLE, USABLE ADA SUBSET
- ACHIEVABLE FORMAL VERIFICATION

DECLARATIONS, TYPES

- "PROHIBIT TRUSTED - NONTRUSTED SOFTWARE INTERFACES FROM USING ACCESS VARIABLES"
- MINIMIZE OPPORTUNITY FOR PILFERING
- "PROHIBIT FORMING PRIVATE, LIMITED PRIVATE OBJECTS AS SIDE EFFECTS OF INCLUDING PRIVATE, LIMITED OBJECTS"
- AVOID MAINTAINABILITY PROBLEMS

SUBPROGRAMS

- "PROHIBIT FUNCTIONS FROM ACHIEVING SIDE EFFECTS"
- AVOID AMBIGUOUS RESULT CALCULATIONS

SYSCON CORPORATION

ADA PROGRAMMING GUIDELINES (CONT.)
TRUSTED SOFTWARE

OBJECTIVES (CONT.)

- TASKS
  - "PROHIBIT CREATION OF TASKS USING ALLOCATORS WHERE NUMBER OF TASKS IS VARIABLE AND INDETERMINATE"
  - REMOVE TIME/DATA DEPENDENCIES FROM VERIFICATION CONSIDERATION
- EXCEPTIONS
  - "INDICATE VIA ANNA OR OTHER ANNOTATIONS WHICH EXCEPTIONS ARE ASSOCIATED WITH WHICH SUBPROGRAMS AND TASK ENTRIES"
  - REMOVE PROGRAM AMBIGUITY, ELIMINATE UNNEEDED EXCEPTION HANDLERS

SYSCON CORPORATION

SOFTWARE PERFORMANCE CRITERIA

TRUSTED SOFTWARE EVALUATION

TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA

D – COMMON PRACTICE
C1 – DISCRETIONARY PROTECTION
C2 – CONTROLLED ACCESS PROTECTION
B1 – LABELED SECURITY PROTECTION
B2 – STRUCTURED PROTECTION
B3 – SECURITY DOMAINS
*A1 – VERIFIED DESIGN (KSOS, GUARD)
*A2 – VERIFIED IMPLEMENTATION

EVALUATE GUARD IMPLEMENTATION BASED ON

GUARD SPECIAL SPECIFICATIONS
COMPUTER SYSTEM SECURITY EVALUATION CRITERIA
CORRESPONDENCE VERIFICATION OF ADA SOURCE CODE
PROGRAM TESTING
PROGRAMMING STYLE
AD HOC FACTORS

SYSCON CORPORATION

SOFTWARE PERFORMANCE EVALUATION TRUSTED SOFTWARE

CRITERIA

- CORRECTNESS, INTEGRITY, RELIABILITY, ROBUSTNESS, FORMAL VERIFICATION

EVALUATION

- GENERALLY STRAIGHT FORWARD ADA IMPLEMENTATION
  - LIMITED ADA SUBSET USED
- VALIDITY OF SPECIAL-TO-ADA MAPPING UNCERTAIN REGARDING
  - INTERPRETATION OF SPECIAL REQUIREMENTS, DESIGNS
  - IDIOSYNCRASIES OF KSOS, UNIX IPC
  - LANGUAGE SYNTAX/SEMANTICS DIFFERENCES
  - COMPILER LIMITATIONS
  - DYNAMICS OF PROGRAM EXECUTION
- REQUIRES VALIDATED COMPILER FOR
  - FULL IMPLEMENTATION OF ORIGINAL DESIGNS
  - SYSTEMATIC COMPARISON BETWEEN SPECIAL AND ADA
  - CONDUCTING STRESS TESTING
  - ASSESSING ALTERNATIVE ARCHITECTURES
- MINIMIZE TRANSLATION DIFFICULTIES
  - USE ANNOTATED ADA (ANNA)
  - FORM ADA/ANNA BASED DESIGN METHODOLOGY
  - ACHIEVE COMPATIBLE SEMANTICS FROM BEGINNING

SYSCON CORPORATION

SOFTWARE PERFORMANCE CRITERIA

SOFTWARE TESTS

TCP/IP/ADCCP

FUNCTIONALITY TESTING:

- TCP
  - MISSING SEGMENT(S)
  - DUPLICATE SEGMENT(S)
  - SEGMENT CHECKSUM ERRORS
  - SECURITY/PRECEDENCE VIOLATIONS
- IP
  - DATAGRAM CHECKSUM
  - DESTINATION-UNREACHABLE
  - TIME-TO-LIVE
  - INVALID-SUBNET-PARAMETERS
- ADCCP
  - OUT-OF-CONTEXT COMMANDS
  - OUT-OF-CONTEXT RESPONSES
  - TIMEOUTS
  - INVALID FRAME ERRORS
  - CRC ERRORS
- LINE CONTROL MODULE (LCM)
  - TIME-OUTS (LINE DROP)
  - DATA ERRORS (BIT DROP)

ACCAT GUARD

FUNCTIONALITY TESTING:

- HIGH-LOW MAIL
- LOW-HIGH MAIL
- HIGH-LOW QUERY
- LOW-HIGH RESPONSE
- LOW-HIGH QUERY
- HIGH-LOW RESPONSE
- DOWNGRADE REJECTION
- HIGH/LOW BUFFER WATERMARKS
- GUARD TERMINATION
- DOWN GRADING
- SANITIZATION

ADA-SPECIFIC EFFICIENCY II CRITERIA

PRAGMAS: CONTROLLED, INLINE, OPTIMIZE, PRIORITY, SHARED, SUPPRESS
TYPES/OBJECTS: DYNAMICALLY VS. STATICALLY CREATED OBJECTS
SUBPROGRAMS: EFFECTS OF EXTENSIVE ELABORATION
TASKS: REGULARITY, ACCURACY OF EVENT TIMING, INTERRUPT PROCESSING
TASK ACCESS ALTERNATIVES
EXCEPTIONS: HANDLING, PROPAGATION, TASKING INTERACTIONS
GENERICS: EFFECTS OF DYNAMIC INSTANTIATIONS
IMPLEMENTATION-DEPENDENT FEATURES: UNCHECKED PROGRAMMING

SYSCON CORPORATION

PROJECT RETROSPECTIVES

TELESOFT ADA COMPILER

- WAS INCORRECT, INCOMPLETE, INCONSISTENT PROTOTYPE
- CAUSED SIGNIFICANT
  - LOSS OF TIME, 2-4 PERSON-MONTHS PER APPLICATION
  - SOFTWARE ARCHITECTURE CHANGES
  - MODULE ARCHITECTURE CHANGES
  - IMPLEMENTATION CHANGES

IMPLEMENTATION CHARACTERISTICS/EMPHASIS

- MORE INDEPENDENT REVIEW OF MACRO/MICRO DESIGNS REQUIRED
- DIFFICULT TO MAINTAIN PROPER BALANCE BETWEEN
  - DESIGN, ANALYSIS OF DESIGNS AND ALTERNATIVE DESIGNS VS
  - IMPLEMENTATION, EXECUTION, EVALUATION OF DESIGNS
- DEVELOPMENT EMPHASIS
  - FULL DESIGN AND BROAD, PARTIAL IMPLEMENTATION VS
  - PARTIAL DESIGN AND NARROW, FULL IMPLEMENTATION
- ATTEMPT AT LIBERAL USE OF ADA FEATURES AT
  - SYSTEM ARCHITECTURE LEVEL
  - MODULE ARCHITECTURE LEVEL
  - PROGRAMMING LEVEL

SYSCON CORPORATION

CONCLUSIONS

SOFTWARE DEVELOPMENT METHODOLOGY

- MACRO/MICRO DESIGN METHODOLOGY
    - WORKS VERY WELL
    - REQUIRES INTEGRATED TOOL SET FOR EFFICIENCY
    - INSUFFICIENT PROTOTYPING PERFORMED
    - PDL USAGE NOT ALWAYS COMPLETE, CONSISTENT

ADA LANGUAGE

- EXCELLENT DESIGN, IMPLEMENTATION FEATURES
- CAUTION REQUIRED REGARDING
    - LANGUAGE SUBTLETIES, UNSPECIFIED OPTIONS
    - IMPLEMENTATION DEPENDENCIES
    - RUN-TIME SUPPORT DEPENDENCIES
- MORE ANALYSIS REQUIRED FOR
    - COMPLETING, VALIDATING TRUSTED SOFTWARE GUIDELINES
    - INCLUDING TRANSPORTABILITY, REUSABILITY GUIDELINES
- REQUIRES INTEGRATED EDUCATION APPROACH ON
    - SOFTWARE ENGINEERING, LANGUAGE FEATURES
    - SOFTWARE DEVELOPMENT METHODOLOGY, TOOLS

SYSCON CORPORATION

CONCLUSIONS (CONT.)

SOFTWARE ERRORS

- DESIGN, PROGRAMMING ERRORS
  - ATTRIBUTED TO EARLY LACK OF EXPERIENCE
  - SUBTLETIES IN ADA SYNTAX, SEMANTICS
  - MINIMIZED WITH EDUCATION, TRAINING, EXPERIENCE
- UNCERTAINTIES REGARDING
  - COMPLEX SOFTWARE ARCHITECTURES
  - FULL, UNCONSTRAINED USE OF ALL ADA FEATURES

PROJECT SUMMARY

- SIGNIFICANT ACHIEVEMENTS MADE
- SIGNIFICANT INSIGHTS GAINED
- MAJOR, MINOR DISAPPOINTMENTS ENCOUNTERED

SYSCON CORPORATION

CONCLUSIONS (CONT.)

SOFTWARE ARCHITECTURES

- SYSTEM ARCHITECTURES
  - OSI REFERENCE, SUBLAYER MODELS INCLUDED
  - SEPARATION OF TRUSTED, NONTRUSTED PROCESSES ACHIEVED
- MODULE ARCHITECTURES
  - LIMITED ALTERNATIVES EXPLORED
  - DESIGNS COMPROMISED BY COMPILER DEFICIENCIES
  - NOT ALWAYS BEST USE OF ADA FEATURES

SOFTWARE PERFORMANCE

- LIMITED ASSESSMENT OF CORRECTNESS, EFFICIENCY-II, INTEGRITY, RELIABILITY, ROBUSTNESS
- SELECTED ADA FEATURES
  - POTENTIALLY IMPROVE PERFORMANCE,
  - POTENTIALLY REDUCE PERFORMANCE
- UNCERTAINTIES IN INTERACTION OF COMPLEX FEATURES
- MAY BE SIGNIFICANTLY DEGRADED BY
  - IMPROPER ARCHITECTURES
  - ARCHITECTURE - RUN-TIME ENVIRONMENT MISMATCHES

Appendix F

Army Secure Operating Systems

Eric Anderson
TRW

Eric Anderson: Army Secure Operating Systems

Eric Anderson is the Project Manager for TRW's Army Secure Operating System (ASOS) project. ASOS will be written in Ada and is required to meet the DoD Computer Security Center's A1 rating.

Defining Security Subjects

Security subjects are the active entities of the system from the point of view of security. The ASOS project had to decide what the security subjects of ASOS are. There were two candidates: a security subject could be an Ada program or a task within an Ada program.

The ASOS project found that tasks within an Ada program can communicate by means both overt (shared global variables, rendezvous) and covert (using the same global packages, having the same devices open to all tasks in a program). The only way to limit this communication would be to place severe restrictions on the use of Ada in the programs running on ASOS. It was felt by the ASOS designers that these restrictions were too severe, and thus all tasks within a given program must be considered to have the same security level. This essentially amounts to having the security subjects be complete Ada programs, so it was decided to define security subjects to be complete Ada programs. The security subjects would therefore communicate, when necessary, by a mechanism outside Ada which would be mediated by the ASOS Security Kernel. This approach also has the advantage of separating security issues from the Ada domanin of intertask communication.

RSL Issues

Ada features such as tasking and exception handling, which are classically functions of the operating system, require extensive runtime support. The Runtime Support Library (RSL) of the Security Kernel must be as secure as the Ada code for the Security Kernel itself.

To solve this problem, the ASOS project developed a layered approach in which the applications programs running on ASOS have an Applications RSL which makes calls to the Security Kernel. The Security Kernel has its own Kernel RSL, which implements a restricted subset of the functions of the Applications RSL. The Kernel RSL is thus intended to be small and verifiable.

A set of restrictions on the Kernel RSL were formulated, which included:

1. No tasking

2. No explicit dynamic storage allocation using NEW

3. No dynamic arrays

4. No use of Ada standard packages other than STANDARD and SYSTEM

5. No exception handling

Under these restrictions, it turned out that there was no Kernel RSL. It was useful, however, to have a small Kernel RSL to propagate exceptions. Many things, such as I/O, which are usually done for Ada programs by the RSL are not done by the RSL for the Security Kernel, but instead are done directly by the Security Kernel using embedded assembly.

Hardware Issues

It was necessary for the Security Kernel to have the following hardware-related capabilities:

1. Access to privileged instructions

2. Access to specific machine addresses

3. Ability to directly address bits in memory

4. Inline capability

Direct access to memory locations was accomplished using the Ada address and representation clauses. Access to privileged instructions was accomplished by the use of machine language gate routines.

Compiler Issues

The security of ASOS depends on the correct compilation of the Security Kernel. The ASOS Project concluded that the validation and certification procedures for Ada compilers provide an adequate degree of assurance for ASOS.

Appendix G

Trust Domains

Jim Freeman
Ford Aerospace

Jim Freeman: Trust Domains

Jim Freeman presented an approach to stating system requirements and modeling systems called Trust Domains. The approach describes, at a high level of abstraction, the various components of a system, how they are related structurally, and what assumptions they make about the behavior of each other and the system's environment. The goal of the approach is to present a more "understandable" view of the system, thereby providing assurance of the system's correctness.

In the approach developed by Ford Aerospace, a system is represented as a set of trust domains. The structural relationships between domains are described by stating which domains "inhabit" other domains, which domains "contain" other domains, which domains are "associated with" other domains, which domains "adjoin" other domains and which domains "adjoin" other domains "via" a third domain. The assumptions which domains make about other domains are stated in terms of "constraint relationships". A domain A may "receive" a constraint X "from" a domain B, which means that A assumes that B meets constraint X, and a domain A may "derive" a constraint X "for" a domain B, which means that A meets constraint X with respect to B.

Examples of trust domains and relationships were given. Most were in graphical form, with the domains represented as nodes and the relationships as links between nodes. One example was given in a textual form which is processable by tools built by Ford Aerospace. The Trust Domains approach was applied to both WIS and the Multinet Gateway.

Structuring Systems for Formal Verification

Richard B. Neely

James W. Freeman

Ford Aerospace & Communications Corporation

This paper appeared in the Proceedings of the 1985 IEEE Symposium on Computer Security and Privacy, April 22 - 24, 1985 at Oakland, California.

Structuring Systems for Formal Verification

Richard B. Neely

James W. Freeman

Ford Aerospace & Communications Corporation

This research was sponsored in part by the USAF Rome Air Development Center under the Multinet Gateway Program, contract number F30602-81-C-0233.

High levels of assurance for the security of a system are obtained, in part, by the description of its trusted computing base in terms of a formal top-level specification. Nevertheless, the use of a single-level specification can result in an inability to link the behavior of the trusted computing base with the security policy of the system as a whole. This paper discusses that disparity and presents an approach to structuring systems that helps to avoid the problem. Such structuring is shown to be effective in bridging the gap between the trusted computing base itself, and the overall system.

1. INTRODUCTION

It is generally accepted that formal methods can be used to increase the level of assurance that a system is secure. In spite of current improved understanding of such methods, concepts used in describing the trustworthiness of components retain the same limitations they have had for a number of years. Only the most simple and monolithic of systems or components can be characterized by a single "top level specification" -- yet the attempt is made to describe even entire operating systems by such means. Additionally, at the completion of the formal verification of a component, it is often not clear that that verification provides any increased understanding of the component and the overall system. Finally, the trustworthiness of an individual component is typically ensured only by assigning some sensitivity label to it and doing an analysis based on this assignment, rather than establishing its trustworthiness in relation to its individual constraints and requirements.

Some work has been accomplished in deriving security requirements from the environment of the component itself, and what the component actually does. Examples of such work include the development of the "separation kernel" as described by Rushby [10] and related work, the modeling approach described by Bartels and Dinolt [1], and various approaches being investigated within the University of Texas environment [4].

Further investigation and development of such concepts are needed. We introduce the notion of a "trust domain" as a partial answer to that need. A trust domain encapsulates a component in terms of "rule abstraction." This allows a characterization of the component in terms of expectations that it places on its environment and on its own implementation, as well as expectations it is prepared to meet for other domains. Application of the trust domain concept promises reduction of proof complexity, better understanding of the formal specification and verification results, and explicit identification of underlying assumptions. If these goals are realized, then an increased level of assurance will follow.

The remainder of this paper focuses on providing motivation for and application of the trust domain concept. First, some problems to be solved are characterized. Then the limitations of previous structuring attempts to solve the problems are recounted. This is followed by a detailed description of the concept of a trust domain together with an explanation of how the concept helps to solve the problems presented. An extended example of the use of trust domains is provided, including an embodiment of the example in a trust domain representation language. Finally, we demonstrate the utility of the trust domain approach for formal verification in terms of the identified problems, and from this demonstration we draw several conclusions.

2. Problem Statement

Several problems related to the structuring of a system for verification are at least partially solved by the use of trust domains. Those problems are described in this section.

2.1 Problem 1: Assessment of Verification Results.

Consider a moderately complex computer system whose adherence in operation to a given security policy is critical. Suppose that the design, development, and documentation of the system follow the guidelines given in the DoD Computer Security Center's _Trusted Computer System Evaluation Criteria_ [2]. One might now ask exactly what was verified, and further, how do the verification results really contribute to the confidence that the system will not violate the security policy. The answer can be formulated and expressed only in terms of the formal structure of the system, how that structure relates to the system environment, and how the proofs relate to the system structure. On one hand, a small quantity of proof output (e.g., "TRUE" or "FALSE") is easy to understand, but not much confidence is gained from such trivial results. On the other hand, a great deal of output, if it is not well-structured or does not relate clearly to a well-structured system, is impossible to understand well enough to be sure what is proved.

At least three criteria need to be used in assessing the specification and verification output:

1. Proofs are small or at least understandable. The proofs are not just terse, but both complete and simple as possible.

2. The specification information and verification results are clearly related to the actual implemented system.

3. There are well-formed boundary conditions (environmental assumptions).

The latter two criteria are directly tied to the system's "architectural reference points." They allow more meaningful discussion of what is actually modeled, the underlying hardware constraints, assumptions, other non-proved components and code correspondence issues. By an architectural reference point we mean a significant aspect of a system architecture that is taken as given and so must be reflected by not only the system implementation, but also by the formal description of the system. For example, in a system of network gateways, an architectural reference point might include specific aspects of the geographical separation of the gateway nodes. Attributes of the layering of the protocols might form another example. The idea of an "architectural reference point" is that it constrains the allowed design space of the system.

2.2 Problem 2: Verifying System Specific Characteristics.

While the previous discussion was in terms of a "moderately complex" system, distributed systems typically possess an especially complex structure. A noteworthy class of examples is the class of communications systems. In an "ADP," or host, system, a case might be made to consider the interface of an operating system kernel as the exclusive province of formal specification; but there is no analog in a communications environment. The software and hardware that implement network functions are structurally and conceptually far removed from the "system interface," i.e., the network system as seen from the "host users."

2.3 Problem 3: Ensuring Trustworthiness.

Many systems, in enforcing a specific set of requirements, such as a given access control scheme based on sensitivity labels, determine properties and derived requirements that are not directly related to the labeling requirements. Sometimes the traceability between the original and derived requirements is weak because the derivation process has not been well documented. Second, some systems have well-defined security requirements that are not given via sensitivity labels. The result is a problem of truly understanding and agreeing on what _trustworthiness_ means in an environment that may include but is not entirely dependent upon sensitivity labels.

2.4 Problem 4: Domain Reusability.

In order to reduce the cost and, hopefully, technical risk of the specification and verification process, different components that are either identical in function, similar with only parametric differences, or significantly different while assuming or providing similar or identical interfaces -- such different components ought not to have to be specified and verified in a completely independent manner. Some means of reusing the formalism of these components is needed. This concept has been discussed and an approach, based on a notion of reusable problem domain theories, has been suggested by Don Good [4]. Additional work is needed in this area.

3. Previous and Current Structuring Attempts

Previous attempts to provide effective verification results have paved the way to the current state of the art. Although much good work has been accomplished, previous attempts fall short of what we feel is possible now. It is instructive to see in which ways this statement is true, in terms of the three criteria presented in the discussion of the assessment of verification results. Although specific examples in the following discussion refer to the Kernelized Secure Operating System (KSOS), and by extension to Honeywell's SCOMP [11], they are relevant to a wider spectrum of contemporary projects. These examples focus on the assessment of verification results of the first identified problem area.

3.1 Criterion 1: Proof Complexity.

Often, verification conditions (VCs) to be proved have been linked to the formal specification in a way very difficult to trace because they were the result of much syntactic manipulation and other processing. Further, even though the most trivial of the VCs were weeded out early, large numbers of proofs were still necessary. And yet, the true complexity of the results was much greater than it would seem from the quantity of verification results generated. A typical location of hidden complexity was in special-purpose VC generation mechanisms. These observations were true, for example, for the KSOS formal verification [9]. The translation steps from the Special language of the Hierarchical Design Methodology (HDM) into VCs suitable for the Boyer-Moore theorem prover were rather large ones. Examination of intermediate forms (to aid understanding of the translation process) was possible. But such examination only added to the complexity to be digested. The Feiertag VC generator [3] is indeed a very complex program, and for the purpose of analyzing complexity, must be included in the actual proofs to be examined.

In addition, the body of verification results was artificially small because it was not the full system that was involved. In KSOS, the kernel interface, along with relevant internal functionality, was the only part formally specified and verified. Yet what the user of KSOS depends on for enforcing security is the KSOS system as a whole, and so statements about the system as a whole are what needed proving. While the Unix emulator was not to be part of the trusted computing base, that fact needed to be a *result* of the specification and proof process, not an *assumption* of it. Further, the Non Kernel Security Related (NKSR) portion was part of the trusted computing base, yet was not part of any integrated proof process. Were the proof of KSOS, using the HDM-Feiertag technology, to be complete and integrated into a single formal top-level specification (FTLS) as ordinarily conceived, the complexity of the proof would be increased by a large factor.

3.2 Criterion 2: True System Representation.

Accurate representation of the target system also fell short in several ways. Security models were typically simple "flow-upward" models based on a lattice structure of security classifications. However, within the system (and sometimes visible to at least certain users), "exceptions" or special privileges had to be allowed for the sake of correct system operation. These never fit within the model, and so had to be either ignored in specification or else allowed to generate "spurious," unprovable VCs, which were then explained away informally. In addition, because of disparity between the structure of the specification and the implemented code, the informal code correspondence argument in KSOS was not as convincing as had been hoped.

3.3 Criterion 3: Boundary Conditions.

Finally, assumptions necessarily made by the system (but not intended to be proved) were dealt with quite informally, and in fact often were never mentioned but made tacitly. Such assumptions involved hardware and other entities with which the trusted computing base must interface and on which it depended, as well as the initial setup of file system data bases and correct administrative procedures. By handling such issues informally, more unprovable VCs were generated. Consequently the specification and verification results were sometimes confusing, lacked convincing power, and missed the opportunity to point out exactly where certain conditions had to be maintained externally for the system to remain secure.

4. TOWARDS A SOLUTION

Progress has been made in each of the four identified problem areas in recent years. What is needed still is a conceptual framework to aid in organizing this type of specification and verification information. While much work remains to be accomplished in solving the problems described, we have seen initial applications of the trust domain concept to offer an effective step. The motivation for the trust domain idea is provided next via a conceptual description. After the concept is established, details of the structural and constraint relationships are given to explain how to apply the concept.

4.1 Trust Domain Description: Concept

A trust domain is characterized by the following list of attributes:

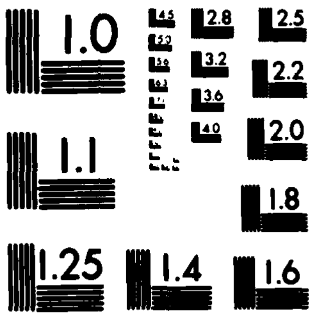
1. It is a part of a system (a component) with a well-defined functional boundary.

2. There are certain properties about its behavior that other domains are entitled to expect.

3. It is entitled to expect the validity of certain assertions about its environment.

AD-A171 507 PROCEEDINGS OF THE IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION O..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C G ROBY DEC 85 IDA-M-146 3/5

UNCLASSIFIED IDA/HQ-85-30658 MDA903-84-C-0031 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

4. It may have internal (non-externally visible) characteristics that are used to provide behavioral guarantees.

The "well-definedness" attribute is essential in producing the limitations on the scope of a trust domain construct to specify what is actually "trusted". A trust domain's trustworthiness is established either by assumptions that may not be proved (but are clearly identified), or else by proofs of assertions based on the environment of the trust domain. Note that part of a trust domain's environment is typically some set of other trust domains. In that case, the guarantees of the other trust domains become part of the environmental assertions of the original trust domain. The assertions to be proved are termed "derived constraints" or resultant theorems. The original trust domain is then said to be constrained by of the other trust domains. The relationship of the other trust domains with the original is spoken of as a constraint relationship. Other constraints external to a trust domain include assertions about the system's environment that must be taken as given in the development process.

The conceptual interface between two trust domains so related consists of functional abstraction (based on functional decomposition); data abstraction (based on private and shared abstract data types); and rule abstraction (based on constraints for function usage and interactions among functions and the data types they govern). A trust domain is pictured in Figure 1.

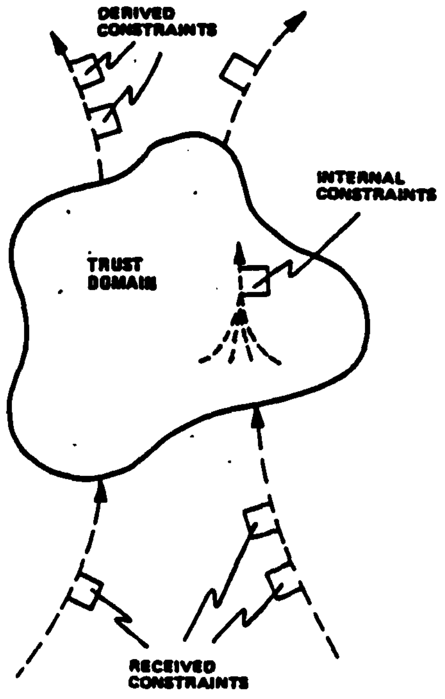


Figure 1. Trust Domain

Trust domains may have not only constraint relationships, but also containment relationships. Figure 2 provides an example of trust domains with constraint relationships; the figure also serves as a basis for illustrating containment relationships.

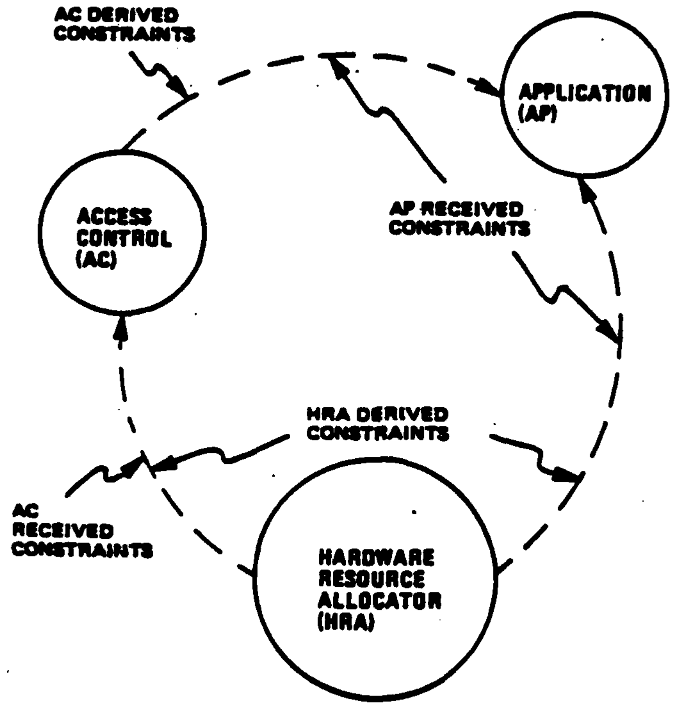


Figure 2. Trust Domain Constraint Concept

Figure 2 could be a picture of the major activities within a simple secure system. The system as a whole is a trust domain that contains the three trust domains shown. This must be so, since it is only about trust domains that properties can be proved, and it is the system as a whole that must be proved secure. Further, the individual trust domains of Figure 2 themselves may (usually at this level, will) contain interior trust domains.

It is important to note that no a priori distinction is made between "trusted" and "untrusted" components, though that distinction falls out of the interpretation of constraint rules. The idea is that external constraints (portrayed as externally applied assumptions) allow a limited scope of activity, so that all that must be proved about the trust domain itself is that, within that limited scope, it will not allow a violation of the constraint rules (its derived constraints) is to enforce. Consequently, a trust domain is "trusted" to satisfy only the derived constraints. If a trust domain is placed so that no expectations need to be applied to its behavior (it has no derived constraints), it is "trusted" to satisfy no constraints. It is then said to be "untrusted". It is this usage in which the term "untrusted" is applied and simply means a trust domain with no resultant theorems or derived constraints. "Trusted", of course, means that derived constraints do exist. Note that this implies that "trusted" is not an absolute term, but is relative to the content of the derived constraints -- i.e., it is always in the sense of "trusted to obey what particular rules".

A key issue, then, in the application of such a concept is the facility to describe the various sets of rules or behavior properties of a given trust domain. The realization of trust domains and associated constraint rules depends on the use of certain established software design techniques, viz., functional abstraction and data type abstraction. A given formally described set of constraints or rules then is represented in terms of the visible functions and the abstract data types.

4.2 Trust Domain Description: Structure and Constraint

The trust domain notion has been described to this point as a structuring concept to of express desired properties and system structure. The previous discussion was to motivate the types of entities needed in a description of a trust domain. The explicit means by which a trust domain is described is outlined now so that specific examples can be described.

A "domain" is an entity with, of necessity, two types of relationships. The types of relationships are structural and constraint. A domain may "adjoin" another domain or "contain" or("inhabit") another domain. These two relationships, adjoin or contain, are structural. If two domains adjoin one another, they cannot contain one another. The identification of which domains adjoin or contain other domains provides a topographical description of the system. A domain may "derive" or "receive" from another domain. These are constraint relationships. Figures 3 and 4 identify and illustrate how the two types of relationships are identified and described (denoted). The following paragraphs provide motivation for such a description.

Suppose domain A adjoins domain B and contains domain C (or C inhabits A). In order to represent actual systems and to describe communications among system entities (whether within a personal computer or among internets), domain are identified as either node or link. This identification is called the gender of the domain. Adjoined domains must be of opposite gender. Figure 3 identifies additional relationships among three domains that share adjoin and contain relationships.

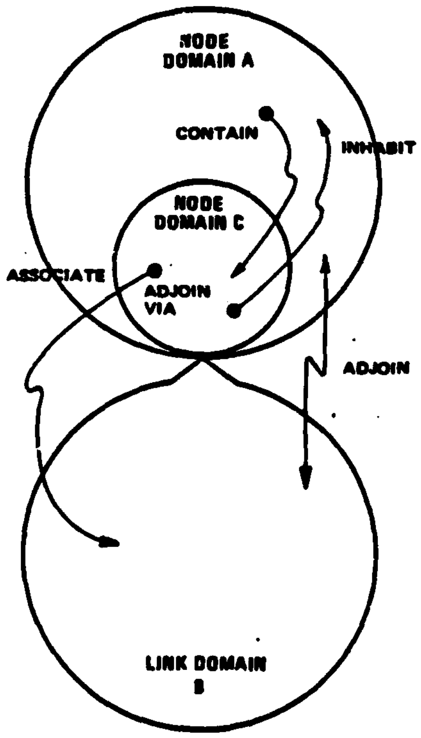


Figure 3. Trust Domain Structure Relationship

First, a domain that inhabits another domain is either a boundary or an interior domain with respect to the containing domain (C is interior to A and is, in fact, a boundary domain of A). Note also that domain A, containing domain C, "associates" C with domain B and C adjoins B "via" A. From A's point of view C is associated with B; from C's point of view C is adjoined to B via A; from B's point of view C is not present or in fact not visible. Thus "adjoin", "adjoin via" and "associates" are distinct structural relationships.

Constraint relationships are now incorporated with the structural relationships. Figure 4 identifies and illustrates this. Note that domain A "derives" constraints Y, Z for domain B, and "receives" constraint X from C; domain B "receives" constraints Y,Z from A and domain C "derives" constraint X for A. Note also that A receives an arbitrary constraint from B if and only if B derives that same constraint for A. This if-and-only-if relationship provides a redundancy to aid in the readibility of the constraints.

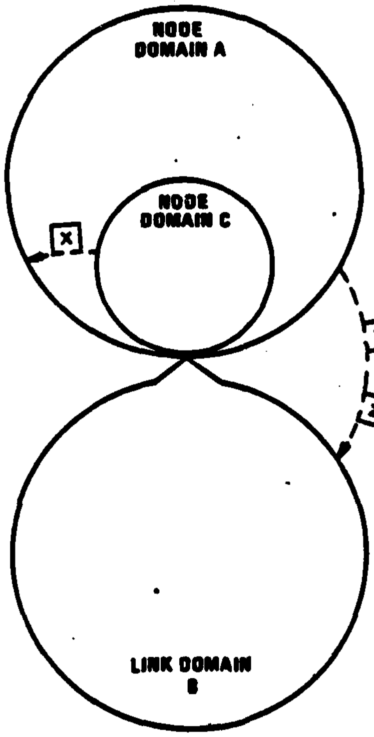


Figure 4. Trust Domain Constraint Relationship

The three domains can be described in terms of a trust domain description language as follows:

```
node domain A shall
    contain boundary node C;
    adjoin link B;
    associate C with B;
    derive for B
        constraint Y;
        constraint Z;
    receive from C
        constraint X;
end A;
```

```
node domain C shall
    inhabit boundary of node A;
    adjoin link B via A;
    derive for A
        constraint X;
end C;

link domain B shall
    adjoin node A;
    receive from A
        constraint Y;
        constraint Z;
end B;
```

It is useful to observe that although the arrows in Figure 3 illustrate the contains, adjoins and associates relationships, they are not integral to the structural description. Such arrows are redundant to the identified relationships. The dotted arrows together with the "boxes" identified in Figure 4 denote what constraints are levied between which domains. They are integral to the description.

Examples of domains using this description are given next.

5. EXAMPLE: STRUCTURING A SYSTEM

This section presents an example that illustrates the previously described aspects of a trust domain. The example is a network system fabricated for purposes of illustration. It includes multiple sites containing local area networks, and inter-site (presumably geographically extensive) transmission. Each site possesses a single mainframe processor operating in multilevel secure mode and multiple workstations. Figure 5 shows the topography of the system.

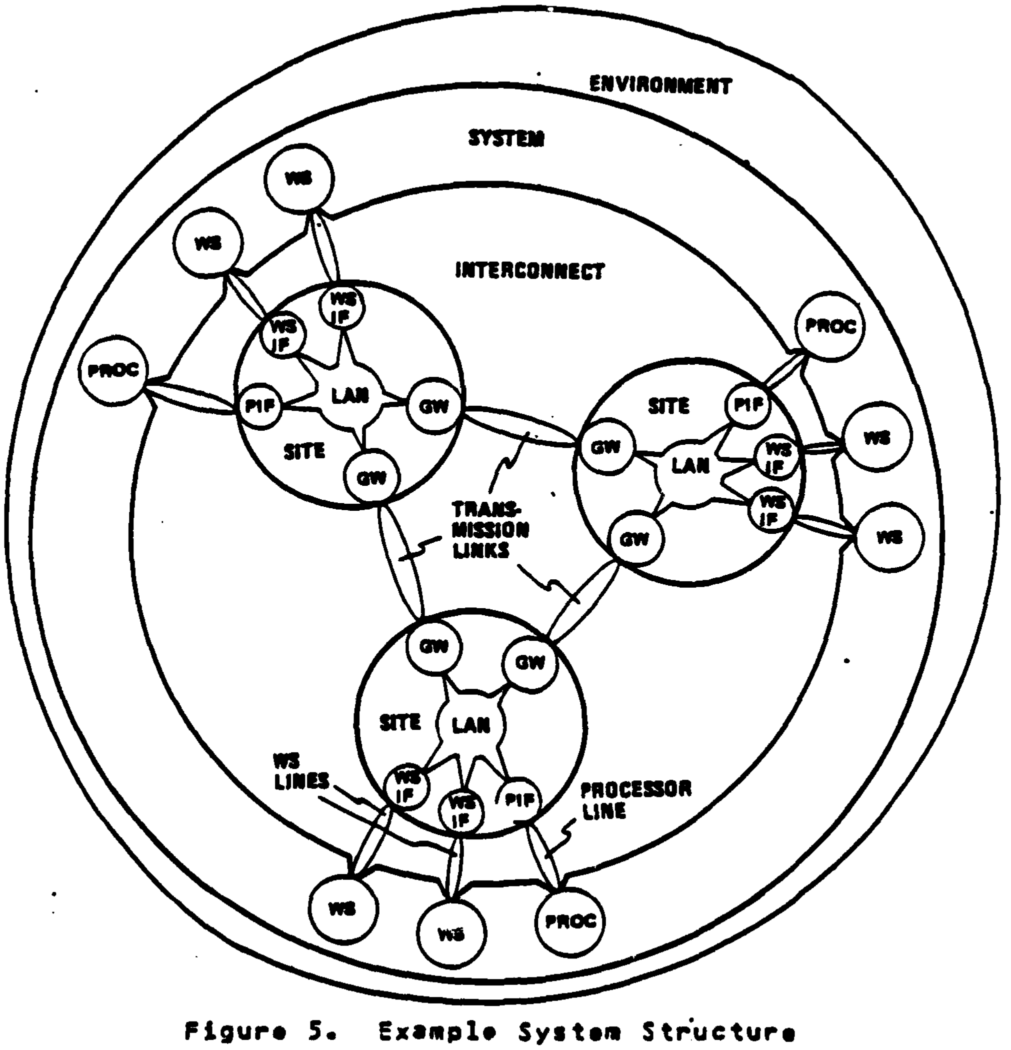


Figure 5. Example System Structure

Note that all Processors and Workstations are considered to be "surface" system components, outside the system communications element (the Interconnect), and thus logically outside of any site. This choice of representation has been seen to provide a clear view of security-related constraints in several actual systems described in terms of trust domains. The level of detail of system representation depicted in Figure 5 might be typical of the initial structuring step performed to provide a security architecture.

Figure 6 describes a portion of the example system, introducing the constraint relationships. In fact Figure 6 is representative of a second step performed in building up a system security architecture. It is characterized by the analysis of the constraints relating the trust domains. The assignment of suggestive names to those constraints lays the groundwork for the addition of rigorous, mathematically oriented representations for each of the constraints.

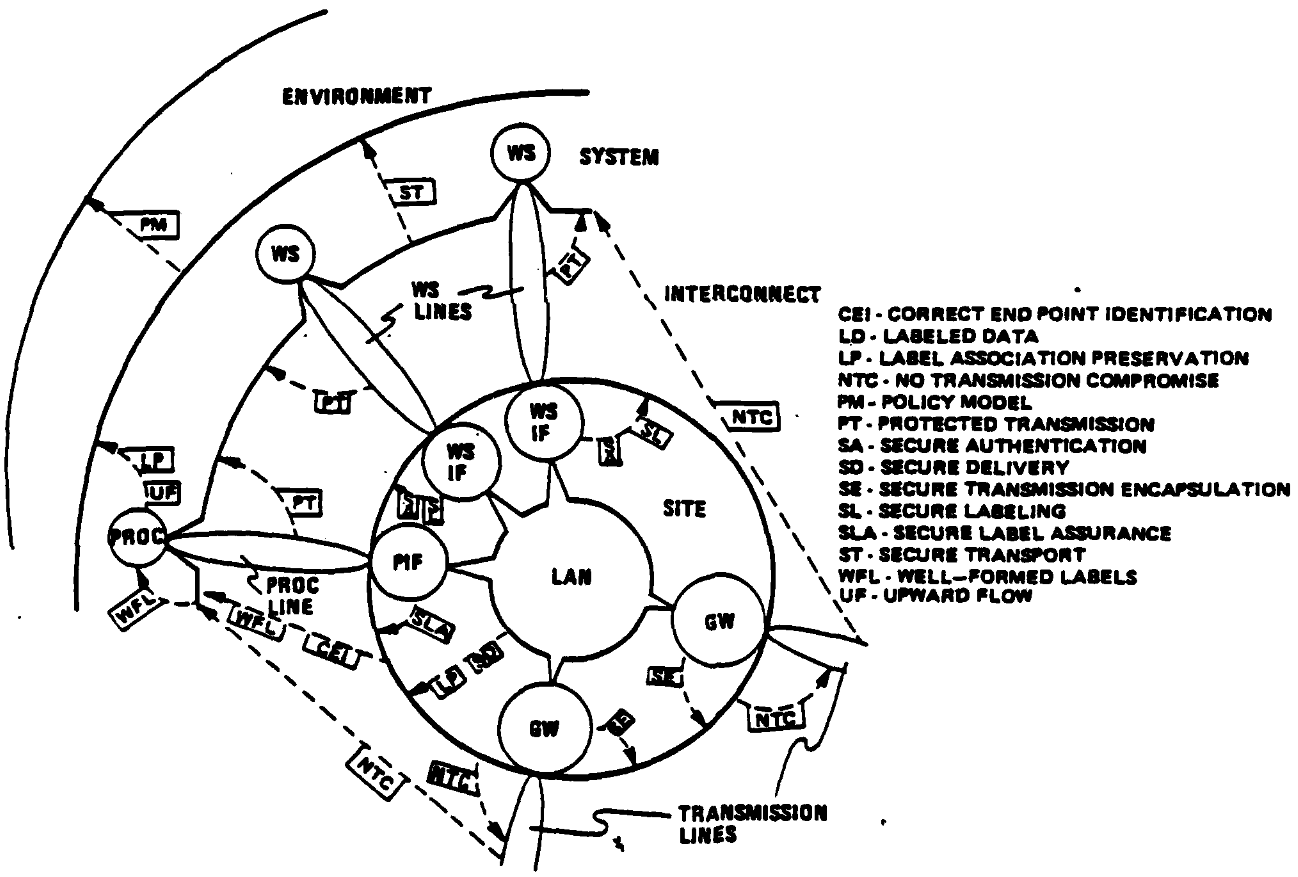


Figure 6. Example System Constraints

Several properties of the constraints in Figure 6 should be noted. A single named property is derived by the system "for the environment." This property is suggestively named "policy_model," which is the set of assertions comprising the model of the system security policy. Other constraints between domains build upon one another, finding their ultimate source in the descriptions of the components of the trusted computing base (and further in its implementation). If these source constraints are correct, and the proofs linking all the constraints are correct, and the trust domain structuring of the system faithfully represents the system's design, then the resultant constraint, the policy_model, has been verified to be valid for the implemented system.

5.1 Trust Domain Description

This subsection contains the text of the description of the example system in terms of the trust domain description language introduced in the description of structural constraints of trust domains.

```
node domain ENVIRONMENT shall
    contain
        interior node System;
    receive from System
        constraint policy_model;
end ENVIRONMENT;

node domain System shall
    inhabit interior of node ENVIRONMENT;
    contain
        interior link Interconnect;
        interior node Processor multiple;
        interior node Workstation multiple;
    derive constraint policy_model;
    receive
        from Interconnect
            constraint secure_transport;
        from Processor
            constraint labeled_data;
            constraint upward_flow_only;
        from Workstation NO_CONSTRAINTS;
end System;

node domain Workstation shall
    inhabit interior of node System;
    adjoin link Interconnect;
    derive NO_CONSTRAINTS;
end Workstation;
```

```
node domain Processor shall
    inhabit interior of node System;
    adjoin link Interconnect;
    derive for System
        constraint labeled_data;
        constraint upward_flow_only;
end Processor;

link domain Interconnect shall
    inhabit interior of node System;
    contain
        boundary link Workstation_Line multiple;
        boundary link Processor_Line multiple;
        interior node Site multiple;
        interior link Transmission_Link multiple;
    adjoin
        node Workstation;
        node Processor;
    associate
        Workstation_Line with Workstation one_to_one;
        Processor_Line with Processor one_to_one;
    derive
        for System
            constraint secure_transport;
        for Processor
            constraint well_formed_labels;
    receive
        from Site
            constraint well_formed_labels;
            constraint correct_endpoint_ID;
        from Workstation_Line
            constraint protected_transmission;
        from Processor_Line
            constraint protected_transmission;
        from Transmission_link
            constraint no_trans_compromise;
end Interconnect;

link domain Workstation_Line shall
    inhabit boundary of link Interconnect;
    adjoin
        node Site;
        node Workstation via Interconnect;
    derive for Interconnect
        constraint protected_transmission;
end Workstation_Line;
```

```
link domain Processor_Line shall
    inhabit boundary of link Interconnect;
    adjoin
        node Site;
        node Processor via Interconnect;
    derive for Interconnect
        constraint protected_transmission;
end Processor_Line;

link domain Transmission_Link shall
    inhabit interior of link Interconnect;
    adjoin node Site many_to_many;
    derive for Interconnect
        constraint no_trans_compromise;
    receive from Site
        constraint no_trans_compromise;
end Transmission_Link;
```

```
node domain Site shall
    inhabit interior of link Interconnect;
    contain
        boundary node Workstation_LAN_IF;
        boundary node Processor_LAN_IF;
        boundary node Gateway;
        interior link Local_Area_Network;
    adjoin
        link Workstation_Line one_to_many;
        link Processor_Line;
        link Transmission_Link many_to_many;
    associate
        Workstation_LAN_IF
            with Workstation_Line one_to_one;
        Processor_LAN_IF
            with Processor_Line one_to_one;
        Gateway with Transmission_Link one_to_one;
    derive
        for Interconnect
            constraint well_formed_labels;
            constraint correct_endpoint_ID;
        for Transmission_Link
            constraint no_trans_compromise;
    receive
        from Local_Area_Network
            constraint label_assoc_preservation;
            constraint secure_delivery;
        from Workstation_LAN_IF
            constraint secure_authentication;
            constraint secure_labeling;
        from Processor_LAN_IF
            constraint secure_label_assurance;
        from Gateway
            constraint secure_trans_encapsulation;
end Site;

node domain Workstation_LAN_IF shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network;
        link Workstation_Line via Site;
    derive for Site
        constraint secure_authentication;
        constraint secure_labeling;
end Workstation_LAN_IF;
```

```
node domain Processor_LAN_IF shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network;
        link Processor_Line via Site;
    derive for Site
        constraint secure_label_assurance;
end Processor_LAN_IF;

node domain Gateway shall
    inhabit boundary of node Site;
    adjoin
        link Local_Area_Network;
        link Transmission_Link via Site;
    derive for Site
        constraint secure_trans_encapsulation;
end Gateway;

link domain Local_Area_Network shall
    inhabit interior of node Site;
    adjoin
        node Workstation_LAN_IF one_to_many;
        node Processor_LAN_IF;
        node Gateway one_to_many;
    derive for Site
        constraint label_assoc_preservation;
        constraint secure_delivery;
end Local_Area_Network;
```

5.2 Selected Detailed Examples

The example in Figure 6 is complete in its identification of major structures and constraints in terms of trust domains. Two areas of expansion remain to be completed. The first is the elaboration of each of the constraints to include a complete mathematical representation of the constraint. Such representation will typically follow the standards of the specification language into which the trust domains are to be mapped. In general, this will involve expressions (e.g., predicate calculus), which are in fact prescribed by the existing trust domain language grammar. The second area is the description of the implementation requirements for the hardware and software interfaces so that they can be related to the rest of structure of the system, including the interface of the system itself. The following two subsections present a limited example of the identified areas.

5.2.1 Constraint Elaboration The trust domain chosen for constraint elaboration is the Processor, depicted in Figure 6 with the notation "Proc." Following is the Processor trust domain description with constraints elaborated in terms of predicate calculus expressions.

```
node domain Processor shall
     inhabit interior of node System;
     adjoin link Interconnect;
     derive for System
          constraint labeled_data:
               FORALL datum,
                    is_valid_label(class(datum));
          constraint upward_flow_only:
               FORALL datum,proc,
                    canread(proc,datum) =>
                         dominates(class(proc),
                                   class(datum))
                    AND canwrite(proc,datum) =>
                         dominates(class(datum),
                                   class(proc));
     specify
          type element, proc_subject,
               proc_object, label;
          variable datum: proc_object;
          variable proc: proc_subject;
          function is_valid_label(label):
                         boolean;
          function class(element): label;
          function
               canread(proc_subject,proc_object):
                         boolean;
          function dominates(label,label): boolean;
end Processor;
```

Note the "specify" clause added to the domain that allows truncated declarations of types, variables, and functions (also constants) to clarify the predicate calculus expressions. With the full form of the constraints, the suggestive names are retained; the two parts of each constraint thus complement each other.

5.2.2 Software-Related Example The example of this section has so far only been described in terms of its major system structures. This description will now be augmented by a selected example that relates the interface of a portion of the trusted computing base to the rest of the system structure.

The domain to be so augmented is the Processor LAN interface (Processor_LAN_IF). A pictorial description of that domain is given in Figure 7.

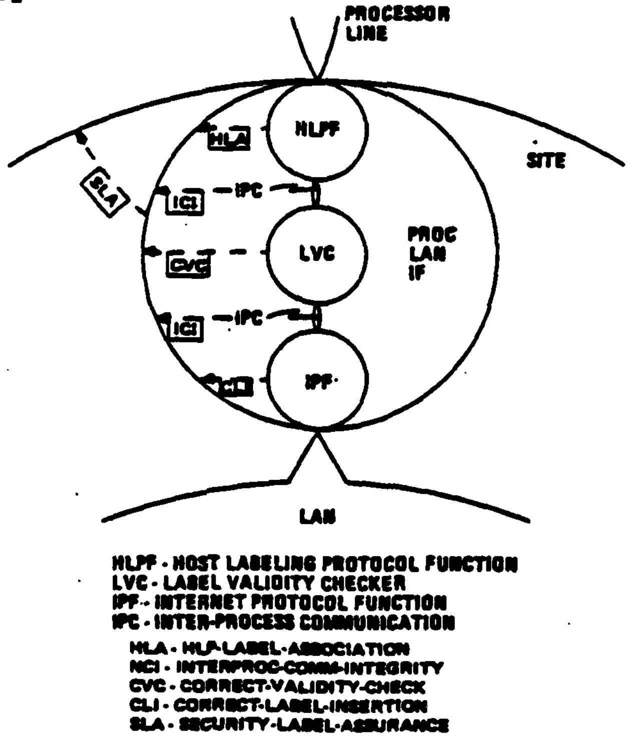

PROCESSOR LINE
HLA
HLPF
SITE
SLA
ICI
IPC
CVC
LVC
PROC LAN IF
ICI
IPC
CLI
IPF
LAN
HLPF - HOST LABELING PROTOCOL FUNCTION
LVC - LABEL VALIDITY CHECKER
IPF - INTERNET PROTOCOL FUNCTION
IPC - INTER-PROCESS COMMUNICATION
HLA - HLP-LABEL-ASSOCIATION
ICI - INTERPROC-COMM-INTEGRITY
CVC - CORRECT-VALIDITY-CHECK
CLI - CORRECT-LABEL-INSERTION
SLA - SECURITY-LABEL-ASSURANCE

Figure 7. Software Domains of Processor_LAN_IF

It consists of three software domains, a Host Labeling Protocol Function, a Label Validity Checker, and an Internet Protocol Function. These domains communicate via a multiply instantiated Inter-Process Communication link domain. As with the previously presented portion of the example, these internal domains derive constraints that allow the proof of the constraint that the Processor_LAN_IF derives for the Site (viz., secure_label_assurance).

This domain, as detailed in Figure 7, is now presented in terms of the trust domain description language.

```
node domain Processor_LAN_IF shall
    inhabit boundary of node Site;
    contain
        boundary node Host_Labeling_Protocol_Fcn;
        boundary node Internet_Protocol_Fcn;
        interior node Label_Validity_Checker;
        interior link Inter_Process_Comm multiple;
    adjoin
        link Local_Area_Network;
        link Processor_Line via Site;
    associate
        Host_Labeling_Protocol_Fcn with Processor_Line;
        Internet_Protocol_Fcn with Local_Area_Network;
    derive for Site
        constraint secure_label_assurance;
    receive
        from Host_Labeling_Protocol_Fcn
            constraint HLP_label_association;
        from Internet_Protocol_Fcn
            constraint correct_label_insertion;
        from Label_Validity_Checking
            constraint correct_validity_check;
        from Inter_Process_Comm
            constraint interproc_comm_integrity;
end Processor_LAN_IF;

node domain Host_Labeling_Protocol_Fcn shall
    inhabit boundary of node Processor_LAN_IF;
    adjoin
        link Inter_Process_Comm one_to_one;
        link Processor_Line via Processor_LAN_IF;
    derive for Processor_LAN_IF
        constraint HLP_label_association;
end Host_Labeling_Protocol_Fcn;

node domain Internet_Protocol_Fcn shall
    inhabit boundary of node Processor_LAN_IF;
    adjoin
        link Inter_Process_Comm one_to_one;
        link Local_Area_Network via Processor_LAN_IF;
    derive for Processor_LAN_IF
        constraint correct_label_insertion;
end Internet_Protocol_Fcn;

node domain Label_Validity_Checker shall
    inhabit interior of node Processor_LAN_IF;
    adjoin link Inter_Process_Comm one_to_many;
    derive for Processor_LAN_IF
        constraint correct_validity_check;
end Label_Validity_Checker;
```

```
link domain Inter_Process_Comm shall
    inhabit interior of node Processor_LAN_IF;
    adjoin
        node Host_Labeling_Protocol_Fcn one_to_one;
        node Internet_Protocol_Fcn one_to_one;
        node Label_Validity_Checker many_to_one;
    derive for Processor_LAN_IF
        constraint interproc_comm_integrity;
end Inter_Process_Comm;
```

6. EMBODIMENT IN SPECIFICATION LANGUAGES

The intented result of the description of a system security architecture in terms of trust domains is to provide the foundation for a well-structured formal specification of the system using an established specification language. Some thought has been given to the applicability of the trust domain description language to be mapped to several formal specification languages, as shown in Figure 8.

Trust Domain	HDM	Anna	Gypsy
domain	abs. machine/ module/ OFUN	package/ procedure	scope/ function
inhabit/ contain	syntactic context	syntactic context	syntactic context
adjoin	EFFECTS_OF	call	call
realization	ILPL function	generics	--
instantiation	--	generics	--
presentation	external	with/use	from

Figure 8. Association of Trust Domain Constructs With Formal Specification Languages

Work is currently under way in the Rome Air Development Center's Multinet Gateway Certification Program at Ford Aerospace. Our work in this program includes establishing more strongly the mapping from trust domains to the Gypsy language, in which the Multinet Gateway System is being specified.

7. EFFECTIVENESS OF TRUST DOMAINS

Four problems have been identified regarding the structuring of a system for verification. The concept of a trust domain is being examined and applied to each of the four problems within the context of Multinet Gateway and other programs. The status and results of the investigation with respect to each of the problems is given in the following subsections.

7.1 The Verification Result Assessment Criteria

Figures 4, 6, and 7, along with the associated descriptions in the trust domain language, demonstrate how the verification result criteria are more effectively met by structuring the system in this way. The received constraints of the most fundamental domains specifically provide the relationships to the system implementation, and to system boundary conditions. Further, the whole verification (particularly proof) process is made more comprehensible by the subdivision of the work into smaller pieces. For each domain, the received constraints, along with internal constraints obtained from formal statements about the implementation, are used to prove the derived constraints of the domain. If the domains have been delineated to the right degree of detail, then each such proof will not be very hard to follow intuitively. Such decomposition of structure and proof have been part of some formal design approaches (e.g., [5], [6], [7], [8]), including this trust domain approach. That this approach allows recomposition using the identified constraints as a closure mechanism is a key to the effectiveness of this approach. Hence, the domain proofs are cascaded on the basis of _modus ponens_ according to the topographic and constraint relationships identified for all the trust domains of the system. This cascading is the basis for the recomposition. The resultant theorem of this cascade of proofs, of course, is the policy model itself. It is generally accepted that a proof is made more understandable if it is broken into smaller pieces. Further, in order for the verification of a system to be understandable, not only must the proof itself be understood, but the relationship of the structure of the proof to the system and its security policy must be clearly understood. These conditions for understandability are met by the trust domain based description of the system structure and related constraints.

7.2 Special System Specific Requirements: Network Systems

The structure of a network itself intensifies the complexity problem. The exterior interface of the system trusted software and hardware is specifically not the interface seen by system users (connected hosts). Thus, some sort of mapping between the interface of the trusted computing base and the interface to the hosts must be provided. Also, in order to verify the system, constraint relationships are necessary to go along with the structural relationships to relate the trusted computing base interface(s) to the user interface. We have demonstrated that such a mapping is provided by decomposing a system into trust domains whose structural and constraining relationships are explicitly identified. In this way, the complexity problem is effectively addressed.

7.3 Ensuring Trustworthiness

The ability of trust domains to allow explicit characterization of formal requirements on system components allows a departure, when appropriate, from ensuring the trustworthiness of a component by the assignment of a security label to the component and limiting the component's behavior only on that basis. The labeling mechanism is certainly appropriate at times; but the use of trust domains allows a choice based on system analysis.

7.4 Reusable Domains

Based on examples generated from Multinet Gateway work, the verification of components with the three identified kinds of similarities stated in Problem 4 can be streamlined by reusing similar trust domains. This is a direct application of methods suggested by Don Good [4]. The three kinds of similarities are (1) actually identical in function (provided by the trust domain "multiple" construct); (2) differing only parametrically (provided by the trust domain "instantiation" construct); and (3) different in internal function while providing similar or identical interfaces (provided by the trust domain "realization" construct).

8. CONCLUSION

The trust domain concept generalizes the idea of trustedness, encompassing the concepts of untrusted components and components of the trusted computing base. The meaning of trustedness can be better described; it is also possible to describe trusted components that are distributed throughout a system.

One aspect common to much of the formal specification and verification work accomplished in past years is that in order to perform a formal verification of a real system, the system has had to be abstracted to a level compatible with the verification method and available tools. Current specification and verification methods cannot directly handle many aspects of actual systems and the result has been a single highly abstracted level. The single or "top" level specification varied from system to system. For relatively small systems the necessary degree of abstraction was not so severe that the designer and evaluator could not intellectually grasp the association between the real system and an abstraction of it. People have seemed to feel comfortable about being able to handle the inherent "complexity gap" by informal memory devices either in the designer's mind or functional descriptions written with whichever specification language the designer felt most comfortable. Attempts have been made to apply directly such concepts to larger or functionally more complex systems. Problems have arisen as a consequence. Attempts to use traditional conceptual frameworks to structure a system and to specify formally and verify more complex systems have engendered an enormous "complexity gap." Such a gap became so large that it was intellectually difficult, or even impossible, to describe a useful relationship between a system's abstraction and its realization on a hardware base. Nearly every methodology developed has had, as a primary purpose, that of reducing this gap.

Are we, with trust domains, introducing yet another methodology and additional layers of abstraction to address a set of problems that an experienced systems designer or software engineer has been facing and solving for some time? In answer, the question should be asked: How well have we been solving the problems, particularly with respect to specifying and verifying properties that directly aid the increased understanding and correctness a system, and thereby increasing the assurance that the system operates according to its behavioral (e.g., security) policy?

Based on an initial assessment and current status of some examples, the use of the trust domain concept is a consistent approach for organizing the relationships among the functional processing requirements, the types of data, architectural reference points, and most importantly, the constraints that are often left unstated or implicitly assumed in the specification and verification of a system or a component. The concept can be incorporated into a formal specification language and verification approach as is presently being done in terms of Gypsy. The conceptual framework can be applied consistently from a level of specification from an "outside user" of a component to a description of how resources can be allocated within a processor.

The trust domain concept is not "yet another methodology." The concept allows the identification and description of a component and the relationships among components that are to be realized within the system. This overall approach, in effect, raises the question of whether a verified system must be an understandable system. We answer in the affirmative.

9. ACKNOWLEDGEMENT

The authors acknowledge the insightful contributions of George W. Dinolt and Peter C. Baker (both of Ford Aerospace) that were provided during very many hours of fruitful discussion. The authors also acknowledge the dedicated support of Michael D. Kranzin (also of Ford Aerospace) in validating many of the ideas presented here.

10. REFERENCES

[1] Denise E. M. Bartels, George W. Dinolt, "A Partition Based Security Model", WDL-TR9780, Ford Aerospace and Communications Corporation, Palo Alto, California, June 23, 1983.

[2] DoD Computer Security Center, Department of Defense Trusted Computer System Evaluation Criteria, CSC-STD-001-83, Fort George G. Meade, Maryland, August 15, 1983.

[3] R. J. Feiertag, "A Technique for Proving Specifications are Multilevel Secure," Tech. Report CSL-109, Computer Science Lab., SRI International, Menlo Park, California, January 1980.

[4] Donald I. Good, "Reusable Problem Domain Theories", Technical Report 31, Institute for Computing Science, University of Texas at Austin, Austin, Texas, September 1982.

[5] Donald I. Good, "Structuring a System for A1 Certification", Internal Note #145-A, Institute for Computing Science, University of Texas at Austin, Austin, Texas, September 1984.

[6] Donald I. Good, Ann E. Siebert, Lawrence M. Smith, "KAIS FEU Design - Volume I", Internal Note #146-A, Institute for Computing Science, University of Texas at Austin, Austin, Texas, September 1984.

[7] Donald I. Good, "XAIS FEU Design - Volume II", Internal Note #147-A, Institute for Computing Science, University of Texas at Austin, Austin, Texas, September 1984.

[3] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 1C, pp. 549-557, October 1974.

[9] Secure Minicomputer Operating System (KSCS), "Computer Program Development Specifications (Type B5)", WDL-TR7932, WDL-TR7933, WDL-TR7934, Ford Aerospace and Communications Corporation, Palo Alto, California, December, 1980.

[10] John M. Rushby, "The Design and Verification of Secure Systems." Proc. 3th ACM Symposium on Operating System Principles, Asilomar CA., pp 12-21, December 1981, (ACM Operating Systems Review, Vol. 15, No. 5).

[11] Johnathan M. Silverman, "Reflections on the Verification of the Security of an Operating System Kernel," Proc. 9th ACM Symposium on Operating System Principles, Bretton Woods, N.H., pp. 143-154, October 1983, (ACM Operating Systems Review, Vol. 17, No. 5).

STRUCTURING SYSTEMS FOR UNDERSTANDING:

RELATIONSHIP TO ADA

J. W. Freeman, R. B. Neely

Presented by
James W. Freeman

Workshop on Formal Specification and Verification in Ada

Institute for Defense Analysis
18-20 March 1985

Ford Aerospace &
Communications Corporation

PURPOSE

ACCEPTED:	Formal methods can provide increased assurance
NOTE:	Concepts describing "trustworthiness" unchangec
FACT:	Increased user demands result in more complex systems
CLAIM:	FTLS is not sufficient to describe many systems
NEED:	To develop better descriptions of system and component "trustworthiness"
IDENTIFIED:	An approach to "trustworthiness" description along with relationship to Ada

Ford Aerospace & Communications Corporation

OUTLINE

* Relationship of presentation to Workshop topics
* Focus on the approach
* Example problem
* Approach: Trust Domains
* Illustrations of approach
* Ada connections
* Conclusion

Ford Aerospace &
Communications Corporation

RELATIONSHIP TO WORKSHOP TOPICS: APPROACH

* Near-term verification
  - structuring systems for verification
  - mapping to Ada and Anna
  - not tied to modification of existing spec. & verif. environments

* Long-term verification & Ada verif. environment stds.
  - provides structuring and constraint guidelines

* Restrictions on Use of Ada
  - provides vehicle for restrictions on Ada

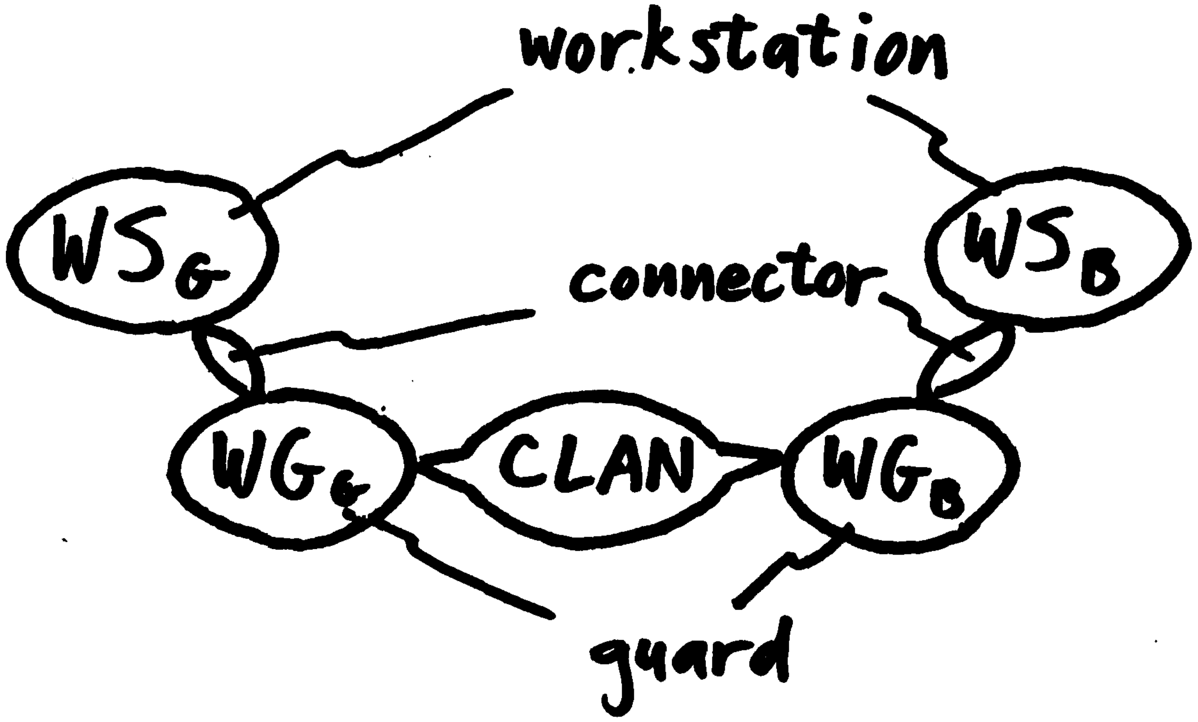
Ford Aerospace & Communications Corporation

KEY POINTS

* Trust Domains include structural and constraint relationships
* Distinction between abstraction and decomposition
* Elements categorized as nodes and links

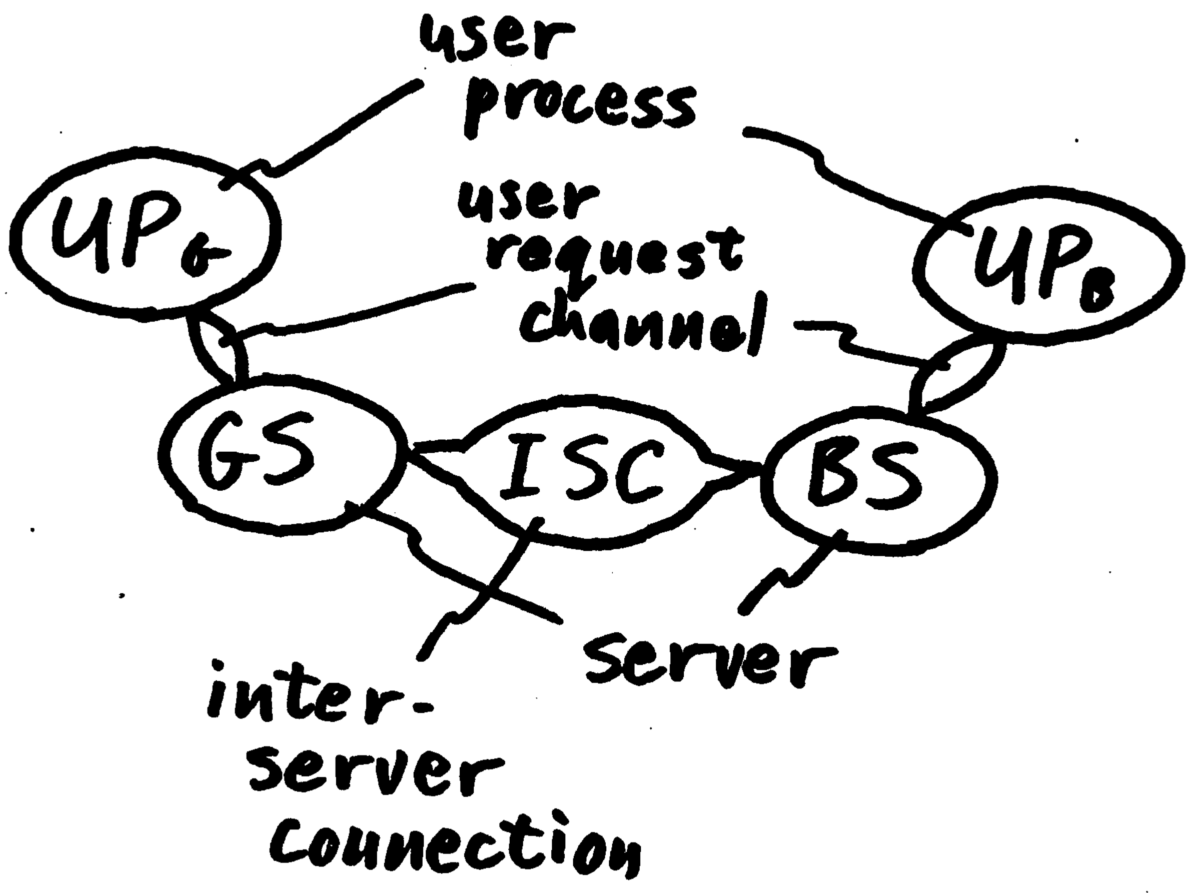
Ford Aerospace & Communications Corporation

EXAMPLE PROBLEM INTRODUCTION: NETWORK FLAVOR'



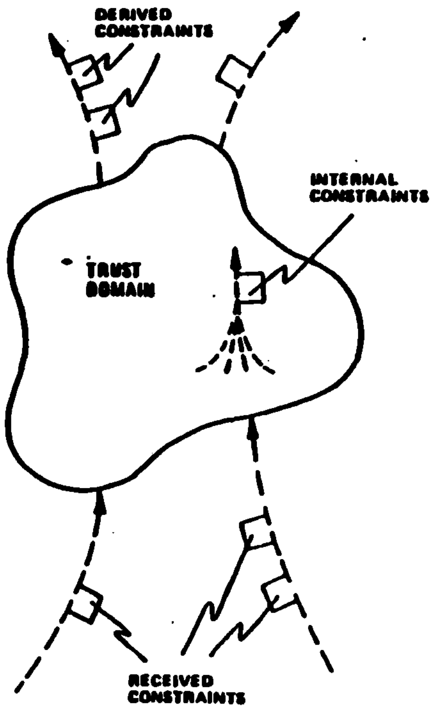
Ford Aerospace &, Communications Corporation

EXAMPLE PROBLEM INTRODUCTION: HOST FLAVOR



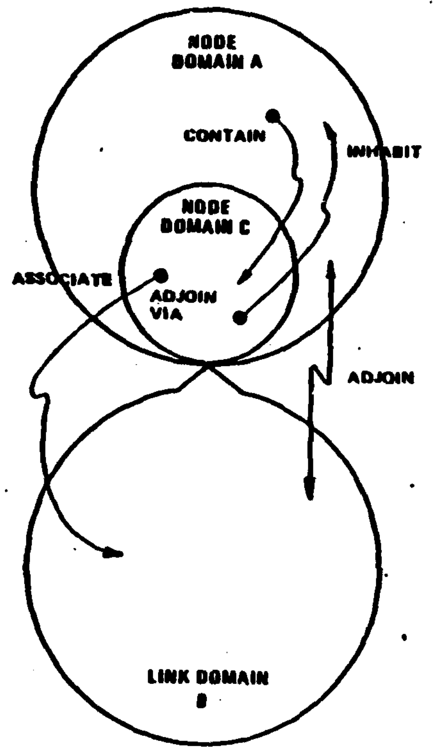
Ford Aerospace & Communications Corporation

TRUST DOMAIN



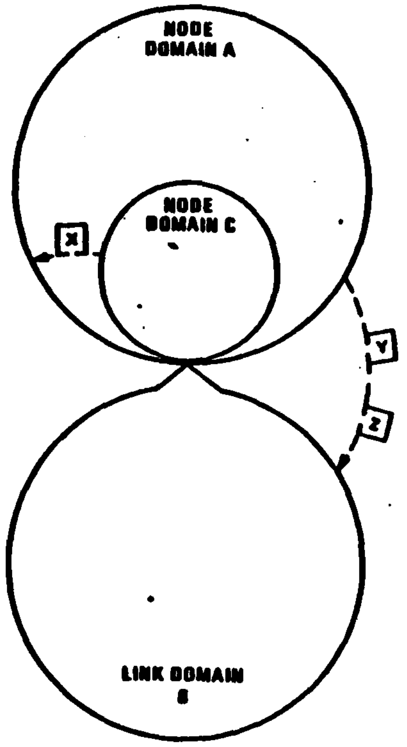
Ford Aerospace & Communications Corporation

TRUST DOMAIN STRUCTURE RELATIONSHIPS



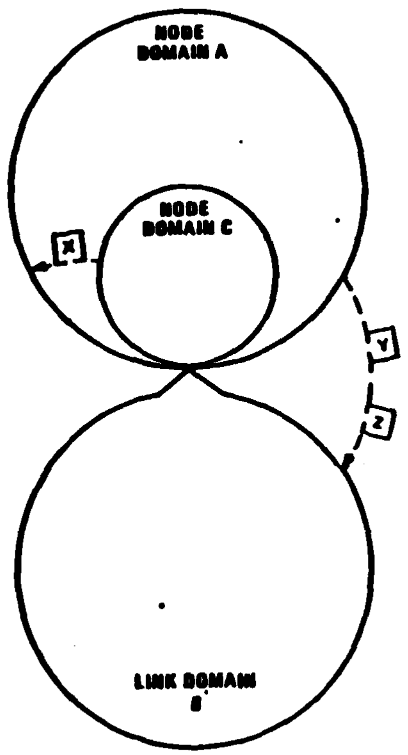
Ford Aerospace & Communications Corporation

TRUST DOMAIN CONSTRAINT RELATIONSHIPS



Ford Aerospace & Communications Corporation

TRUST DOMAIN LANGUAGE INTRODUCTION



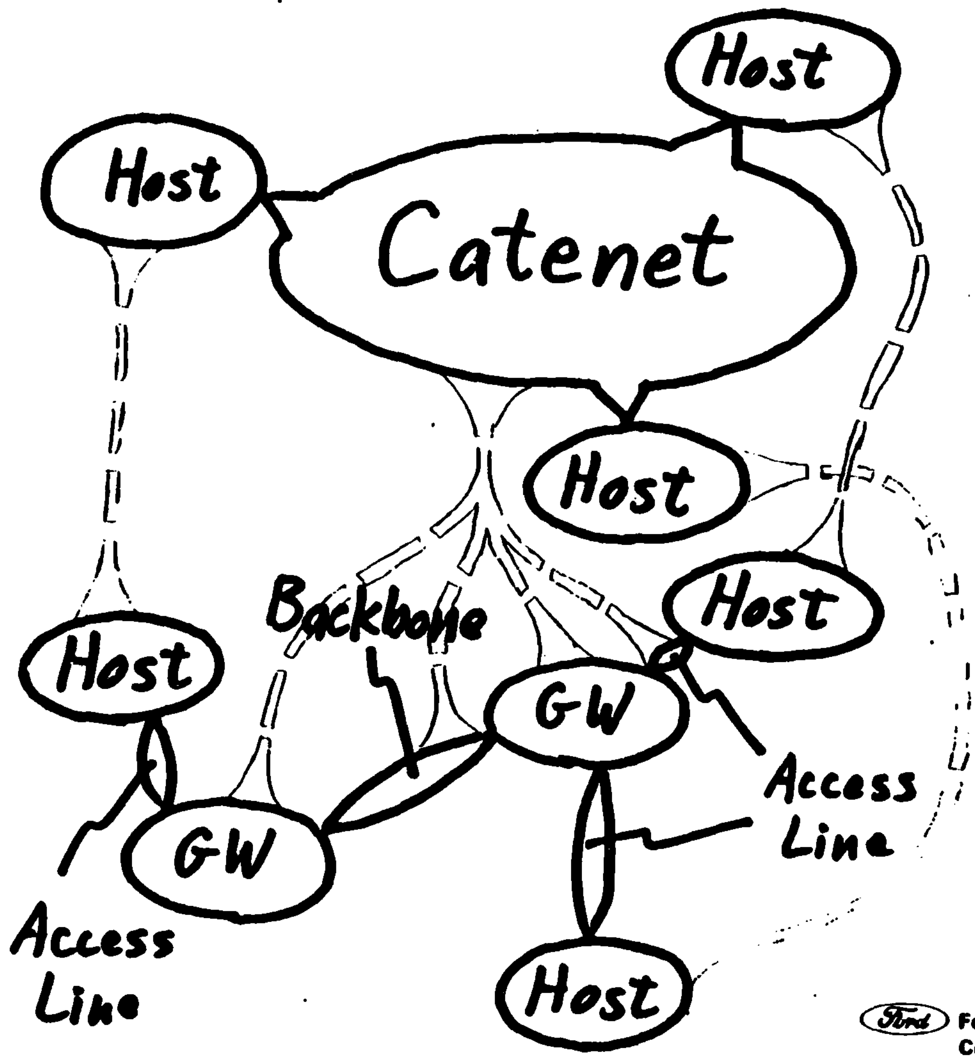
```
node domain A shall
    contain boundary node C;
    adjoin link B;
    associate C with B;
    derive for B
        constraint Y;
        constraint Z;
    receive from C
        constraint X;
end A;


node domain C shall
    inhabit boundary of node A;
    adjoin link B via A;
    derive for A
        constraint X;
end C;

link domain B shall
    adjoin node A;
    receive from A
        constraint Y;
        constraint Z;
end B;
```

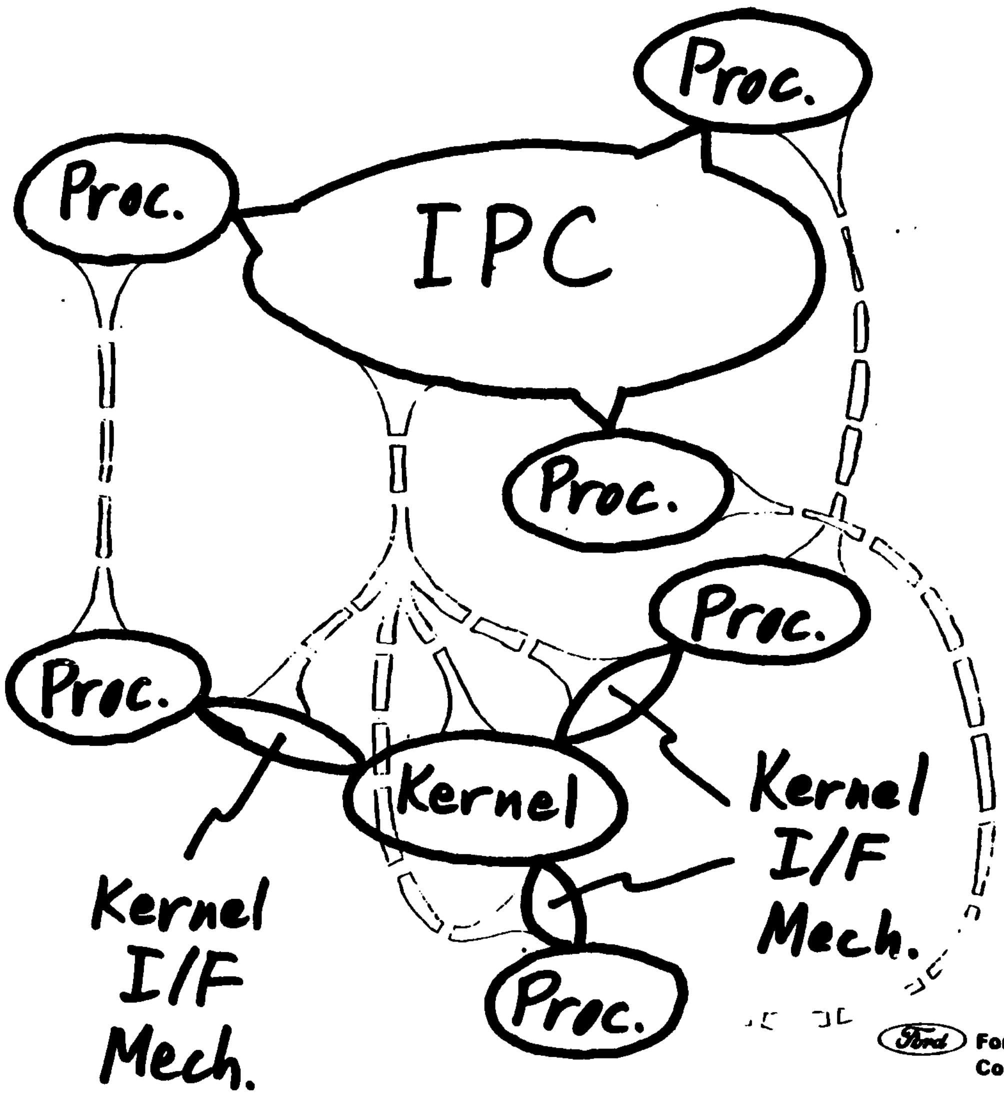
Ford Aerospace &
Communications Corporation

NETWORK DECOMPOSITION



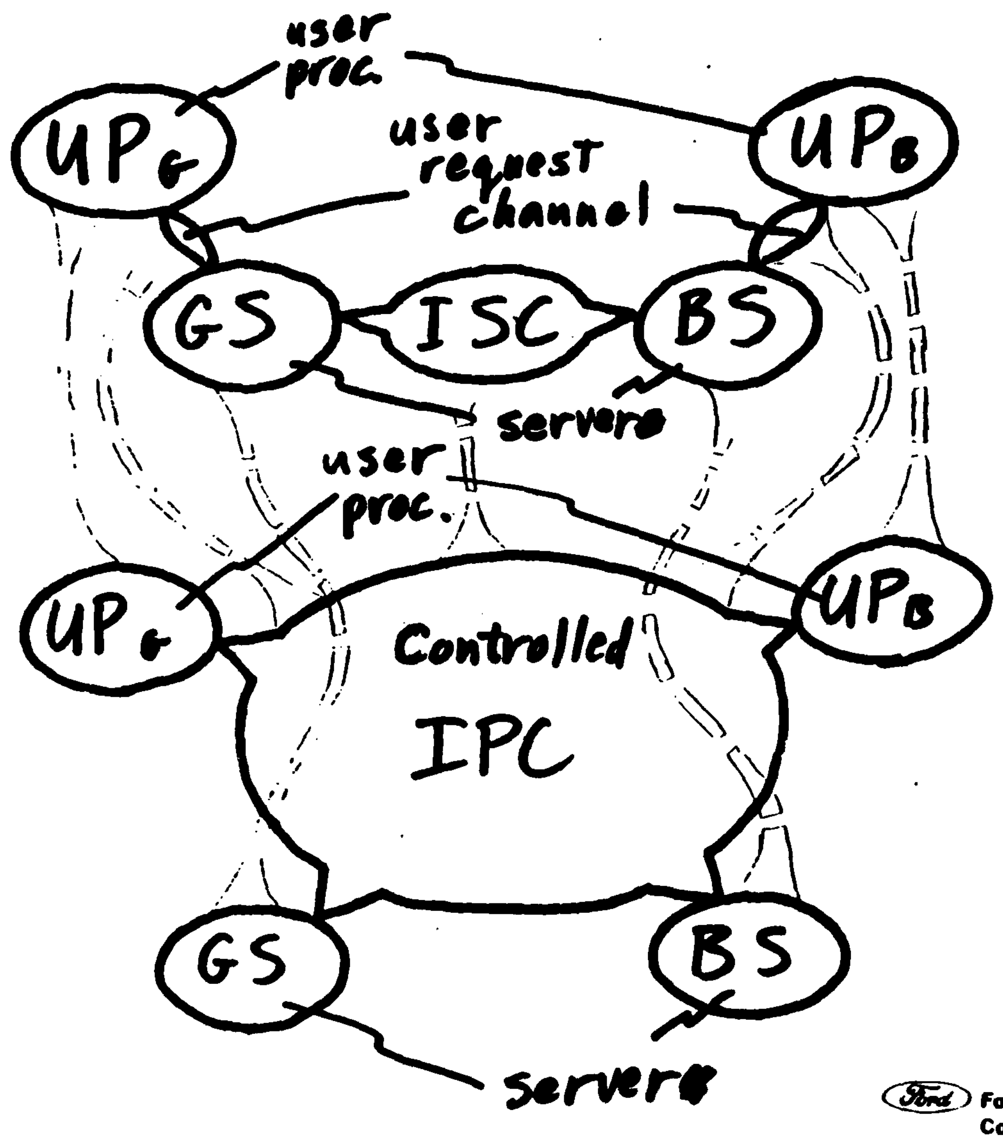
Ford Aerospace & Communications Corporation

HOST DECOMPOSITION



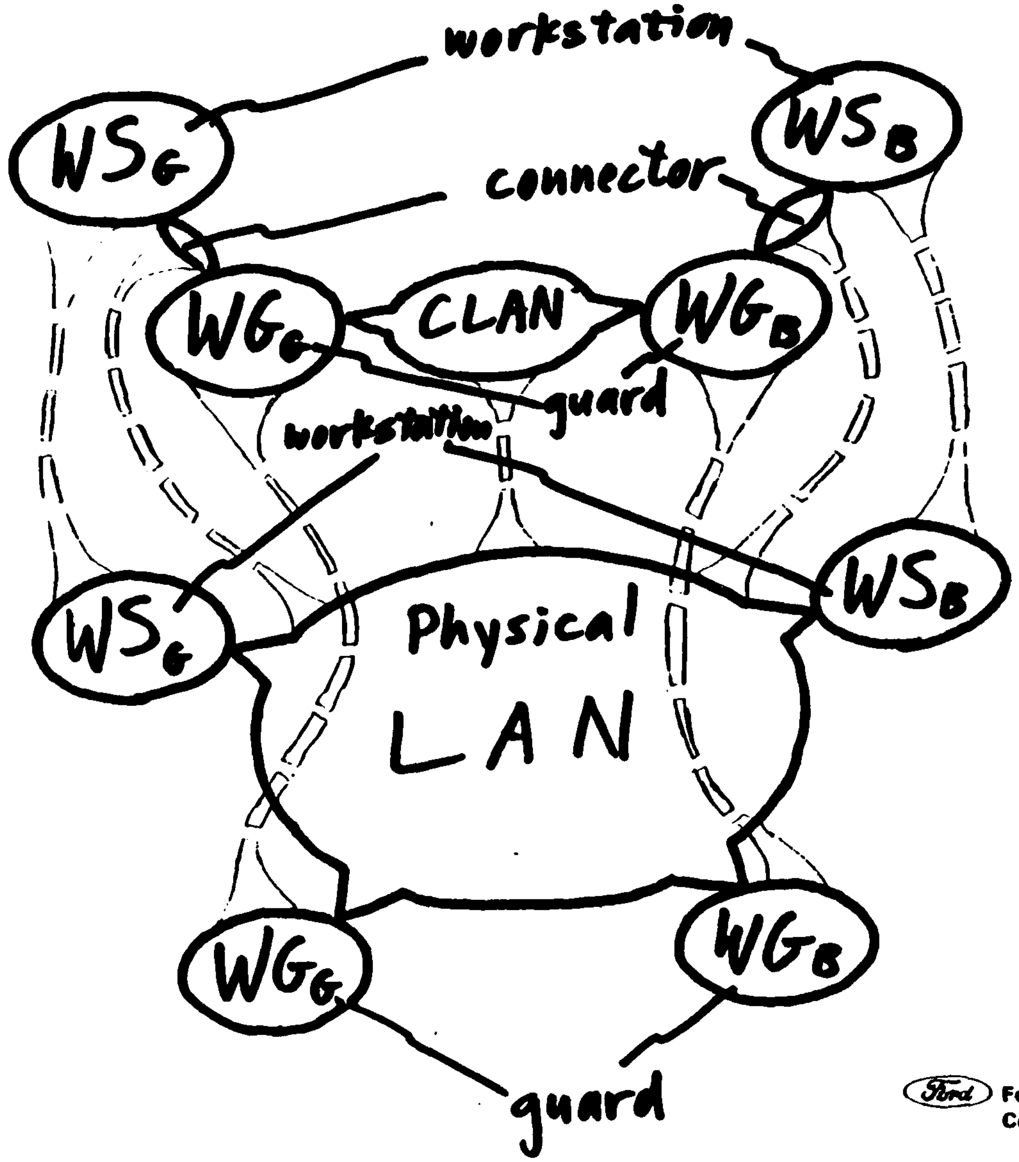
Ford Aerospace & Communications Corporation

HOST ABSTRACTION



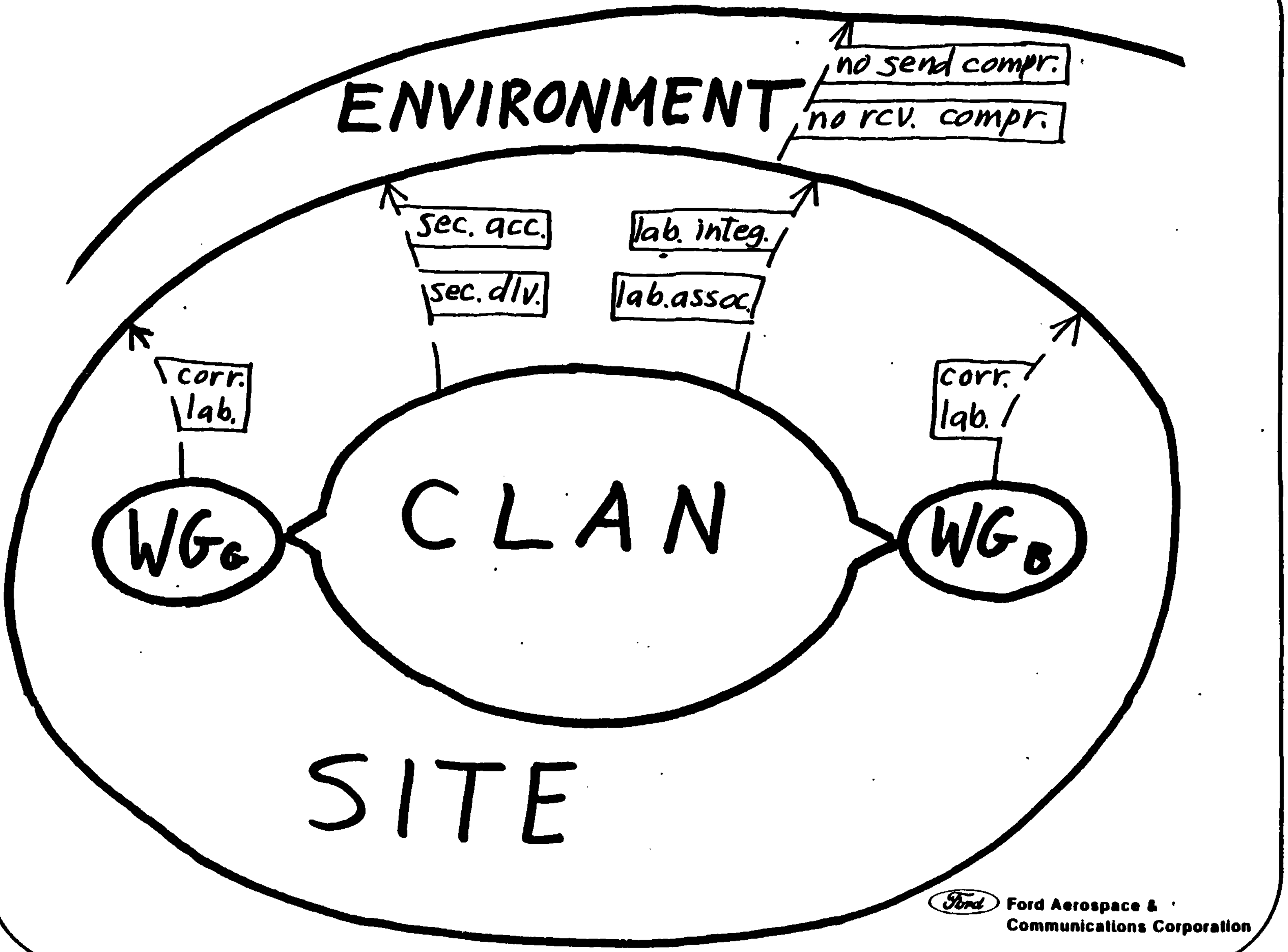
Ford Aerospace & Communications Corporation

NETWORK ARSTRACTION



Ford Aerospace & Communications Corporation

EXAMPLE PROBLEM: SOLUTION VIA CONSTRAINTS



Ford Aerospace & Communications Corporation

CONCLUSION

* Problem: assessment of verification results
* Problem: verifying system-specific characteristics
* Problem: ensuring trustworthiness
* Problem: domain reusability

Ford Aerospace & Communications Corporation

Appendix H

Navy Technology and Ada

LCDR. Philip Myers
NAVALEX

LCDR Philip Myers: Navy Technology and ADA

LCDR Myers spoke on the relationship of verification technology to the Navy's anticipated use of Ada.

There is a need for near term "successes" to sell verification technology to Program Managers for Navy software development projects. Program managers need to be convinced that spending money on verification during the development phase of a system will minimize costs incurred in the maintenance phase due to incorrect software. In addition, verification technology must be useable by a wide class of people, rather than being limited to users with extensive training in logic.

The Navy in particular is adopting the following policies for Navy software systems:

1. The Navy is procuring Ada for its standard set of processors, the AN/UYK-43, AN/UYK-44, and AYK-14. These processors are the ones required for use by all mission critical systems. That is the policy.

2. All Ada software will be developed and maintained on the ALS/N. This is an "envisioned policy" (not promulgated, but anticipated). The ALS/N is being acquired as a Navy Tactical Embedded Resource and would fall under the same policy as the processors.

3. There will be one set of runtime environments for the one set of processors because the ALS/N will have one set of standard compilers for the standard processors.

One implication of the above policies is that if verification technology is to be used in the development of Navy systems, it must be used in the near term, since systems which are developed in the near term will not be replaced for a long time, on the order of 20 to 45 years (the life of the ship and its systems, including software).

Appendix I has been intentionally left blank

Appendix J has been intentionally left blank.

Appendix K

Correctness Proofs of Distributed Termination Algorithms

Krzysztof R. Apt
IBM Yorktown Heights

Krzysztof Apt: Reconsidering Correctness of CSP Programs

First a review of Hoare's communicating sequential processes (CSP) was presented. Then a very simple proof system for a subset of CSP was introduced. This proof system can be used to handle simple Ada programs using a restricted form of tasking.

First CSP programs without nested parallelism are put in normal form. This normal form has one iteration command and all the I/O commands are in the guards. If the respective branches of matching I/O commands re-establish the invariant, then the invariant holds for the iteration command.

In contrast to other CSP proof systems this one does not necessitate an introduction of several auxiliary notions and can be used directly. The proof system has been successfully used to verify a non-trivial example.

In the ensuing discussion the following points were brought out. The transformation to normal form is like changing a program with GOTOs to one with one WHILE loop. A change in size is possible, but many actual programs are nearly in normal form. The global invariant cannot be decomposed into invariants for the individual processes. As a result the method is not modular. However, the individual (sequential) processes can be specified in any high level specification language.

This excerpt consists of sections 1 through 4. The complete paper will appear in:

NATO ASI Series, Vol. F13
Logics and Models of Concurrent Systems
Ed. K. R. Apt.
Springer-Verlag Berlein Heidelberg 1985

CORRECTNESS PROOFS OF DISTRIBUTED TERMINATION ALGORITHMS

Krzysztof R. Apt
L.I.T.P, Université Paris 7
2, Place Jussieu, 75251 Paris, FRANCE

Abstract The problem of correctness of the solutions to the distributed termination problem of Francez [F] is addressed. Correctness criteria are formalized in the customary framework for program correctness. A very simple proof method is proposed and applied to show correctness of a solution to the problem.

1. INTRODUCTION

This paper deals with the distributed termination problem of Francez [F] which has received a great deal of attention in the literature. Several solutions to this problem or its variants have been proposed, however their correctness has been rarely discussed. In fact, it is usually even not explicitly stated what properties such a solution should satisfy.

A notable exception in this matter are papers of Dijkstra, Feijen and Van Gasteren [DFG] and Topor [T] in which solutions to the problem are systematically derived together with their correctness proofs. On the other hand they are presented in a simplistic abstract setting in which for example no distinction can be made between deadlock and termination. Also, as we shall see in the next section, not all desired properties of a solution are addressed there. Systematically derived solutions in the abstract setting of [DFG] are extremely helpful in understanding the final solutions presented in CSP. However, their presentation should not relieve us from providing rigorous correctness proofs of the latter ones - an issue we address in this paper.

Clearly, it would be preferable to derive the solutions in CSP *together* with their correctness proofs, perhaps by transforming accordingly the solutions provided first in the abstract setting. Unfortunately such techniques are not at present available.

This paper is organized as follows. In the next section we define the problem and propose the correctness criteria the solutions to the problem should satisfy. Then in section 3 we formalize these criteria in the usual framework for program correctness and in section 4 we propose a very simple proof method which allows to prove them. In section 5 we provide a simple solution to the problem and in the next section we give a detailed proof of its correctness. Finally, in section section 7 we assess the proposed proof method.

Throughout the paper we assume from the reader knowledge of Communicating Sequential Processes (CSP in short), as defined in Hoare [H], and some experience in the proofs of correctness of very simple loop free sequential programs.

2. DISTRIBUTED TERMINATION PROBLEM

Suppose that a CSP program

$$P \equiv [P_1 \| \cdots \| P_n],$$

where for every $1 \leq i \leq n$ $P_i :: INIT_i ; *[S_i]$ is given. We assume that each S_i is of the form $\underset{j \in \Gamma_i}{\Box} g_{i,j} \rightarrow S_{i,j}$ for a multiset Γ_i and

i) each $g_{i,j}$ contains an i/o command adressing P_j,
ii) none of the statements $INIT_i$, $S_{i,j}$ contains an i/o command.

We say then that P is in a *normal form*. Suppose moreover that with each P_i a *stability condition* B_i, a Boolean expression involving variables of P_i and possibly some auxiliary variables, is associated. By a *global stability condition* we mean a situation in which each process is at the main loop entry with its stability condition B_i true.

We now adopt the following two assumptions :

a) no communication can take place between a pair of processes whose stability conditions hold,

b) whenever deadlock takes place, the global stability condition is reached.

The *distributed termination problem* is the problem of transforming P into another program P' which eventually properly terminates whenever the global stability condition is reached.

This problem, due to Francez [F], has been extensively studied in the literature.

We say that the global stability condition is (not) reached in a computation of P' if it is (not) reached in the natural restriction of the computation to a computation of P. In turn, the global stability condition is reached (not reached) in a computation of P if it holds in a possible (no) global state of the computation. We consider here partially ordered computations in the sense of [L].

We now postulate four properties a solution P' to the distributed termination problem should satisfy (see Apt and Richier [AR]) :

1. Whenever P' properly terminates then the global stability condition is reached.
2. There is no deadlock.

3. If the global stability condition is reached then P' will eventually properly terminate.
4. If the global stability condition is not reached then infinitely often a statement from the original program P will be executed.

The last property excludes the situations in which the transformed parallel program endlessly executes the added control parts dealing with termination detection. We also postulate that the communication graph should not be altered.

In the abstract framework of [DFG] only the first property is proved. Second property is not meaningful as deadlock coincides there with termination. In turn, satisfaction of the third property is argued informally and the fourth one is not mentioned.

Solutions to the distributed termination problem are obtained by arranging some additional communications between the processes P_i. Most of them are programs $P' \equiv [P_1 \| \ldots \| P_n]$ in a normal form where for every i, $1 \le i \le n$

```
$P_i$ :: $INIT_i$ ;...,
       *[ $\Box$ ... ; $g_{i,j} \rightarrow$ ... ; $S_{i,j}$
         $j \in \Gamma_i$
          $\Box$ CONTROL $PART_i$

        ]
```

where ... stand for some added Boolean conditions or statements not containing i/o commands, and CONTROL $PART_i$ stands for a part of the loop dealing with additional communications. We assume that no variable of the original process P_i :: $INIT_i$; $*[S_i]$ can be altered in CONTROL $PART_i$ and that all i/o commands within CONTROL $PART_i$ are of new types.

We now express the introduced four properties for the case of solutions of the above form using the customary terminology dealing with program correctness.

3. FORMALIZATION OF THE CORRECTNESS CRITERIA

Let p,q,I be assertions from an assertion language and let S be a CSP program. We say that $\{p\}$ S $\{q\}$ holds in the sense of *partial correctness* if all properly terminating computations of S starting in a state satisfying p terminate in a state satisfying q. We say that $\{p\}$ S $\{q\}$ holds in the sense of *weak total correctness* if it holds in the sense of partial correctness and moreover no computation of S starting in a state satisfying p fails or diverges. We say that S is *deadlock free relative to* p if in the computations of S starting in a state satisfying p no deadlock can arise. If $p \equiv$ <u>true</u> then we simply say that P is *deadlock free*.

Finally, we say that $\{p\}$ S $\{q\}$ holds in the sense of *total correctness* if it holds in the sense of weak total correctness and moreover S

is deadlock free relative to p. Thus when $\{p\}$ S $\{q\}$ holds in the sense of total correctness then all computations of S starting in a state satisfying p properly terminate.

Also for CSP programs in a normal form we introduce the notion of a global invariant I. We say that I is a *global invariant of* P *relative to* p if in all computations of P starting in a state satisfying p, I holds whenever each process P_i is at the main loop entry. If p ≡ true then we simply say that I is a *global invariant of* P.

Now, property 1 simply means that

$$\{\underline{true}\}\ P'\ \{\bigwedge_{i=1}^{n} B_i\} \qquad (1)$$

holds in the sense of partial correctness.

Property 2 means that P' is deadlock free.

Property 3 cannot be expressed by refering directly to the program P'. Even though it refers to the termination of P' it is not equivalent to its (weak) total correctness because the starting point - the global stability condition - is not the initial one. It is a control point which can be reached in the course of a computation.

However, *in the case of* P' we can still express property 3 by refering to the weak total correctness of a program derived from P'. Consider the following program

CONTROL PART ≡
$[P_1 :: *[\text{CONTROL PART}_1] \| \ldots \| P_n :: *[\text{CONTROL PART}_n]]$.

We now claim that to establish property 3 it is sufficient to prove for an appropriately chosen global invariant I of P'

$$\{I \wedge \bigwedge_{i=1}^{n} B_i\}\ \text{CONTROL PART}\ \{\underline{true}\} \qquad (2)$$

in the sense of total correctness.

Indeed, suppose that in a computation of P' the global stability condition is reached. Then $I \wedge \bigwedge_{i=1}^{n} B_i$ holds where I is a global invariant of P'. By the assumption a) concerning the original program P no statement from P can be executed any more. Thus the part of P' that remains to be executed is equivalent to the program CONTROL PART. Now, on virtue of (2) property 3 holds.

Consider now property 4. As before we can express it only by refering to the program CONTROL PART. Clearly property 4 holds if

$$\{ I \wedge \neg \bigwedge_{i=1}^{n} B_i \} \text{ CONTROL PART } \{\underline{true}\} \qquad (3)$$

holds in the sense of weak total correctness. Indeed, (3) guarantees that in *no computation of P' the control* remains from a certain moment on indefinitely within the added control parts in case the global stability condition is not reached.

Assuming that property 2 is already established, to show property 3 it is sufficient to prove (2) in the sense of weak total corrctness. Now (2) and (3) can be combined into the formula

$$\{I\} \text{ CONTROL PART } \{\underline{true}\} \qquad (4)$$

in the sense of weak total correctness.

The idea of expressing an eventuality property of one program by a termination property of another program also appears in Grumberg et al. [GFMR] in one of the clauses of a rule for fair termination.

4.PROOF METHOD

We now present a simple proof method which will allow us to handle the properties discussed in the previous section. It can be applied to CSP programs being in a normal form. So assume that $P \equiv [P_1 \parallel \ldots \parallel P_n]$ is such a program.

Given a guard $g_{i,j}$ we denote by $b_{i,j}$ the conjunction of its Boolean parts. We say that guards $g_{i,j}$ and $g_{j,i}$ *match* if one contains an input command and the other an output command whose expressions are of the same type. The notation implies that these i/o commands address each other, i.e. they are within the texts of P_i and P_j, respectively and address P_j and P_i, respectively.

Given two matching guards $g_{i,j}$ and $g_{j,i}$ we denote by $Eff(g_{i,j}, g_{j,i})$ the effect of the communication between their i/o commands. It is the assignment whose left hand side is the input variable and the right hand side the output expression.

Finally, let

$$\text{TERMINATED} \equiv \bigwedge_{\substack{1 \le i \le n, \\ j \in \Gamma_i}} \neg b_{i,j}.$$

Observe that TERMINATED holds upon termination of P.

Consider now partial correctness. We propose the following proof rule:

RULE 1 : PARTIAL CORRECTNESS

$$\frac{\begin{array}{l}\{p\}\ INIT_1\ ;\ldots;\ INIT_n\ \{I\},\\ \{I \wedge b_{i,j} \wedge b_{j,i}\}\ Eff(g_{i,j},\ g_{j,i})\ ;\ S_{i,j}\ ;\ S_{j,i}\ \{I\}\\ \text{for all } i,j \text{ s.t. } i \in \Gamma_j,\ j \in \Gamma_i \text{ and } g_{i,j},\ g_{j,i} \text{ match}\end{array}}{\{p\}\ P\ \{I \wedge TERMINATED\}}$$

This rule has to be used in conjunction with the usual proof system for *partial* correctness of nondeterministic programs (see e.g. Apt [A1]) in order to be able to establish its premises. Informally, it can phrased as follows. If I is established upon execution of all the $INIT_i$ sections and is preserved by a joint execution of each pair of branches of the main loops with matching guards then I holds upon exit. If the premises of this rule hold then we can also deduce that I is a global invariant of P relative to p.

Consider now weak total correctness. We adopt the following proof rule:

RULE 2 : WEAK TOTAL CORRECTNESS

$$\frac{\begin{array}{l}\{p\}\ INIT_1\ ;\ldots;\ INIT_n\ \{I \wedge t \geq 0\},\\ \{I \wedge b_{i,j} \wedge b_{j,i} \wedge z{=}t \wedge t \geq 0\}\ Eff(g_{i,j},g_{j,i});S_{i,j};S_{j,i}\{I \wedge 0 \leq t < z\}\\ \text{for all } i,j \text{ s.t. } i \in \Gamma_j,\ j \in \Gamma_i \text{ and } g_{i,j},\ g_{j,i} \text{ match}\end{array}}{\{p\}\ P\ \{I \wedge TERMINATED\}}$$

where z does not appear in P or t and t is an integer valued expression.

This rule has to be used in conjunction with the standard proof system for *total* correctness of nondeterministic programs (see e.g. Apt [A1]) in order to establish its premises. It is a usual modification of the rule concerning partial correctness.

Finally, consider deadlock freedom. Let

$$BLOCKED \equiv \wedge \{\neg b_{i,j} \vee \neg b_{j,i} : 1 \leq i,j \leq n,\ i \in \Gamma_j,\ j \in \Gamma_i,\ g_{i,j} \text{ and } g_{j,i} \text{ match}\}$$

Observe that in a given state of P the formula BLOCKED holds if and only if no communication between the processes is possible. We now propose the following proof rule

RULE 3 : DEADLOCK FREEDOM

$$\frac{\begin{array}{l}I \text{ is a global invariant of } P \text{ relative to } p,\\ I \wedge BLOCKED \rightarrow TERMINATED\end{array}}{P \text{ is deadlock free relative to } p}$$

The above rules will be used in conjunction with a rule of auxiliary variables.

Let A be a set of variables of a program S. A is called the set *of auxiliary variables* of S if

i) all variables from A appear in S only in assignments,
ii) no variable of S from outside of A depends on the variables from A . *In other words* there does not exist an *assignment* $x:=t$ in S such that $x \notin A$ and t contains a variable from A.

Thus for example $\{z\}$ is the only (nonempty) set of auxiliary variables of the program

$[P_1 :: z:=y ; P_2 ! x \parallel P_2 :: P_1 ? u ; u:=u+1]$

We now adopt the following proof rule first introduced by Owicki and Gries in [OG1, OG2].

RULE 4 : AUXILIARY VARIABLES

Let A be a set of auxiliary variables of a program S. Let S' be obtained from S by deleting all assignments to the variables in A. Then

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S'\ \{q\}}$$

provided q has no free variable from A.
Also if S is deadlock free relative to p then so is S'.

We shall use this rule both in the proofs of partial and of (weak) total correctness. Also without mentioning we shall use in proofs the well-known consequence rule which allows to strengthen the preconditions and weaken postconditions of a program.

Acknowledgements We would like to thank to L. Bougé, C. Delporte-Gallet, N. Francez and A. Pnueli for interesting and helpful discussions on the subject of this paper. Also we are grateful to Mrs A. Dupont for her speedy and efficient typing of the manuscript.

REFERENCES

[A1] APT, K.R., Ten years of Hoare's logic, a survey, part II, Theoretical Computer Science 28, pp. 83-109, 1984.

[A2] APT, K.R., Proving correctness of CSP programs, a tutorial, Tech. Report 84-24, LITP, Université Paris 7, 1984 (also to appear in the Proc. International Summer School "Control Flow and Data Flow : Concepts of Distributed Programming", Marktobedorf, 1984).

[AC] APT, K.R. and CLERMONT Ph., Two normal form theorems for CSP programs, in preparation.

[AFR] APT, K.R., FRANCEZ N. and DE ROEVER, W.P., A proof system for Communicating Sequential Processes, ACM TOPLAS 2 No 3, pp. 359-385, 1980.

[AR] APT, K.R. and RICHIER, J.L., Real time clocks versus virtual clocks, Tech. Report 84-34, LITP, Université Paris 7, 1984, (also to appear in the Proc. International Summer School "Control Flow and Data Flow : Concepts of Distributed Programming", Marktobedorf, 1984).

[DFG] DIJKSTRA, E.W., FEIJEN, W.H. and van GASTEREN, A.J.M., Derivation of a termination detection algorithm for distributed computations, Inform. Processing Letters 16, 5, pp. 217-219, 1983.

[EF] ELRAD, T.E. and FRANCEZ, N., Decomposition of distributed programs into communication closed layers, Science of Computer Programming 2, No 3, pp. 155-174, 1982.

[F] FRANCEZ, N., Distributed termination, ACM TOPLAS 2, No 1, pp. 42-55, 1980.

[FRS] FRANCEZ , N., RODEH, M. and SINTZOFF, M., Distributed termination with interval assertions, in : Proc. Int Colloq. Formalization of Programming Concepts, Peniscola, Spain, Lecture Notes in Comp. Science, vol. 107, 1981.

[GFMR] GRUMBERG, O., FRANCEZ N., MAKOWSKY J., and DE ROEVER W.P., A proof rule for fair termination of guarded commands, in : J.W. de Bakker and J.C. Van Vliet eds., Algorithmic languages, IFIP, North Holland, Amsterdam, pp. 399-416, 1981.

[H] HOARE, C.A.R., Communicating sequential processes, CACM 21, 8, pp. 666-677, 1978.

[L] LAMPORT, L., Time, clocks and the ordering of events in a distributed system, CACM 21, 7, pp. 558-565, 1978.

[MP] MANNA, Z. and PNUELI, A., How to cook a temporal proof system for your pet language, in : Proc. of the Symposium on Principles of Programming Languages, Austin, Texas, 1983.

[T] TOPOR, R.W., Termination detection for distributed computations, Inform. Processing Letters 18, 1, pp. 33-36, 1984.

Frank Oles: Thoughts on an ADA-based Design Language

Design languages created by those interested in prototyping turn out too awkward to use for formal verification, and design languages created by those interested in formal verification are difficult to use for rapid prototyping. It was argued that the proper course is to regard a design language as a coherent framework for relating three distinct sublanguages: an implementation language, a specification language, and a prototyping language. The dangers of regarding all specifications as executable were analyzed. Desireable features of the prototyping language and the specification language were enumerated.

Discussion

In the discussion the advantages of different specification languages for different purposes was emphasized. The speaker said that set theory was chosen as the basis for prototyping because it was familiar. Furthermore, untyped set theory as a basis for a typed language presented no special difficulties.

Thoughts on ADA-Based Design Languages

Frank J. Oles

Exploratory Computer Science
IBM T. J. Watson Research Center
Yorktown Heights, NY

IBM

Those interested in FORMAL VERIFICATION create one kind of design language.

Those interested in RAPID PROTOTYPING create another.

Is
COHERENT
RECONCILIATION
possible?

Desirable Components in a Design Language

- An Implementation Language
  - -- An Imperative Programming Language
  - -- Store-Oriented Semantics
  - -- ADA
- A Specification Language
  - -- More General Semantics
  - -- Nonexecutable
  - -- ANNA annotations
- A Prototyping Language
  - -- An Imperative Programming Language
  - -- Extension of the Implementation Language
  - -- Similar to ADA Virtual ADA in ANNA

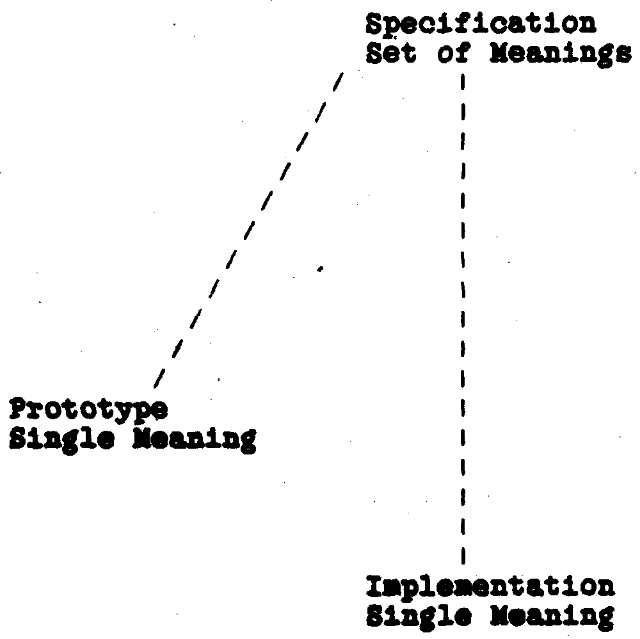
03/85 (Frank J. Oles)

Appendix L

Thoughts on an Ada-Based Design Language

Frank Oles
IBM Yorktown Heights

RELATIONSHIPS AMONG COMPONENTS



Varieties of Specifications

A specification is

- <u>strong</u> if its set of meanings consists of a single element,
- <u>weak</u> if its set of meanings consists of more than one element.

It is unnatural and misleading to define weak specs in imperative terms.
(Sometimes operational nondeterminism is confused with weak specification.)

The strong specifications are the possibly executable ones.

A prototyping language should make it possible to develop strong specs easily.

03/85 (Frank J. Oles)

Executable Specifications

Should the two high-level components be combined?

1. A Specification Language
2. A Prototyping Language

Whether it can be done without making all specs for subprograms executable is doubtful.

If only executable specs are supported, then the software designs will suffer from overspecification.

03/85 (Frank J. Oles)

Prototyping Language Features

- Should be a high-level extension of ADA
- Type system of ADA
  - ☆ is focused on compile-time typing.
  - ☆ makes it hard to visualize sets of values for some types, independent of storage considerations.
- Needed: a more refined type structure
  - ☆ Value types
  - ☆ Phrase types

03/85 (Frank J. Oles)

Value Types

If A and B are value types, then more value types are SET OF A, SEQ OF A, MAP(A,B), REL(A,B).

DISJOINTUNION would be an important value type constructor.

Introduce value types as solutions of recursive equations.

Any type with a definition involving arrays or access types is NOT a value type.

Imperative constructs needed

- FIND
- SELECTION

03/85 (Frank J. Oles)

Specification Language Features

- General enough to apply to both ADA and the Prototyping Language.
- Supports precondition/postcondition method of specifying subprograms.
- Supports direct specification of implementation by prototype, when desirable.
- Distinguishes between the specification of user-defined value types and state-machines.
- Supports specification of types BOTH by abstract models and by axioms.
- Has multiple levels of refinement BOTH for subprograms and for packages.
- Formally incorporates examples of use in specification of functions.

03/85 (Frank J. Oles)

Evils of Overspecification

Overspecification: the inclusion in a software design of conceptually unnecessary or undesirable constraints.

Consequences of overspecification:

- Unnecessary early design errors.
- Irrelevant detail impairs understanding.
- Harder to maintain correct relationship between spec and implementation.
- Limits reusability.

03/85 (Frank J. Oles)

Appendix M

Axiomatic Semantics for Ada

Norman Cohen
SofTech

Norman Cohen: Axiomatic Semantics for ADA

The goal is to prove selected properites about ADA programs. These properties include the implementation of algebraically defined data abstractions, numeric properties, the absence of unanticipated exceptions, and the absence of erroneous execution.

Erroneousness arises because some problems are too expensive or too complicated to be detected by the compiler. The compiler assumes that certain rules are obeyed, and if they are not, then the effect of further execution is unpredictable. It is valuable to prove the absence of erroneousness. Restricting the language can make this job easier.

A verifiable subset will forbid aliasing, changing shared variables in tasks, unchecked conversion, and address clauses.

An ADA verifier can make use of the fact that the compiler is a verifier for the static semantics of the languages. A DIANA tree can be input to both the code generation part of the compiler and the verifier for ADA dynamic semantics. Still an UNambigous, Resolved, Expanded ADA Depicting All Bindings in the Lexical Environment (UNREADABLE) is necessary for verification. Distinct variables and subprograms are given distinct names, so proof rules based on textual substitution are valid.

Proposed solutions to problems with implementation-dependent behavior, optimization, and exceptions were also presented.

Discussion

In the discussion the speaker pointed out that an important goal of ADA is portabiltity. One may place limits on compilers (like how they optimize code), but one should not tie verification to a particular compiler. Also pointed out, was the different approaches to dealing with erroneousness. In some cases it is ruthlessly forbidden. In other cases, like aliasing, there is a responsibility to prove that there is no aliasing.

GOAL: PROVE SELECTED PROPERTIES OF SELECTED COMPONENTS

- CORRECT IMPLEMENTATION OF ALGEBRAICALLY DEFINED DATA ABSTRACTIONS
- NUMERIC PROPERTIES
- ABSENCE OF UNANTICIPATED EXCEPTIONS
- ABSENCE OF ERRONEOUS EXECUTION

SOFTECH

vg1375wo/1 3/18/85

THE ROLE OF ERRONEOUSNESS

- COMPILERS CAN PASS THE BUCK TO THE PROGRAMMER:
  - CERTAIN RULE VIOLATIONS TOO EXPENSIVE OR COMPLICATED TO DETECT AT EITHER COMPILE TIME OR RUN TIME
  - COMPILER MAY <u>ASSUME</u> THE RULES ARE OBEYED
  - IF THE RULES ARE VIOLATED, EFFECT OF FURTHER EXECUTION IS UNPREDICTABLE
- FOR VERIFIERS, THE BUCK STOPS HERE:
  - IF A PROOF RULE ASSUMES THE ABSENCE OF ERRONEOUSNESS, THE ABSENCE OF ERRONEOUSNESS MUST BE PROVEN BEFORE APPLYING THE RULE
  - ABSENCE OF ERRONEOUSNESS IS A VALUABLE PROPERTY TO PROVE
  - A VERIFIABLE SUBSET CAN SIMPLIFY PROOFS OF NONERRONEOUSNESS

vg1375wo/2 3/18/85

SOFTECH

VERIFIABLE SUBSET

- NO ALIASING:
  - A GLOBAL REFERENCE TO A(I) WITHIN A SUBPROGRAM BODY MEANS
    - A(I) IS AN IMPLICIT PARAMETER IF I IS STATIC
    - ALL OF A IS AN IMPLICIT PARAMETER IF I IS NOT STATIC
  - IF A(I) AND A(J) ARE IMPLICIT OR EXPLICIT ACTUAL PARAMETERS OF MODE <u>OUT</u>, I/=J IS A PRECONDITION TO THE CALL
- NO REFERENCES WITHIN A TASK BODY TO VARIABLES OR COLLECTIONS THAT CAN BE CHANGED FROM OUTSIDE THE TASK BODY
- NO UNCHECKED CONVERSION
- NO ADDRESS CLAUSES

vg1375wo/3 3/18/85

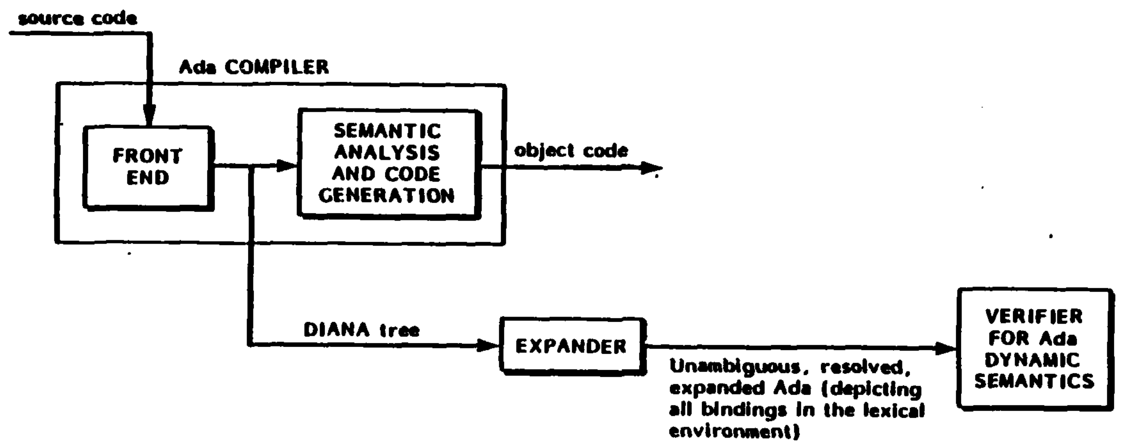
SOFTECH

STATIC VERSUS DYNAMIC SEMANTICS

	ALGOL 60	PL/I	Ada
STATIC	22%	26%	55%
DYNAMIC	78%	74%	45%

SOURCE: BJØRNER AND OEST, TOWARDS A FORMAL DESCRIPTION OF Ada, LNCS VOLUME 98

- COMPILERS ARE:
  - VERIFIERS FOR STATIC SEMANTICS
  - INTERPRETERS OF STATIC INFORMATION (e.g. OVERLOAD RESOLUTION)


source code
Ada COMPILER
FRONT END
SEMANTIC ANALYSIS AND CODE GENERATION
object code
DIANA tree
EXPANDER
Unambiguous, resolved, expanded Ada (depicting all bindings in the lexical environment)
VERIFIER FOR Ada DYNAMIC SEMANTICS

vg1375wo/4 3/18/85

SOFTECH

UNambiguous Resolved, Expanded Ada, Depicting All Bindings in the Lexical Environment (UNREADABLE)

- EACH NAME IS A FULLY EXPANDED NAME BEGINNING WITH "STANDARD."
- SUBPROGRAM NAMES END WITH PARAMETER/RESULT TYPE PROFILES
- EXAMPLE: A-B MIGHT BECOME

  Standard."-"[Standard.Integer, Standard.Integer → Standard.Integer]
  (Standard.Main.A, Standard.Main.B)

- DISTINCT VARIABLES AND SUBPROGRAMS HAVE DISTINCT NAMES, SO PROOF RULES BASED ON TEXTUAL SUBSTITUTION ARE VALID

SOFTECH

vg1375wo/5 3/18/85

ELIMINATING RENAMING DECLARATIONS FROM "UNREADABLE"

- Ada :

```
task Sensor_Task is
   entry Reset_Sensor (1 .. 5) (X, Y : in Float = 0.0);
   ...
end Sensor_Task;

procedure Reset_Main_Sensor
   (Main_X : in Float; Main_Y : in Float := Current_Y)
   renames Sensor_Task.Rest_Sensor (Current_Sensor);

...

Reset_Main_Sensor (Main_X => 1.25);
```

- UNREADABLE:

```
$index1 : constant Integer range 1 .. 5 := Main_Sensor;
...
Sensor_Task.Reset_Sensor ($index1)(X => 1.25; Y => Current_Y);
```

- IN RENAMING OBJECTS, ACCESS VALUES AND ARRAY INDICES ARE HANDLED LIKE THE ENTRY INDEX ABOVE

- RENAMINGS FOR EXCEPTIONS AND PACKAGES ARE SIMPLE STATIC STRING SUBSTITUTION

SOFTECH

vg1375wo/6 3/18/85

SOME OTHER ASPECTS OF "UNREADABLE"

- FUNCTION CALLS AND ALLOCATORS GIVEN TWO PARAMETERS TO ENSURE THAT TEXT OF THE CALL OR ALLOCATOR CAN DENOTE THE VALUE IT RETURNS
  - $action_counter (IMPLICITY INCREMENTED AFTER EACH STATEMENT AND AT CERTAIN OTHER TIMES)
  - STATIC TAG (DISTINGUISHING CALLS AND ALLOCATORS WITHIN A STATEMENT)
- EXPRESSIONS WRITTEN AS MANY TIMES AS THEY ARE EVALUATED
- EXAMPLES:

```
Node:=Build_Tree(new Integer, new Integer);

Node:=
   Build_Tree
      (new ($action_counter,1)Integer,
       new ($action_counter,2)Integer

A,B:Calendar.Time:=Calendar.Clock;

A:Calendar.Time:=Calendar.Clock($action_counter,1);
B:Calendar.Time:=Calendar.Clock($action_counter,1);
```

- DEFAULTED PARAMETERS EXPLICITY SUPPLIED
- UNIFORM, EXPLICIT NOTATION:

```
A.all(3)        (1=>x, 2=>x, 3=>x)
```

SOFTECH

INCORRECT ORDER DEPENDENCE

- TYPICAL PROOF RULE IF ACTION A CONSISTS OF SUB-ACTIONS S1, S2, and S3, PERFORMED "IN SOME ORDER THAT IS NOT DEFINED BY THE LANGUAGE":

$$\frac{\begin{array}{c} P\{S1;S2;S3;\}Q \\ P\{S1;S3;S2;\}Q \\ P\{S2;S1;S3;\}Q \\ P\{S2;S3;S1;\}Q \\ P\{S3;S1;S2;\}Q \\ P\{S3;S2;S1;\}Q \end{array}}{P\{A\}Q}$$

- COMBINATORIAL EXPLOSION CAN BE AVOIDED WHEN SUB-ACTIONS (TYPICALLY EVALUATION OF EXPRESSIONS) HAVE NO SIDE EFFECTS

IMPLEMENTATION-DEPENDENT, UNSPECIFIED OR INDETERMINATE BEHAVIOR

- EXAMPLE: PROVE

```
{Standby_Cell/=null}
begin
     A := new Integer'(0);
exception
     when Storage_Error=>
          Standby_Cell.all := 0;
          A := Standby_Cell;
end;
{A.all=0}
```

- THERE IS NO PORTABLE WAY TO CHARACTERIZE STATES IN WHICH ALLOCATION WILL NOT RAISE STORAGE_ERROR
- FIRST APPROACH
  - HAVE NO PROOF RULE JUSTIFYING THE CONCLUSION THAT STORAGE_ERROR IS RAISED OR THAT IT IS NOT
  - REJECTED BECAUSE THIS APPROACH IS TOO WEAK FOR THE PROOF ABOVE

SOFTECH

IMPLEMENTATION-DEPENDENT, UNSPECIFIED OR INDETERMINATE BEHAVIOR: SOLUTION

- INTRODUCE AN UNINTERPRETED LOGICAL PREDICATE, SAY P
- INTUITIVELY, TRUTH OF P IMPLIES ALLOCATION WILL SUCCEED
- PROOFS ASSUME NOTHING ABOUT THE VALUE OF P, BUT THE USUAL RULES OF PREDICATE CALCULUS APPLY
- BLOCK PRECONDITION:
  (P and True) or (not P and Standby_Cell/= null)
- THIS IS IMPLIED BY Standby_Cell/= null)

SOFTECH

OPTIMIZATION: REASSOCIATING OPERATORS

"...FOR A SEQUENCE OF PREDEFINED OPERATORS AT THE SAME PRECEDENCE LEVEL (AND IN THE ABSENCE OF PARENTHESES IMPOSING A SPECIFIC ASSOCIATION), ANY ASSOCIATION OF OPERATORS WITH OPERANDS IS ALLOWED IF [IT IS MATHEMATICALLY EQUIVALENT]."
- RM 11.6(5)

- CONSERVATIVE INTERPRETATION:
  AFTER OVERLOADING HAS BEEN RESOLVED AND OPERAND TYPES HAVE BEEN DETERMINED, THE ASSOCIATIVE LAW MAY BE USED TO REASSOCIATE SUCCESSIVE APPLICATIONS OF AND, OR, XOR, +, &, AND *, PROVIDED THAT OPERAND TYPES ARE UNAFFECTED

  - EXAMPLE:

```
I1, I2:Integer;
F     :Some_Fixed_Point_Type;

I1*F*I2    => I1*(FI2)
F*I1*I2    ≠> F*(I1*I2)
```

SOFTECH

REASSOCIATING OPERATORS: SOLUTION

- IN UNREADABLE, A+B+C+D REPRESENTED AS

  SUM(<VERSION OF "+">, A,B,C,D)
- NOTHING CAN BE PROVEN ABOUT EVALUATION OF THE EXPRESSION UNLESS IT CAN BE PROVEN FOR ALL POSSIBLE ASSOCIATIONS
- WILL COMBINATORIAL EXPLOSION BE A PROBLEM IN PRACTICE?

SOFTECH

OPTIMIZATION: CODE MOTION

- EXAMPLE FROM RM 11.6(10):

```
--A and K are global variables.
declare
  N: Integer;
begin
--Evaluation of A(K) may be moved here
  N := 0;
  for J in 1..10 loop
     N := N + J**A(K);
  end loop;
  Put (N);
exception
  when others =>Put ("some error arose"); Put (N);
end;
```

- SOLUTION:

  IF SOME CONDITION IS TO BE ASSUMED UPON ENTRY TO A HANDLER FOR A PREDEFINED EXCEPTION, IT MUST HOLD AT EACH "INTERMEDIATE STEP" WITHIN THE SEQUENCE OF STATEMENTS PRECEDING A POINT WHERE THE EXCEPTION MIGHT NORMALLY BE RAISED

- INTERPRETATION:

  AN "INTERMEDIATE STEP" OCCURS BETWEEN STATEMENTS AND BETWEEN SUBEXPRESSION EVALUATIONS WITHIN A STATEMENT

SOFTECH

RAISE STATEMENTS WITHOUT EXCEPTION NAMES

- A STATEMENT OF THE FORM

  ```
  raise;
  ```

  IS ONLY ALLOWED INSIDE A HANDLER, WHERE IT RERAISES THE EXCEPTION THAT BROUGHT CONTROL TO THE HANDLER

- THIS EXCEPTION CANNOT ALWAYS BE DETERMINED STATICALLY:

  ```
  when Constraint_Error | Numeric_Error =>
  when others =>
  ```

- HANDLERS MAY BE NESTED

```
when A =>
   begin
      ...
   exception
      when B =>
            <<Label>>
            ...
         begin
            ...
         exception
            when C =>
               begin
                  ...
               exception
                  when D => goto Label;
                  when E => raise;
               end;
               raise;
         end;
         raise;
   end;
   raise;
```

SOFTECH

SOLUTION: "EXCEPTION CONTEXTS"

- AN ORDERED LIST OF CURRENTLY ACTIVE EXCEPTIONS, WITH THE "INNERMOST" ACTIVE EXCEPTION OCCURRING FIRST
- IF AN EXCEPTION WITH A HANDLER IS RAISED, THE EXCEPTION IS APPENDED TO THE FRONT OF THE LIST BEFORE THE HANDLER IS ENTERED
- A RAISE STATEMENT WITHOUT AN EXCEPTION NAME RAISES THE EXCEPTION AT THE FRONT OF THE LIST
- UPON DEPARTURE FROM A HANDLER, THE FRONT EXCEPTION IS REMOVED FROM THE LIST
- A goto FROM A HANDLER $\underline{n}$ LEVELS DEEP TO AN OUTER HANDLER $\underline{n} - \underline{k}$ LEVELS DEEP REMOVES THE FRONT $\underline{k}$ ITEMS FROM THE LIST

REMARKS: EVERY PROOF RULE FOR COMPOUND STATEMENTS (AS WELL AS "raise;" AND "goto L;") IS IN TERMS OF AN EXCEPTION CONTEXT.

IN PRACTICE, HANDLERS ARE ALMOST NEVER NESTED

SOFTECH

vg1375wo/15 3/18/85

SUMMARY

- THE POSSIBILITY OF ERRONEOUSNESS MUST BE CONFRONTED, NOT IGNORED
- A VERIFIABLE SUBSET CAN BE DEFINED CONSISTING OF BOTH STATIC AND DYNAMIC RULES AND INCLUDING MOST OF Ada
- SEMANTICS CAN BE SIMPLIFIED BY LEAVING STATIC ISSUES TO A COMPILER AND PREPROCESSOR
- PROBLEMS ARISING FROM DEFINITION OF Ada:
  - INCORRECT ORDER DEPENDENCIES
  - INDETERMINATE BEHAVIOR
  - ALLOWABLE OPTIMIZATIONS (ASSOCIATIVE LAW, CODE MOTION)
  - EXCEPTION CONTEXTS

SOFTECH

[illegible]

Appendix N

Teaching Programmers About Proofs

David Gries
Cornell University

David Gries: Teaching Programmers about Proofs

Programmers must be given simple rules. Thus an important goal is the **simplicity of** proof rules. The following simple proof rule for procedure calls, due to Alain Martin, **Acta Informatica** 1984 was described and justified:

Let p(**in** x; **inout** y; **out** z) be a procedure with body S, such that the following Hoare triple is valid:

```
{P} S {Q}
```

where we assume that in P, only x and y are free; in Q only x, y, and z are free, and that S doesn't change x. We are interested in finding a suitable precondition for:

```
? p(a,b,c) {R}
```

Execution of the call P, using call by value, is equivalent to

```
x,y:= a,b;    -- this is ordered simultaneous assignment, so that,
              -- e.g. b,b:= 6,7 is admissible

S;

b,c:= y,z;
```

Let A be a predicate containing only x as free program variable. Then the following rule is valid:

```
Q ^ A => R[b,c<-y,z]
_____________________________

{P[x,y<-a,b] ^ A[x<-a]} p(a,b,c) {R}
```

Simple examples of the use of the rule were given.

A plea was made for replacing the formal parameter/actual parameter terminology by parameter/argument.

The same rule works for call-by-reference, if aliasing is prohibit‿ed.

The simple rule for assignments

```
{R[x<-e]} x:= e; {R}
```

has the instance

$\{y = 0\}\ x := e;\ \{y = 0\}$,

and thus assumes that there are no side effects. There are rules, but more complicated ones, for assignments permitting side effects.

Discussion

The speaker maintained that good programmers do actually use these sorts of rules, at least, unconsciously. The evidence is mostly anecdotal, in the speaker's experience no one has shown a loop correct, except by a variant of these methods. Teaching these rules to all programmers promotes better awareness, more care for details, and faster programming. Real programs, big programs are composed of small programs. Putting small programs together is still an unsolved problem.

Alain J. Martin

General Proof Rule

for

Procedure Call

Acta Informatica 84

```
proc p ( in    x;
         inout y;
         out   z );

     { P }              S              { Q }
       |                |                |
       |                |                |
     only           does not           only
     x, y            change          x, y, z
     free              x               free

                    p (a, b, c)  { R }
```

```
x, y := a, b;

S;

b, c := y, z;
```

$$\begin{array}{lcl} \{\ P^{x,y}_{a,b} & \wedge & A^{x,y}_{a,b}\ \} \\ \{\ P\ \} & & \{\ A\ \} \\ & & \{\ A\ \} \\ -\ \{\ Q\ \} & \longleftrightarrow & \{\ R^{b,c}_{y,z}\ \} \\ \{\ R\ \} & & \end{array}$$

$$Q \wedge A \Rightarrow R^{b,c}_{y,z}$$

only free vars: x

$$Q \wedge A \rightarrow R^{b,\ c}_{y,\ z}$$

$$\{\ P^{x,\ y}_{a,\ b} \wedge A^{x}_{a}\ \} \quad p\ (a,\ b,\ c)\ \{\ R\ \}$$

(A contains as free program variables only x)

A is the <u>adaptation</u>.

proc inc (in x; out y)

$\{ \; P : \text{true} \; \}$ $\qquad$ $\{ \; Q : y = x + 1 \; \}$

$\{ \; a + 1 = 3 * a \; \}$

$\{ \; (x + 1 = 3 * a)^{x}_{a} \; \}$

inc (a, c)

$\{ \; R : c = 3 * a \; \}$

$Q \wedge A \Rightarrow R^{c}_{y}$

$y = x + 1 \wedge A \Rightarrow y = 3 * a$

$\qquad\qquad\qquad\qquad x + 1 = 3 * a$

$Q \wedge A \Rightarrow R^{c}_{y}$

```
y = x + 1     ^      A      =>     y = co + 1
                    / \
                   /   \
             A : x = co
```

```
proc  inc  ( in  x;  out  y )

    {   P : true   }     y := x + 1    {   Q : y = x + 1   }

                 y = x + 1    ∧              -> c = co + 1

    {   true   ∧   $( x = co )^{x}_{c}$   }

    inc ( c, c )

    {   R : c = co + 1   }
```

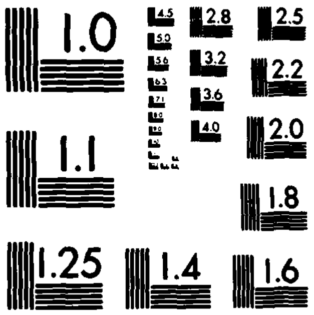
Appendix O has been intentionally left blank.

Appendix P

Discussion of Papers from the Session on Advanced Verification

AD-A171 507 PROCEEDINGS OF THE IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION O..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C G ROBY DEC 85 IDA-M-146 4/5

UNCLASSIFIED IDA/HQ-85-30658 MDA903-84-C-0031 F/G 9/2 NL



!CROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Discussion of papers on Advanced Verification

Oles was asked, why should one not want specifications to be executable? He said that it is mainly a sociological problem. If you tell programmers specifications are executable, they will give up on abstraction. For example, the specification of security is a limited property that has no functionality. An executable specification requires saying much more.

Gries was asked to address the real-world problem of programming well, even if not correctly. His response was that there are no other methods so crystallized as to be teachable.

Apt was asked about the prospects of going beyond CSP to prove correctness of ADA tasks. Apt suggested that clustering might be a possibility. Since nested accepts are very awkward in CSP looking for a normal form for ADA programs might be profitable. On the other hand, Apt has pessimistic about dynamic task creation. In response to another qeustion Apt said that procedures and packages can be used if they are inside the individual processes.

McHugh offered his experience with verified software in practice. First, he noted that some software verified by Gypsy was not being used for extraneous reasons. On the other hand, an informally hand-verified program (1000-1500 lines of assembly code) has been used successfully for a long period of time. The IBM "cleanroom" method -- hand proofs about control structures (in place of unit testing) -- has been successfully used. IBM has trained 2000 such people and will teach this to anyone for a fee. It is now being taught to freshmen at the University of Maryland.

Appendix Q

Practical Verification Systems

Jim Wiliams
MITRE

Jim Williams: Practical Verification Systems

The practical verification system has three goals. (1) Soundness: a rigorous logical basis with a single, expressive logic. (2) Performance: minimize user-supplied, application-dependent proofs. (3) User-friendliness: language independent, usable by the non-logician.

The major components of a verification system: the underlying logic, theorem proving, and the user interface, should be pursued in parallel and properly factored.

The practical verification system uses a transformational approach. The specifications are transformed to a functional description, then to a detailed program description, and finally to the source code.

Already there is a PROLOG prototype. The internal logic is in the final debugging stages. The theorem prover is based on the Argonne logic machine architecture and is related to parallel hardware efforts. The user interface is designed to support a variety of languages and has been ported to SUN workstations.

PRACTICAL VERIFICATION SYSTEM

MARCH 1985

PVS REQUIREMENTS

SOUNDNESS

PERFORMANCE

USER-FRIENDLINESS

* single internal language

* minimize user-supplied, application-dependent proofs

* user-interface software
* track user's understanding

* very expressive logic

* high-level reusable theorems
  knowledge-based theorem proving
  transformational paradigm

user's domain of discourse
(language independence)

A FURTHER DERIVATION

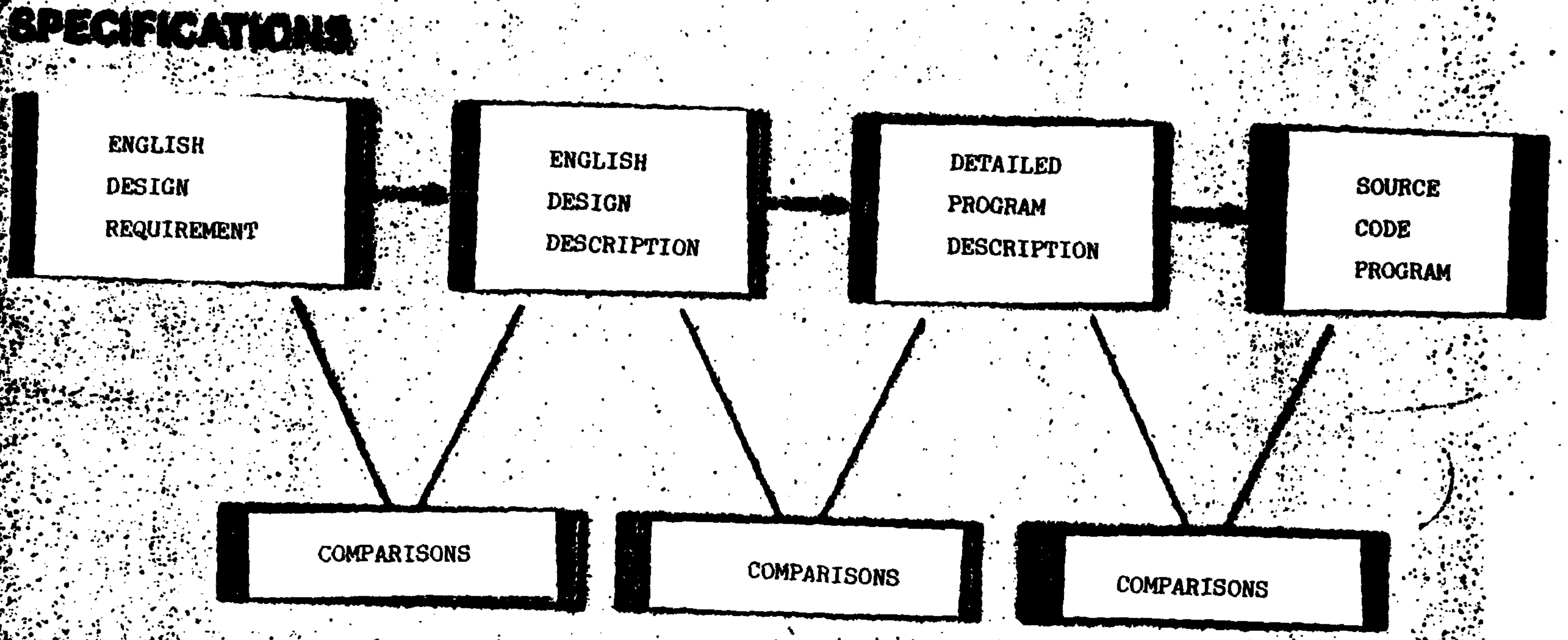
Components of verification technology, namely:

* symbolic logic and applications
* automatic theorem proving
* user-interface development

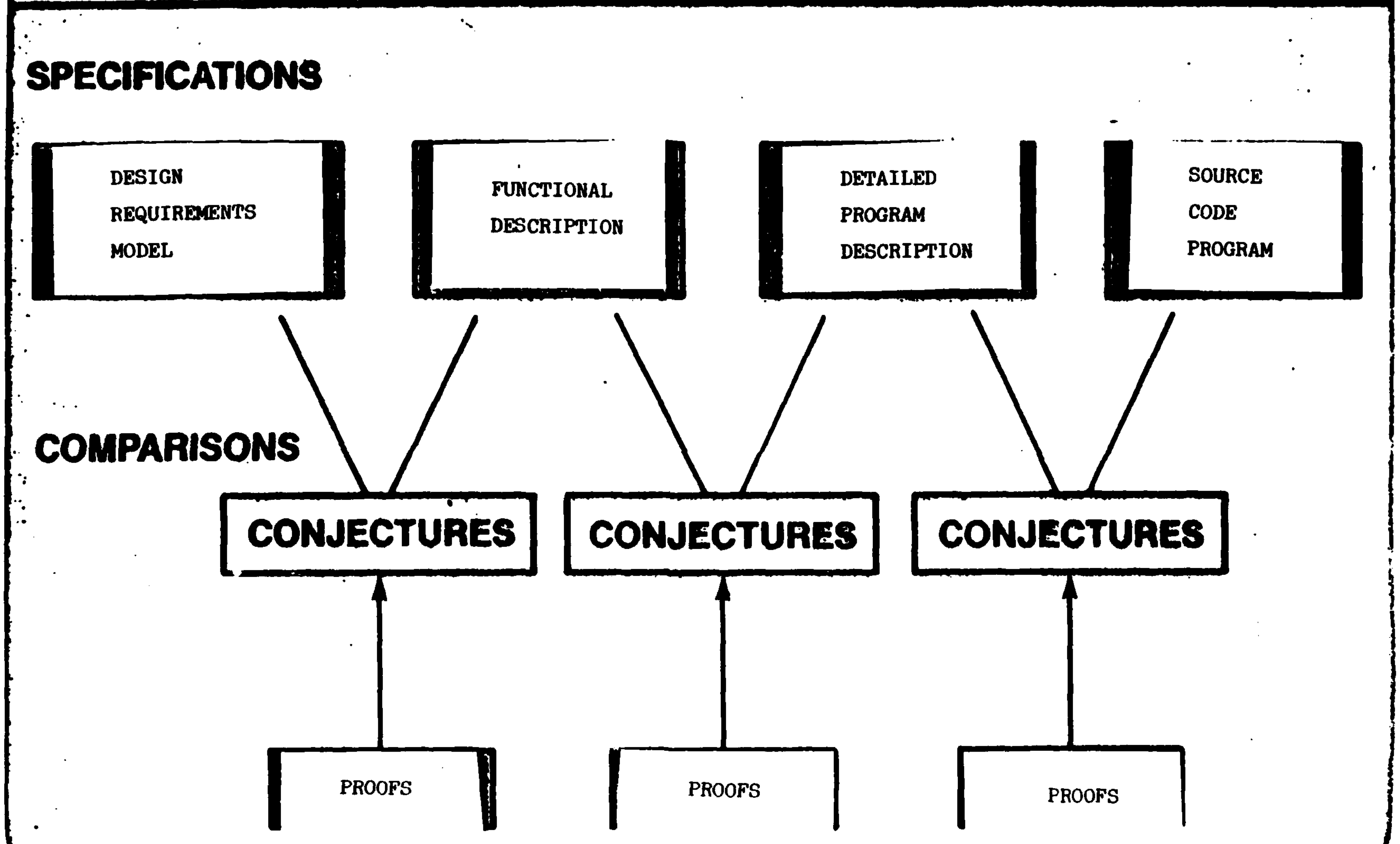
Should be:

* pursued in parallel
* properly factored

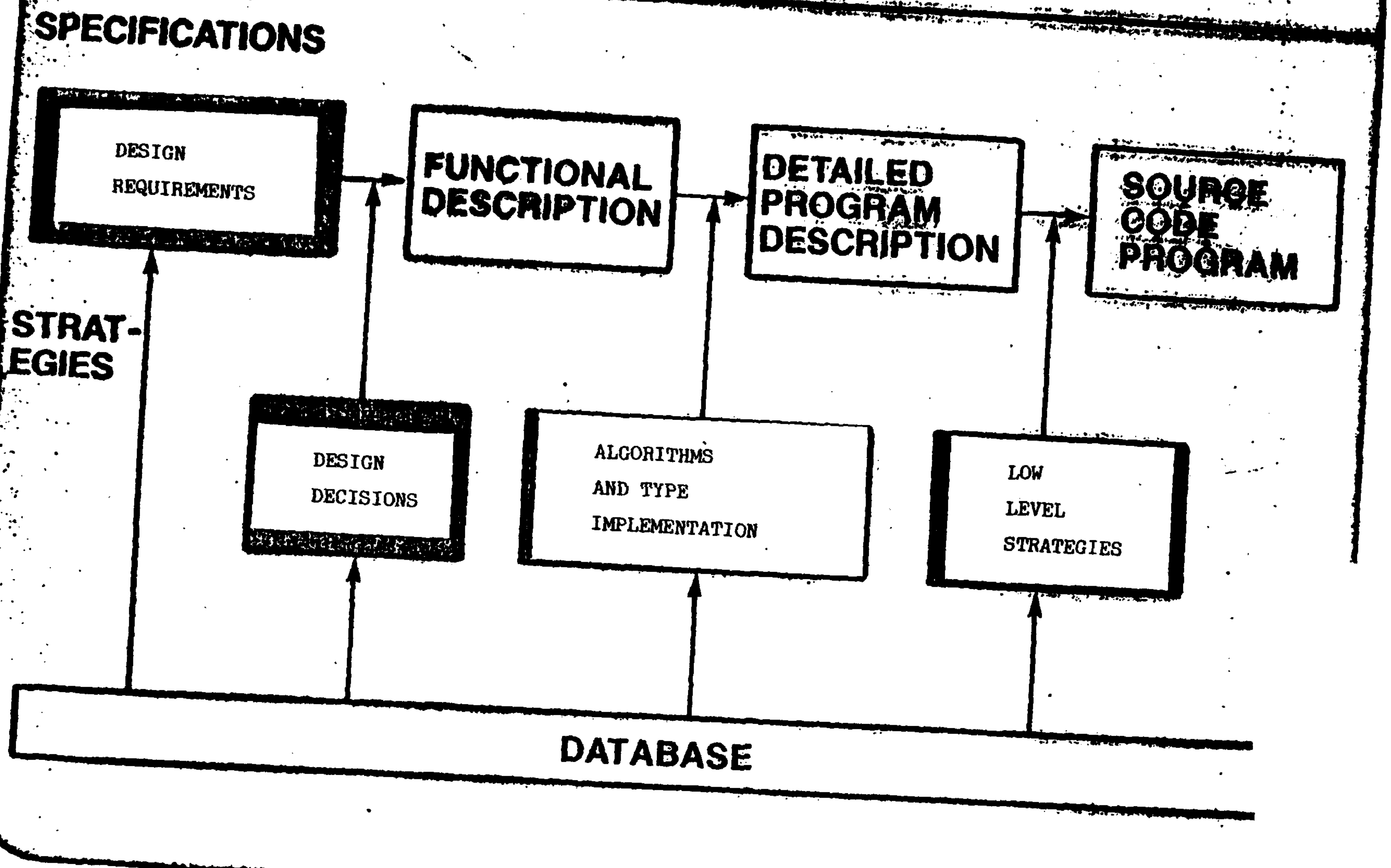
Traditional Verification and Validation


SPECIFICATIONS
ENGLISH DESIGN REQUIREMENT
ENGLISH DESIGN DESCRIPTION
DETAILED PROGRAM DESCRIPTION
SOURCE CODE PROGRAM
COMPARISONS
COMPARISONS
COMPARISONS

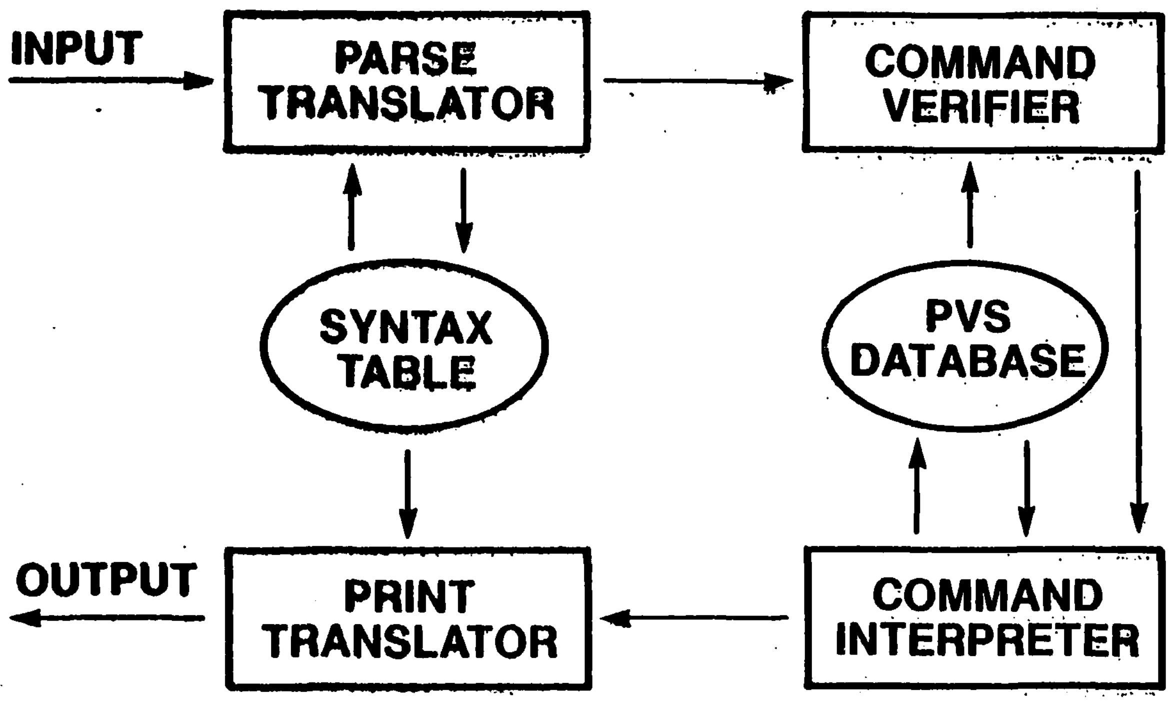
Traditional Mathematical Verification



Transformational Verification Process



Overall Organization Of The PVS System



RECENT ACCOMPLISHMENTS

* PROLOG PROTOTYPE & DOCUMENTATION

* INTERNAL LOGIC
  - final debugging stages
  - intensive 7 person effort

* AUTOMATIC THEOREM PROVING
  - Argonne Logic Machine Architecture
  - 3 person design team
  - related to parallel hardware efforts

RECENT ACCOMPLISHMENTS (Cont'd)

* USER INTERFACE SOFTWARE

  - a variety of languages

  - ported to Sun workstations

Publications

ACM 84, "PVS-Design for a PVS"

IEEE TOSE, "Defining Database Views as Data Abstractions"

APPLICATION TO ADA

* Challenge and Opportunity

* Approach

    -Ada Formal Semantics

    -Specs and Strategies

* Benefits

Challenge: Concurrency

-in general

-in Ada

Opportunity: Generic Packages

-reduction of application-
dependent theorem proving

ADA FORMAL SEMANTICS

* Explicit Mathematical Model

* Prudent Restrictions (ala ORA)

* Simplify Definitions via
  Powerful Logic

* Suppress Recoverable detail via
  User-Interface Translation

SPECIFICATIONS & STRATEGIES

* Sequences
* Interactive Sequential Processes
* Pipes, Buffers, Sockets
* Suitable Implementation Strategies

BENEFITS

* From Ada Formal Semantics:

  Concise characterization of "Portable Ada"

  Verified Gypsy to Ada translation

* From Specification and Implementation Strategies:

  PVS - based Verification Capability

Appendix R

Near-Term Ada Verification

Ryan Stansifer
ORA

Ryan Stansifer: Near-Term ADA Verification

Modifying existing verifications systems, e.g., GYPSY, AFFIRM, Stanford Pascal verifier, to be ADA verification systems is nearly as hard as starting from scratch, since one must go down to the level of the LISP implementation. A possible exception is the PL/CV verifier which uses a high-level description language to define the actions of the underlying logic engine given the particular syntax of the programming language to be verified.

A new approach is suggested for the near term which makes use of the Cornell synthesizer generator (Teitelbaum and Reps). The synthesizer generator takes an attribute grammar as input and produces as syntax-directed editor as output. By the correct choice of attributes (Reps and Alpern) a verification system can be made which produces a set of verification conditions (VCs) from an ADA program. By giving these VCs to a theorem prover the program can be checked against its specification.

To illustrate the technique here is a simplified sample of the input to the synthesizer generator for a syntax-directed VC generator.

```
S  ::=  x := e
                S.VC := {}
                S.PreCond := subst e for x in S.PostCond

S  ::=  if b then A else B
                S.VC :=  A.VC + B.VC
                S.PreCond := (b => A.Precond) & (~b => B.PreCond)
                A.PostCond := S.PostCond
                B.PostCond := S.PostCond
```

The method allows experimentation and is easily extensible to new proof rules for ADA as they are developed. The system can use any theorem prover to prove the VCs, so that existing theorem provers can be used. Better theorem provers can be used later as they become available.

Near Term ADA Verification

Modify existing systems

- GYPSY, AFFIRM, SPV, PL/CV

Cornell synthesizer generator

- existing software
- extensible, allows experimentation
- current verification methodology

Cornell synthesizer (Teitelbaum and Reps)

- language-based programming environments
- syntax-directed editor
- generalized to a synthesizer generator

Attribute Grammars

Context free grammars with attributes attached to the symbols

Associated with each production is a semantic definition

Two kinds of attributes: synthesized and inherited

Semantic definitions define values for all synthesized attributes of the LHS nonterminal, and all inherited attributes for RHS symbols.

Simple WHILE languages

$P ::= S$

$S ::= S_1 \; ; \; S_2$

$S ::= x := e$

$S ::= \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2$

$S ::= \textbf{while } b \textbf{ inv } I \textbf{ do } S_1 \textbf{ end}$

Attributes

Nonterminal P (program)

synthesized attributes:

VCs	(set of verification conditions)
ProgPre	(assertion)
ProgPost	(assertion)

Nonterminal S (statement)

synthesized attributes:

VC	(set of verification conditions)
PreCond	(assertion)

inherited attributes

PostCond	(assertion)

VC Generator

```
P  ::=  S  { Q }

          P.VC  :=  S.VC

          P.ProgPre  :=  S.PreCond

          P.ProgPost  :=  Q


S  ::=  x := e

          S.VC  :=  {  }

          S.PreCond := Substitute e for x in S.PostCond


S  ::=  if  b  then  S_1  else  S_2

          S.VC  :=  S_1.VC  +  S_2.VC

          S.PreCond  :=
                  (b => S_1.PreCond) & (~b => S_2.PreCond)

          S_1.PostCond  :=  S.PostCond

          S_2.PostCond  :=  S.PostCond
```

Interface with TP

Particular syntax

e.g., Cambridge prefix

Translate references

```
A [i]  -->  ArraySelect (A, i)
```

Preface of facts and definitions

Appendix S

Verifying Ada Programs

Raymond J. Hookway
Case Western Reserve University

Verifying Ada* Programs

Raymond J. Hookway

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

December 14, 1984

The inductive assertion technique, which has been used successfully as a basis for verifying programs written in Pascal and some of its derivatives[9,10,15,17,20], is directly applicable to a large part of Ada. However, Ada also includes a number of constructs whose verification is not as well understood as the verification of constructs found in Pascal. These include packages, generic program units, tasks and exceptions. The following is a description of our approach to the verification of these constructs.

Packages

Packages in Ada can be used in a number of different ways. One way to use a package is just as the name of a collection of data and type declarations. This kind of use poses no special verification problems and can be handled using standard techniques.

The more important use of packages is their use for the implementation of abstract data types. Packages can be used to support data abstraction in two ways. One of these is to associate an abstract object with each package (or each instantization of a generic package). The entries to the package are then viewed as operations on the abstract object. This constitutes one of the "standard" approaches to data abstraction. Packages used in this way can be verified using the method first proposed by Hoare [11].

The other way to achieve data abstraction is to associate an abstract type with a private type which is declared in a package specification. Ada supports this kind of abstraction by restricting the operations that can be performed on a (limited) private type to (assignment and equality test plus) the entries to the package in which the type is declared. This latter approach to data abstraction is also found in Modula where types can be exported from a module. Packages which contain private types can be verified using techniques developed at Case (Ernst and Ogden [8], Hookway [14]) for the specification and verification of data abstraction in Modula programs.

Generic Program Units

Our approach to specifying and verifying generic program units is to allow generics to have parameters which are predicates and functions of the specification language. This is an extension of the usual Floyd/Hoare assertion language found in the literature. A brief description of this approach is given below. A more detailed description is given in Ernst and Hookway [6].

Consider a generic program unit G that has a type parameter T and a procedure parameter p(x,y). (This description is not concerned with the types of parameters or Ada syntax.) The specifications of G depend on what p does. This can be specified by giving pre- and post-conditions for p. These assertions, like ordinary assertions, contain predicates and functions of the specification language (and also individual variables and constants). Unlike ordinary assertions, some of these predicates and functions are formal parameters of G just like T and p.

No special techniques are required to deal with type parameters (like T in the above example). This is because type parameters play the usual role of types in verification, they assure that the program is "well-formed". Specifications are also required to be well formed.

Each generic program unit is required to have a precondition which may contain specification language functions and predicates that are parameters to the generic. This assertion specifies the properties of these parameters which can be assumed in verifying the body of the generic.

*Ada is a registered trademark of the U.S. Department of Defense (OUSDRE-AJPO)

Instantiation of generic program units is handled by substituting actual parameters for formals. The specifications of the resulting ordinary program unit (IG) are just those of the generic with formal parameters replaced by actuals. This substitution removes all formal predicates and functions of the specification language. The specifications of the instantiated program unit thus have the same form as a non-generic program unit. Of course, it must be verified that the actuals satisfy the assumptions made about them in the generics specifications.

Tasks

We hope to adapt the method described in Ernst and Hookway [5], and Ernst [4] to the problem of verifying concurrent Ada programs. This method requires concurrent programs to be structured as a collection of modules**. Each module defines one or more data abstractions, and any number of processes may be declared local to the body of the module. The purpose of these processes, called *realization processes*, is to manipulate the module's local variables in a way that does not affect the value of the abstract objects represented by the module. Although this is a very specific way to structure programs, it appears that most real software can be naturally structured in this manner. This approach allows modules to be verified separately even though the realization processes of one module execute concurrently with those of other modules.

Modules are verified by dividing the process and the entry procedures to the module into single mutex segments (SMSs) each of which contains at most a single critical section. The proof technique relies on the fact that, under certain restrictions, every concurrent execution of the SMSs produces the same result as some sequential execution of them. Sequential verification techniques can then be used to prove that the SMSs have the properties required for the module to meet its specifications.

The soundness of this approach depends on the fact that shared variables are accessed only under mutual exclusion. This is a severe restriction to place on the implementation. In order to ease this restriction, ownership specifications are added to modules. Ownership specifications allow shared variables to be treated as local to a process. Ownership is dynamic. A variable may be "owned" by one process at a given time and a different process at a later time. Processes are also allowed to "own" components of structured variables. Thus, one process can "own" one component of an array at the same time that another process "owns" a different component of the same array. However, it must be verified that two processes never "own" the same object at the same time and that processes only reference objects which they own.

The approach to verifying concurrent programs described above is the subject of active research at Case. Significant additional effort will be required to extend this approach to apply to Ada. In particular, the synchronization primitives used in Ada tasks are quite different from those studied by Ernst and Hookway [5] and exception handling in multi-task programs remains to be examined. Despite these difficulties, this appears to be a very promising approach to the verification of multi-task Ada programs.

Exceptions

Exceptions can be handled using the technique developed by Luckham [19]. Extensions to the this technique need to be developed to integrate exception handling with the techniques for data abstraction discussed above.

A Prototype Verifier

We are currently in the process of implementing a verifier for an Ada subset which is roughly equivalent to Modula. This implementation includes packages and private types. Addition of generics and exceptions, as described above, should be straight forward. The verifier will use the Case interactive theorem prover which is part of the Modula verifier described below.

**Modules correspond closely to packages in Ada and processes to tasks. The exact relationship of the concepts described in Ernst and Hookway [5] to Ada remains to be worked out.

A Design Environment

We feel that development of reliable software will require support of an integrated design environment. This environment should support a variety of approaches to verification from testing to theorem proving. However, it should be based on the notion of developing designs that are consistent with *precise* specifications. The environment should provide a framework for reasoning about designs. For example, it should track arguments about why portions of the design are correct, whether the arguments are based on test data, informal arguments, or formal (mechanical) proofs. Whatever form these arguments take, we expect them to be based on an understanding of what is required to formally verify the design.

We plan to build a series of incrementally updatable design environments based on the above ideas. The Ada verifier will be one component of these environments. Other components will include tools for developing and analyzing specifications, a facility for rapid prototyping, and a programming environment.

The Case Modula Verifier

The Case Verifier is an interactive system for verifying Modula programs. The verifier consists of two major components, a verification condition generator (vcg) and an interactive theorem prover. The source language is Modula, minus concurrent programming constructs and extended by the constructs described in Ernst and Ogden[8] and Hookway[14] for specifying Modula programs. The vcg generates verification conditions by symbolically executing the source program as described in Dannenberg and Ernst[3].

The theorem prover is an interactive, natural deduction theorem prover which was developed at Case. The design of this theorem prover is described in Ernst and Hookway[7]. The goal of this design was to produce a small, efficient theorem prover to support our research in verification methodology.

The verifier has been used to verify a small linking loader[14]. The loader is approximately four *hundred lines long, divided equally between specifications and code. Selected verification conditions were* proved using the theorem prover described above.

References

1. Ada Programming Language, Military Standard MIL-STD-1815, U.S. Department of Defense, December, 1980.

2. Formal Definition of the Ada Programming Language, Institut National de Recherche en Informatique et en Automatique, November, 1980.

3. Dannenberg, R.B. and Ernst, G.W., Formal Program Verification Using Symbolic Execution, *IEEE Transactions on Software Engineering*, Jan., 1982.

4. Ernst, G.W., A Method for Verifying Concurrent Processes, Report No. CES-82-1, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

5. Ernst, G.W. and Hookway, R.J., The Use of Data Abstraction in the Specification and Verification of Concurrent Programs, Internal Document, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

6. Ernst, G.W. and Hookway, R.J., Specification and Verification of Generic Program Units in Ada, Internal Document, Computer Engineering and Science Dept., Case Western Reserve Univ., 1982

7. Ernst, G.W. and Hookway, R.J., Mechanical Theorem Proving in the Case Verifier, *Machine Intelligence 10, Practice and Perspective*, Ellis Horwood Ltd., 1982, pp.123-145.

8. Ernst, G.W. and Ogden, W.F., Specification of Abstract Data Types in Modula, *ACM Transactions on Programming Languages and Systems*, Oct., 1980, pp.522-543.

9. German, S.M., An Extended Semantic Definition for Pascal for Proving the Absence of Common Runtime Errors, Report No. STAN-CS-80-811, Stanford University, June, 1980.

10. Good, D.I., Cohen, R.M., Hoch, C.G., Hunter, L.W. and Hare, D.F., Report on the Language Gypsy, Version 2.0, Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

11. Hoare,C.A.R., Proof of Correctness of Data Representations, *Acta Informatica*, 1972, pp.271-81.

12. Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *Communications of the ACM*, Oct., 1969, pp.576-580.

13. Hoare, C.A.R. and Wirth, N., An Axiomatic Definition of the Programming Language Pascal, Acta Informatica, 1973, pp.335-355.

14. Hookway, R.J., Verification of Abstract Data Types whose Representations Share Storage, Report CES-80-2, Computer Engineering and Science Dept., Case Western Reserve Univ., 1980.

15. Hookway, R.J. and Ernst, G.W., A Program Verification System, *Proc. of the Annual Conference of the ACM*, 1976, pp.504-508.

16. Ichbiah, J.D., Barnes, J.G.P, Heliard, J.C., Krieg-Brueckner, B., Roubine, O. and Wichmann, B.A., Rationale for the Design of the Ada Programming Language, *SIGPLAN Notices*, June, 1979.

17. Igarashi, S., London, R.L. and Luckham, D.C., Interactive Program Verification: A Logical System and Its Implementation, Acta Informatica, March 1975, pp.145-182.

18. London,R.L., Guttag,J.V., Horning,J.J., Lampson.B.W., Mitchell,J.G. and Popek,G.J., Proof Rules for the Programming Language Euclid, *Acta Informatica*, Jan. 1978, pp.1-26.

19. Luckham, D.C. and Polak, W., Ada Exception Handling - An Axiomatic Approach, *ACM Transactions on Programming Languages and Systems*, May, 1980.

20. Luckham, D.C., German, S.M., v.Henke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak, W. and Scherlis, W.L., Stanford Pascal Verifier User Manual, Report STAN-CS-79-731, Computer Science Department, Stanford University, 1979

21. Luckham, D.C. and Suzuki, N., Verification of Array, Record and Pointer Operations in Pascal, *ACM Transactions on Programming Languages and Systems*, Oct. 1979, pp.226-244.

22. Wirth,N., Modula: A Language for Modular Multiprogramming, *Software--Practice and Experience*, Jan., 1977, pp.3-35.

```
module stacks;
    define stack, push, pop, empty;

    type stack = record
        a : array [1..max] of integer;
        p : 0..max;
    end;

    procedure push(const x : integer; var stk : stack);
    begin
        if stk.p = max then error('stack overflow') end;
        stk.p := stk.p + 1;
        stk.a[stk.p] := x;
    end push;

    procedure pop ...
    function empty ...

end stacks;
```

```
package stacks is
    stack_overflow, stack_underflow: exeception;
    type stack is limited private;
    procedure push(x : integer; stk : in out stack);
    procedure pop(x : out integer; stk : in out stack);
    function empty(stk : stack) return boolean;
private
    max: constant := 100;
    type integer_vector is
        array (integer range <>) of integer;
    type stack is
        record
            a : integer_vector(1..max);
            p : integer range 0..max := 0;
        end record;
end stacks;

package body stacks is

    procedure push(x : integer; stk : in out stack) is
    begin
        if stk.p = max then
            raise stack_overflow;
        end if;
        stk.p := stk.p + 1;
        stk.a(stk.p) := x;
    end push;

    procedure pop ...
    function empty ...

end stacks;
```

```
module stackmodule;
     define symbolstack, push, pop, empty;

     abstract type symbolstack;
          abstract structure as : seq of symbol;
          realization structure rs : record
               sa : array [1..max] of symbol;
               p : 0..max;
               av : array [0..max] of seq of symbol end;
          correspondence as = rs.av[rs.p];
          invariant 0 <= rs.p <= max & rs.av[0] = emptyseq &
                    forall i (1 <= i <= rs.p →
                         rs.av[i] = concat(mkseq(rs.sa[i]), rs.av[i − 1]));
          initialization;
          entry assertion true;
          exit assertion as = emptyseq;
          begin
               rs.p := 0; rs.av[0] := emptyseq
          end initialization;
     end symbolstack;

     procedure push(const s : symbol; var stk : symbolstack);
     entry assertion true;
     exit assertion stk = concat(mkseq(s), #stk);
     begin
          with sa ::= stk.sa, p ::= stk.p, av ::= stk.av do
          if p = max then error('symbolstack overflow') end;
          p := p + 1; sa[p] := s;
          av[p] := concat(mkseq(s), av[p − 1])
     end
     end push;

     procedure pop ...
     function empty ...
end stackmodule;
```

```
with types; use types;
package stackmodule is
     stack_overflow, stack_underflow: exeception;
     type symbolstack: symbolsequence
          initially symbolstack'as = emptyseq
          is limited private;
     procedure push(s: symbol; stk: in out symbolstack) is
          exit assertion stk = concat(mkseq(s), stk'init);
     end push;
     procedure pop(s: out symbol; stk: in out symbolstack) is ...
     function empty(stk: symbolstack) return boolean is ...
private
     max: constant := 100;
     type symbolstack is
          record
               sa: symbol_vector(1..max);
               p : integer range 0..max := 0;
               av: auxiliary symbolsequence_vector(0..max) :=
                         (0 => emptyseq,others => emptyseq);
          end record;
          correspondence
               symbolstack'as = symbolstack'rs.av(symbolstack'rs.p);
          invariant
               0 <= symbolstack'rs.p <= max and
               symbolstack'rs.av(0) = emptyseq and
               forall i (1 <= i <= symbolstack'rs.p →
                    symbolstack'rs.av(i) =
                         concat(mkseq(symbclstack'rs.sa(i)),
                              symbolstack'rs.av(i - 1)));
     end symbolstack;
end stackmodule;
```

```
with types; use types;
package stacks is
     stack_overflow, stack_underflow: exeception;
     type stack: pair_sequence
          initially stack'as = emptyseq
          is limited private;
     procedure push(s: symbol; v: integer; stk: in out stack) is
          exit assertion exists x (stk = concat(mkseq(x), stk'init) and
               x.s = s and x.v = v);
     end push;
     procedure pop(s: out symbol; v: out integer; stk: in out stack) is ...
     procedure assign(stk1: out stack; stk2: stack) is ...
     function empty(stk: stack) return boolean is ...
private
     n: constant := 5280;
     sq: array (0..n) of pair_sequence :=
               (0 => emptyseq, others => emptyseq);
     type stack is
          record
               p: integer range 0..n := 0;
          end record;
          correspondence stack'as = sq(stack'rs.p);
          invariant 0 <= stack'rs.p <= n;
     end stack;
end stacks;
```

```
package body stacks is
    l:   array (1..n) of integer;
    sv: array (1..n) of pair;
    free: integer := n;
    f:   array (1..n) of boolean := (others => true);

    invariant
        0 <= free <= n and ...;

    procedure push(s: symbol; v: integer; stk : in out stack) is
        uses l, sv, free, f, sq;
        x: pair := (s, v);
        y: integer := free;
    begin
        if free = 0 then
            raise stack_overflow;
        end if;
        sq(free) := concat(mkseq(x), sq(stk));
        f(free) := false;
        free := l(free); sv(y) := x; l(y) := stk.p; stk.p := y;
    end push;

...
begin
    for i in l'range loop alter l, sq;
        maintain
            forall x (1 <= x <= i →
                l(x) = x − 1 and
                sq(x) = concat(mkseq(sv(x)), sq(x − 1)) and
                f(x)) and
            sq(0) = emptyseq;
        l(i) := i − 1;
        sq(i) := concat(mkseq(sv(i)), sq(i − 1));
    end loop;
end stacks;
```

```
package body stacks is
    l:   array (1..n) of integer;
    sv: array (1..n) of pair;
    free: integer := n;
    f:   array (1..n) of boolean := (others => true);

invariant
    0 <= free <= n and forall i (1 <= i <= n → 0 <= l(i) <= n) and
    1 <= free → f(free) and
    forall i (1 <= i <= n and 1 <= l(i) <= n → (f(i) ↔ f(l(i)))) and
    forall i (1 <= i <= n →
        (f(i) → forall j (1 <= j <= stack'alloc → stack'r(j).p /= i)) and
        (free = i → forall j (1 <= j <= n → l(j) /= i)) and
        forall j (1 <= j <= n and l(j) = i and f(j) →
            forall k (1 <= k <= n and j /= k → l(k) /= i))) and
    sq(0) = emptyseq and
    forall i (1 <= i <= n → sq(i) = concat(mkseq(sv(i)), sq(l(i))));

...
end stacks;
```

```
with types; use types;
package st: array (symbol) of integer initially (forall s st(s) < 0) is
     procedure enter(s: symbol; v: integer) is
          uses st;
          exit assertion st(s) = v and forall i (i /= s → st(i) = st'init(i));
     end enter;
     procedure lookup(s: symbol; v: out integer) is
          uses st;
          exit assertion v = st(s);
     end lookup;
end st;

with stacks; use stacks;
package body st is
     m: constant := 64;
     ht: array (1..m) of stack;
     correspondence forall s (st(s) = assoc(s, ht(hash(s))));
     procedure enter(s: symbol; v: integer) is
          uses ht;
     begin
          push(s, v, ht(hashfn(s)));
     end enter;
     procedure lookup(s: symbol; v: out integer) is
          uses ht;
          s1: symbol; stk: stack;
     begin
          assign(stk, ht(hashfn(s)));
          loop alter s1, v, stk;
               maintain assoc(s, ht(hash(s))) = assoc(s, stk);
               if empty(stk) then v := -1; exit; end if;
               pop(s1, v, stk);
               exit when s = s1;
          end loop;
     end lookup;
end st;
```

```
generic type item is private;
package stacks is
     stack_overflow, stack_underflow: exeception;
     type stack is private;
     procedure push(x : item; stk : in out stack);
     procedure pop(x : out item; stk : in out stack);
     function empty(stk : stack) return boolean;
private
     max: constant := 100;
     type item_vector is
          array (integer range <>) of item;
     type stack is
          record
               a : item_vector(1..max);
               p : integer range 0..max := 0;
          end record;
end stacks;

package body stacks is

     procedure push(x : item; stk : in out stack) is
     begin
          if stk.p = max then
               raise stack_overflow;
          end if;
          stk.p := stk.p + 1;
          stk.a(stk.p) := x;
     end push;

     procedure pop ...
     function empty ...

end stacks;
```

```
generic type $T$ is private;
    with $f$ ...;
    with $Q$ ...;
    with procedure $p$ ($x$ : in $integer$; $y$ : in out $T$) is
        entry assertion ...;
        exit assertion $forall\ u,v(f(x) < u\ and\ u < v \rightarrow Q(x,v))$;
    end $p$;
    require $forall\ x,y,z\ (Q(x,y)\ and\ Q(y,z) \rightarrow Q(x,z))\ and$ ...;
package $G$ is
    procedure $r$ is
        exit assertion ...; -- uses $f$ and $Q$
    end $r$;
end $G$;
```

Appendix T

Adapting the Gypsy Verification System to Ada

John McHugh
Research Triangle Institute

Karl Nyberg
Verdix Corporation

Adapting the Gypsy Verification System to Ada

Workshop on Formal Specification and Verification of Ada
Institute for Defense Analysis
18-20 March 1985

John McHugh - Research Triangle Institute
Karl Nyberg - Verdix Corporation

1. Introduction

DoD directive 5000.31 [DoD] requires that new mission critical computer programs written for the department of defense be written in Ada[1] [Ada]. The statutory definition of mission critical (10 USC 2315) includes security applications specifically. Computer security has been one of the principle driving forces for applied verification work in recent years. These factors lead us to one of two conclusions: 1) The time is rapidly approaching when it will be necessary to apply verification techniques to programs written in Ada; or 2) DoD 5000.31 will have to be modified to exclude secure systems. While there exists a well known antipathy towards Ada within parts of both the verification and the computer security communities, it is unlikely that the DoD policy towards Ada will undergo substantial change in the near future. If this is the case, it will be necessary to develop an Ada verification capability in the near future.

There are several ways in which such a capability could be developed. A first option would be to start from scratch, using any of the formal models of program specification and verification and build a system specifically designed to verify Ada programs. A second option is to ignore the Ada specific aspects of the problem entirely. Under the current certification criteria of the DoDCSC, it is not necessary to deal with the implementation language for a system in a formal manner, so it could be argued that current systems are just as suitable (or unsuitable) for Ada as for any other language. In this case, it is only necessary to provide a convincing argument for the conformance of the Ada implementation code to the verified formal top level specification of the system in question. Finally, it is possible to adapt an existing verification system to deal with Ada.

The first approach is possible, but would take an excessive amount of time and resources. Current verification systems represent investments of ten or more man years each, expended over periods of five to ten years. The second approach is representative of the practice followed for the Honeywell SCOMP, a product currently approaching A1 certification by the DoDCSC. It appears that the requirement for a convincing argument concerning the equivalence of the FTLS and the implementation resulted in an extremely complex and concrete FTLS and greatly increased the verification effort. Being able to verify an Ada based FTLS for an Ada based implementation should

[1] Ada is a registered trademark of the Ada Joint Project Office.

obviate these difficulties. Additionally, there is substantial interest in systems which go beyond the A1 criteria by requiring code verification for which second approach would not be viable. The third approach offers a chance to capture much of the investment in a current verification system while gaining experience with the verification of Ada. We argue for such an approach, based on the Gypsy [Good78] system, suggesting that it will lead to a prototype code verification system for Ada with minimum (although not insubstantial by any means) effort. Taking advantage of the Ada packaging mechanism, we feel that verified packages can function within a larger Ada environment, making possible the implementation of security kernels and the like.

The remainder of the paper discusses some of the problems associated with the verification of Ada, suggests ways in which these problems might be addressed, and indicates ways in which the Gypsy system could be combined with the front end of an Ada compiler and transformed into a prototype system for the verification of Ada.

2. Trouble spots in Ada

Although one of the early design objectives for Ada (in the days when it was still known as DoD-1) was to facilitate proofs of program properties, the committee nature of the requirements process resulted in a language which was required to carry a certain amount of the baggage of 1960s style programming languages. Among the potentially most troublesome of these are the presence of arbitrary control flow constructs i.e. the "go to" statement, and unrestricted access to global variables which, in addition to complicating proofs about sequential programs, render concurrent programs intractable under many circumstances. Other features of the language include the possibility of side effects from function invocations, exceptions during expression evaluation, and the lack of an explicit evaluation order for the operators of an expression. These factors, combined with the lack of a formal definition for the semantics of the language, have lead some workers to despair of verifying any aspect of the language. Indeed, it has been noted that given the proper Ada context, it may be impossible to prove anything about the value of X after the execution of so simple a statement as

```
X := 1;
```

We maintain that the situation is not quite as grim as indicated above. Just because a language contains a particular feature does not mean that all programs written in the language will contain that feature. The adverse interaction among features of the language, does not mean that all of them must be discarded, or that all occurrences of a feature in a given program are intractable. Although the word "subset" is an anathema to the Ada world, we feel that a useful set of Ada constructs and programming practices can be defined in such a way that realistic and functional programs can written and verified using them. Although the task is substantially more difficult, because of the extra complexity of the language, we feel that a theory of verifiable Ada can be developed in much the same way as Boyer and Moore developed their FORTRAN

[Boyer80] theory. Platek [Odyssey84] and his colleagues at Odyssey Associates have recently defined an initial subset of Ada which they feel is suitable for verification. One feature which they rule out is the exception mechanism. We feel that the Ada exception mechanism is sufficiently like the Gypsy mechanism so that its verification is tractable, and we propose to include exceptions in our system.

Ada as currently defined has no specification mechanism. While it is possible to use an external specification mechanism, i.e. one in which the program and specification are joined only during the verification process, we are more comfortable with an internal mechanism, similar to that used in Gypsy. At the same time, we would like our verifiable code to be acceptable to a variety of Ada translators. An extension of Luckham's Anna notation [Luckham84] to accommodate exception returns from routines appears to be the most promising mechanism available at the present time, although a specification language using the Ada PRAGMA construct cannot be ruled out.

3. A hybrid system

We propose to base our prototype Ada verification system on a combination composed of an existing Ada compiler and an existing verification system. The Ada compiler is the one developed and recently validated by the Verdix corporation of McLean, Virginia, while the verification system is the Gypsy Verification Environment, developed at the University of Texas. There are several reasons for the choice of such a hybrid system. Ada is a large language with a complex syntax and semantics. Using an existing front end from a validated compiler eliminates much of the effort required to implement a front end for the verification system. It also provides a direct method for providing executable versions of the verified programs, as well as facilitating systems which contain mixtures of verified and unverified programs. The use of a modified version of the GVE as a back end for the Ada verification system offers similar advantages. We feel that the initial set of Ada constructs which can be verified will be roughly equivalent in power and flavor to the Gypsy language. Previous efforts to model Ada constructs in Gypsy [Akers83], and vice versa provide evidence for this assumption. Although Ada type rules are "stronger" than those for Gypsy, it is possible to write Gypsy as though it were typed like Ada. The Gypsy exception mechanism, though somewhat more tractable than the Ada exception mechanism is suitable for modeling Ada. Most of the Ada operators are already present in Gypsy.

The proposed hybrid consists of three primary components, the Ada front end, the intermediate form translator, and the verification back end. Each of these are described briefly in the sections which follow.

4. The Ada front end

As noted above, the front end of the proposed system is based on the parser and semantic checker of an existing, validated, Ada compiler. The parser and semantic checker will require some modifications to accept Ada with embedded specifications. The output of the modified front end will consist of the compiler's internal representation of Ada programs, extended to include the specification constructs. Assuming that a specification language such as Anna is chosen, these modifications should be relatively straight forward. The internal representation will be captured at a stage in the compilation process where name resolution has been performed and operator overloading has been removed so as to simplify subsequent operations.

5. The intermediate form translator

The intermediate form translator serves a dual purpose. Its primary function is to convert the Ada compiler's representation of a program into a representation which can be entered into the verification back end as though it were the output of the Gypsy parser. Its secondary function is to ensure that the code to be verified conforms to the set of constructs acceptable to the verification system, i.e. that the program to be verified is in fact written in the verifiable Ada subset. Given that both the Ada front end and the Gypsy back end use internal representations which are abstractions of prefix trees, the translation operation is a straightforward, if complex, syntactic one. The enforcement function, on the other hand, may involve substantial semantic analysis. It is hoped to simplify both of these tasks by taking advantage of utilities, already present within the front end, for manipulating the internal form of Ada programs.

6. The modified GVE

The output of the translation process will be a Gypsy-like representation of the Ada code to be verified in a form suitable for loading into the modified GVE. Once such an Ada database has been restored into the GVE, verification conditions can be generated and proved in the same way these steps are performed for Gypsy programs in current versions of the system. To support Ada verification, substantial modifications will be required for a number of components of the GVE. The verification condition generator will require modification to reflect the semantic differences between Ada and Gypsy statements. In a similar fashion, the expression simplifier will also require modification and extension. The prefix to infix conversion routine, used to display internal forms to the user will be modified to use an Ada syntax. We hope to take advantage of the previous work on a Gypsy to Ada translator for much of this step. It is hoped that the prover will require little or no modification. Modifications to the top-level or user interface to the system should be restricted to the removal of unneeded functionality and system components such as the optimizer and code generators.

7. Summary and conclusions

We have proposed a prototype Ada verification system based on a hybrid of an existing compiler and verification system. Although such a system is not capable of supporting verification of the entire Ada language, it is claimed that it will support a language comparable to those now being verified and suitable for similar programs. While the construction of such a system involves a substantial effort, we are confident that the effort is much less than that involved in building a verification system for Ada from scratch. A hybrid system, such as we propose, will allow the verification community and the growing applications community it supports to obtain experience with Ada verification in the near future. Such experience will provide a sound basis for future revisions of the language to support verification should this prove necessary or desirable.

8. References

[Ada] - *Ada Programming Language,* ANSI/MIL-STD-1815A, Department of Defense, 22 January 1983.

[Akers83] - Akers, Robert L., *A Gypsy-to-Ada Program Compiler,* Technical Report 39, Institute for Computing Science, The University of Texas at Austin, Austin, TX 78712, December 1983.

[Boyer80] - Boyer, Robert S., Moore, J Strother, *A Verification Condition Generator for Fortran,* Technical Report CSL-103, SRI International, June, 1980.

[DoD] - DeLauer, Richard D., "DoD Policy on Computer Programming Languages", Department of Defense Directive 5000.31, The Under Secretary of Defense, Washington, DC, 20301, June 1983.

[Good78] - Good, Donald I., Cohen, Richard M., Hoch, Charles G., Hunter, Lawrence W., Hare, Dwight F., *Report on the Language Gypsy, Version 2.0,* Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX 78712, September 1978.

[Luckham84] - Luckham, David C., von Henke, Friedrich W., Krieg-Brueckner, Bernd, Owe, Olaf, *Anna - A Language for Annotating Ada Programs, Preliminary Reference Manual,* Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

[Odyssey84] - Odyssey Research Associates, Inc. *A Verifiable Subset of Ada,* (Revised

Preliminary Report), Odyssey Research Associates, Inc., 713 Clifton St., Ithaca, NY 14850, 14 September 1984.

Adapting the Gypsy Verification System to Ada

19 March 1985

John McHugh

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, NC

Karl Nyberg

Verdix Corporation
Westgate Research Park
McLean, VA

Why?

- DoD Directive 5000.31
    - Ada required for mission critical software
- 10 USC 2315
    - Mission critical includes security
- TCSEC
    - Verification required for A1
- QED

How?

- Start over
- Ignore Ada
- Adapt existing tools

Start Over

- Pro
  - Language is different
  - Need a second generation system
- Con
  - Excessive cost and effort
  - No experience with many Ada constructs

Ignore Ada

- Pro
  - TCSEC does not require code proofs
  - Design verification is easier
- Con
  - Possible conflicts between code and specifications
  - The SCOMP experience

Adapt tools

- Pro
  - Minimize effort
  - Early availability
  - Proven base products
- Con
  - Restricted set of constructs
  - Warps Ada in particular direction

Adapting Gypsy

- Strong intersection
  - Types
  - Exceptions
  - Expression languages
  - Aliasing and side effect semantics
- Disjoint features
  - Tasks vs. cobegins
  - I/O packages vs. buffers
  - Reals vs. rationals

Trouble spots in Ada

- See David Guaspari
    - But take what he says with a grain of salt

Hybrid system

- Verdix front end
  - Modified for Anna-like specifications
- Diana to Prefix translator
  - Enforces verifiable intersection
- Gypsy back end
  - Ada semantics where different
  - Ada syntax for interaction

Front end

- Extensions for specifications
- Enforcement of specification semantics
- Issues:
  - Diagnostics
  - Diana modifications
  - Details

Translator

- Visualized as front end extension
- Checks for verifiable subset
- Convert from Diana / Ada to Gypsy / Prefix
- Produces Gypsy database
- Issues:
    - Incremental Methods

Back end

- Ada syntax
  - Use Gypsy —> Ada facility
- Ada semantics
  - Expressions — modify expression evaluator
  - Statements — modify verification condition generator

System constraints

- Exceptions imply constraints on:
    - Expression evaluation
    - Parameter passing
- Implementation details:
    - Affect semantics of operations
    - Provide basis for proofs about exceptions

Appendix U

Discussion of Papers from the Session on Near-Term Verification Systems

Discussion of papers on Near-Term Verification Systems

The speakers were invited to offer estimates of the time needed for completion of their proposals.

J. Williams: 2 years for the first stage; 1990 for completion.

R. Stansifer: 6 months for the well-understood part of ADA.

R. Hookway: 12 months.

J. McHugh: 1 to 2 years.

The following is a condensed version of the discussion that took place about formal semantics.

D. Milton: If formal semantics equired for ADA, is it to be the sort done for preliminary ADA? Is that the kind of formal semantics that would be useful?

R. Platek: You do need a mathematical model against which you prove that your proof rules are sound. You cannot use the Reference Manual, which is written in English, to prove that a proof rule really captures the language.

There is no doubt that to feel completely assured you have to go by the route of an abstract model. Now what kind of devices we will use, whether it will be denotational semantics, or what, that is open.

LCDR Myers: Would the model stand by itself?

R. Platek: It could.

LCDR Myers: Can this community use such a thing?

D. Luckham: What was the experience with the pseudo-denotational semantics produced for the preliminary 1979 version of ADA?

N. Cohen: Several type inconsistencies were discovered.

D. Luckham: First of all, they could not accommodate the semantics of tasking at all, is that correct? Secondly, was there any agreement that the formal semantics they defined was the definition of the language?

N. Cohen: That couldn't possibly be because it was incomplete.

D. Luckham: One of the purposes of this formal definition was as a guide to compiler writers. I understand that no compiler writer ever used it as a guide.

N. Cohen: In Denmark they developed a semantics for the 1978 preliminary ADA, at the last minute they updated it to 1980 ADA, and apparently behind the scenes they continued working on it because they then used the formal definition as a basis for a compiler system which has now been validated. Thus that compiler was generated from the formal definition. The generation was not wholly automatic: there was a substantial manual involvement. There was an operational definition of tasking.

J. McHugh: I think that for such a definition to be useful not only for this group but for the ADA community as a whole, it will have to be judged to be the arbiter of all the disputes about the definition of the language.

N. Cohen: Currently it is the position of the AJPO that the Reference Manual is such a definition, but there are omissions in the Reference Manual, and there are places where it is vaguely worded and the ADA Board has to meet and decide on interpretations.

D. Luckham: The formal definition produced by INRIA is available. You can read it and ask your question of that and do a symbolic computation to get the answer--try executing the recurrence relations.

B. Abrams: There is a formal definition of JOVIAL: the entire language is specified, as an attribute grammar, in some 300 pages of BNF. When you look at all that, you cannot be assured that it is correct and that there is nowhere an error.

LCDR Myers: I am hearing that the language isn't there, that you need to formalise the semantics of this language.

Answers of both "yes" and "no" were heard.

R. PLatek: I think it has to be done right--in a way that is usable.

The style of the INRIA manual makes it an unusable model. There are other styles of presenting models, one of which might turn out to be a form that is usable... What is "usable" is a theological question.

D. Luckham: We should standardise ADA by picking a compiler. Minsky suggested taking the LISP interpreter as a formal definition. It seems to me that a well-written compiler is a good standard. You can run it; you cannot run the formal definition of ADA.

R. Platek: That does not mean that some form of mathematical model could not be run.

N. Cohen: ADA was designed as essentially several dialects; There is no way to take a compiler as a definition and to keep ADA objectives such as portability.

D. Luckham: The thing just tells you that this is an implemention feature when you ask it a question.

N. Cohen: Well, if your standard is a compiler, it is not going to tell you that it is implementation dependent, it will tell you that the word size is 16 bits.

D. Luckham: No, no, no, not a compiler in the stupid sense but a compiler designed to be a standard which can answer questions.

R. Platek: OK, if you are proposing that as the quickest and cheapest method, fine. That is decided. Now let us go on to mid-term. I think it is mid-term to get a mathematical model. Ten years.

LCDR Myers: I hear waffle, waffle, waffle. You say that you need something along this line to do your job. Well, I think I am going to recommend that you all get together and figure out what it is.

An unidentified speaker: The Orange Book does not call for code verification. It appears that the DoD is not willing to trust design systems that are not based on a formal basis. How comfortable are you with user systems that are not grounded in formality ?

LCDR Myers: I would feel a lot better if things were grounded in any other way than the present one. The mere process of going through some sort of formalization is useful I'm for improvement; there is no perfection; we want serious mid- and near-term things.

LCDR Myers: You want a common ground--what should that be?

R. Platek: The piecemeal construction of a theorem prover leads to false theorems. What is needed is a model serving as a final arbiter.

B. Abrams: Just as the Reference Manual is a mixture of formality and examples, so a real definition would mix mathematical and operational bits.

K. Apt: Verification of programs and verification of soundness of proof systems against semantics, and also writing large programs--I believe that all these belong, at a certain level of abstraction, to the same problem, namely it is how we manage the complexity of large tasks. I believe that people should agree that verification of large progams is not a simple issue.

As soon as large subsets of PASCAL or ALGOL-like languages were addressed, subtle errors emerged, such as that found, four or five years after its publication, in Cook's proof of the completeness of a fragment of ALGOL 60 without the use of recursion. This story shows that at a certain level we do not see the details: they are too formal. The error, in brief, was that the semantics was overspecified. Similar errors can occur in compilers. We should not therefore hope that validation of proof systems will be any easier than verification of programs.

D. Luckham: I have no faith in the ability of theoretical mathematicians to turn out formal semantics of any value.

Appendix V

Uses of Formal Verification

Norman Cohen
SofTech

Norman Cohen: Uses of Formal Verification

There are two practical obstacles to proving ADA progams correct. One is that of sheer scale: typical ADA applications are hundreds of thousands of lines long. The other is that a proof of what is commonly called "correctness" is really only a proof of consistency with specifications, and thus makes no attempt to show that the specifications correctly formalise the user's requirements. Formalisation is difficult: about half of all software errors are ones of specification; and so this is a serious deficiency.

We therefore concentrate on proving selected properties of selected program components. We select those components that are amenable to formal specification, use nonobvious algorithms, perform especially critical functions, or are to be re-used, and we aim, where appropriate, to establish the correctness of data abstraction implementations, the absence of unanticipated exceptions, the absence of erroneousness, and certain numeric properties.

Proving the absence of erroneous execution is an ideal application for verification, for proofs can work where traditional compile-time checks are too weak and run-time checks are too expensive.

Proving the absence of unanticipated exceptions forces the explicit documentation of implicit assumptions, reconciles reliability and efficiency by "certifying" uses of the suppress pragma, allows specifications to be as simple as a list of anticipated exceptions, and has other benefits.

Establishing the correctness of data abstractions is a principal use for ADA's most characteristic feature, packages, and permits the building of verified libraries of reusable, possibly generic, components. It may be accomplished by defining the abstract behaviour of a data type in terms of algebraic axioms, which may be given as comments in the package specification, perhaps in ANNA, and then proving that the subprograms in the package body fulfil the axioms. ADA's notions of data type encapsulation help because you know that you can verify the desired property just by looking at the package itself and not outside. The axioms can be used later in verifying other components using the package that provides the data abstraction. The ANNA out attribute provides an excellent vehicle for accommodating procedures, and exceptions are a nice solution to a problem that always existed in the early literature on verifying data abstraction.

In the case of numeric properties the prospects are not quite so bright, but there may be some cases in which reasonable results can be obtained. Typically because machine approximations to real numbers do not obey the mathematical laws of the field of real numbers verification efforts have ignored them. ADA has tried to formalize the behaviour of real numbers using model intervals. These have two important characteristics: they have reasonable formal properties, and they happen to be consistent with typical machine implementations of real numbers. Interval arithmetic, though, is too pessimistic: hardware usually provides more precision and errors tend to average out and cancel, while the model intervals grow wider and wider. Unfortunately the rules of ADA do not justify any stronger method of reasoning about real arithemtic unless you exploit personal knowlege about the underlying representation. So on the one hand it is hard to write portable ADA numeric software; if it can't be proved convergent using model intervals then it may not be convergent in all ADA implementations. On the other hand, for those numeric algorithms where you can prove a numeric property, you can prove portability of the numeric algorithm.

IT IS IMPRACTICAL TO PROVE Ada PROGRAMS CORRECT

- SPECIFICATION PROBLEM
  - A PROOF OF "CORRECTNESS" IS REALLY ONLY A PROOF OF CONSISTENCY WITH SPECIFICATIONS
  - ABOUT HALF OF ALL SOFTWARE ERRORS ARE SPECIFICATION ERRORS
  - FORMALIZING REQUIREMENTS IS DIFFICULT
- SCALE PROBLEM
  - TYPICAL Ada APPLICATIONS ARE HUNDREDS OF THOUSANDS OF LINES LONG

VG1375wo/1 3/18/85

SOFTECH

A MORE FEASIBLE GOAL

- PROVE SELECTED PROPERTIES OF SELECTED PROGRAM COMPONENTS
  - INGREDIENTS OF CORRECT BEHAVIOR
- TARGET COMPONENTS:
  - COMPONENTS AMENABLE TO FORMAL SPECIFICATION
  - COMPONENTS THAT USE NONOBVIOUS ALGORITHMS
  - COMPONENTS PERFORMING ESPECIALLY CRITICAL FUNCTIONS
  - COMPONENTS THAT ARE TO BE REUSED
- TARGET PROPERTIES:
  - CORRECTNESS OF DATA ABSTRACTION IMPLEMENTATIONS
  - NUMERIC PROPERTIES
  - ABSENCE OF UNANTICIPATED EXCEPTIONS
  - ABSENCE OF ERRONEOUSNESS
- ENCOURAGE REUSABILITY BY ENCOURAGING PRECISE STATEMENT OF PRECONDITIONS/POSTCONDITIONS
- AUTOMATE AS MUCH AS POSSIBLE

VG1375wo/2 3/18/85

SOFTECH

CORRECTNESS OF DATA ABSTRACTIONS

- A PRINCIPAL USE FOR Ada PACKAGES
- WELL UNDERSTOOD
- DEFINE ABSTRACT BEHAVIOR OF A DATA TYPE IN TERMS OF ALGEBRAIC AXIOMS
  - AXIOMS CAN BE GIVEN AS COMMENTS IN PACKAGE SPECIFICATION
  - AXIOMS CAN BE USED TO PROVE PROPERTIES OF COMPONENTS USING THE PACKAGE
  - EXCEPTIONS AND PROCEDURES CAN BE ACCOMMODATED
- PROVE THAT THE SUBPROGRAMS IN THE PACKAGE BODY FULFILL THE AXIOMS
  - PRIVATE TYPES GIVE PACKAGE FULL CONTROL OF THE TYPE, MAKE TEXT OUTSIDE THE PACKAGE IRRELEVANT
  - REPRESENTATION FUNCTIONS MAP INTERNAL REPRESENTATIONS TO ABSTRACT VALUES
- BUILD VERIFIED LIBRARIES OF REUSABLE, POSSIBLY GENERIC, DATA ABSTRACTIONS

VG1375wo/3 3/18/85

SOFTECH

NUMERIC PROPERTIES

- MACHINE REALS DON'T BEHAVE LIKE MATHEMATICAL REALS, SO VERIFICATION EFFORTS HAVE TYPICALLY IGNORED THEM
- Ada CHARACTERIZES BEHAVIOR OF REALS IN TERMS OF MODEL INTERVALS
    - AMENABLE TO FORMAL REASONING
    - AMENABLE TO NATURAL IMPLEMENTATION
- CAN DEFINE A THEORY OF REALS IN WHICH OBJECTS ARE MODEL INTERVALS
    - 10.2 MAY DENOTE [10.1875, 10.2500]
    - 6.3* 0.1 MAY DENOTE [6.2500, 6.3125]*[0.0625, 0.1250], OR [0.3750, 0.8125]

vg1375wo/4 3/18/85

SOFTECH

NUMERIC PROPERTIES (CONT.)

- IN PRACTICE, INTERVAL ARITHMETIC IS TOO PESSIM~~E~~ISTIC
  - HARDWARE USUALLY PROVIDES MORE PRECISION
  - ERRORS TYPICALLY CANCEL
- RULES OF Ada DO NOT JUSTIFY A STRONGER METHOD OF REASONING ABOUT REAL ARITHMETIC
- VERIFICATION ESTABLISHES NUMERIC PORTABILITY
  - IT MAY RARELY BE POSSIBLE TO ESTABLISH NUMERIC PORTABILITY WITHIN TIME, SPACE, AND ACCURACY CONSTRAINTS

VG1375wo/3 3/18/83

SOFTECH

ABSENCE OF UNANTICIPATED EXCEPTIONS

- SIMILAR TO GERMAN'S RUNCHECK VERIFIER FOR ABSENCE OF RUNTIME ERRORS IN PASCAL
- IN Ada, RUNTIME ERRORS RAISE EXCEPTIONS
- DIFFERENCES BETWEEN PASCAL AND Ada:
  - Ada PROVIDES HANDLERS
  - SOME Ada COMPONENTS INTENTIONALLY PROPAGATE EXCEPTIONS
  - SOME UNPREVENTABLE CONDITIONS MANIFEST THEMSELVES ONLY THROUGH EXCEPTIONS LIKE STORAGE_ERROR, NAME_ERROR, DEVICE_ERROR
- THEREFORE, WE ARE ONLY INTERESTED IN ABSENCE OF <u>UNANTICIPATED</u> EXCEPTIONS

SOFTECH

vg1375wo/6 3/18/85

ABSENCE OF UNANTICIPATED EXCEPTIONS (CONT.)

- BENEFITS:
  - VALIDATE PROGRAM LOGIC
  - PROVE THAT BOUNDARY CONDITIONS AND UNPREDICTABLE EVENTS HAVEN'T BEEN OVERLOOKED
  - FORCE EXPLICIT DOCUMENTATION OF IMPLICIT ASSUMPTIONS
  - RECONCILE RELIABILITY AND EFFICIENCY BY "CERTIFYING" USES OF THE SUPPRESS PRAGMA
  - SPECIFICATIONS CAN BE AS SIMPLE AS A LIST OF ANTICIPATED EXCEPTIONS

vg1375wo/7 3/18/85

SOFTECH

ABSENCE OF ERRONEOUS EXECUTION

- CERTAIN RULES HARD TO ENFORCE AT COMPILE TIME OR RUN TIME
    - AN IMPLEMENTATION NEED NOT ENFORCE THEM
    - COMPILERS MAY ASSUME WHEN GENERATING CODE THAT THE RULES ARE OBEYED
    - VIOLATION OF THE RULES IS <u>ERROUNEOUS</u> <u>EXECUTION</u>, AND THE EFFECT IS UNPREDICTABLE
- AN IDEAL APPLICATION FOR VERIFICATION
    - PROOFS CAN WORK WHERE TRADITIONAL COMPILE-TIME CHECKS ARE TOO WEAK AND RUN-TIME CHECKS ARE TOO EXPENSIVE
    - CLOSES A MAJOR GAP IN Ada CHECKING
- NO PROPERTY IS REALLY PROVEN UNTIL ERRONEOUS EXECUTION IS PROVEN IMPOSSIBLE

VG1373wo/8 3/18/85

SOFTECH

FORMS OF ERRONEOUSNESS

- FORMS WHOSE ABSENCE MAY BE PROVABLE:
  - OCCURENCE OF AN EXCEPTION FOR WHICH CHECKS HAVE BEEN SUPPRESSED
  - EVALUATION OF UNINITIALIZED SCALAR VARIABLES
  - REFERENCE TO A DEALLOCATED VARIABLE
  - CHANGING A DISCRIMINANT VALUE AS A SIDE EFFECT WHILE ASSIGNING TO A DEPENDENT COMPONENT
- FORMS WHOSE ABSENCE CAN BE GUARANTEED BY IMPOSING ADDITIONAL RESTRICTIONS:
  - DEPDENCE ON A PARAMETER-PASSING MECHANISM
    (RESTRICTION: NO ALIASING)
  - UNCHECKED CONVERSION THAT VIOLATES PROPERTIES OF THE TARGET TYPE
    (RESTRICTION: NO UNCHECKED CONVERSION)
  - UNSYNCHRONIZED USE OF SHARED VARIABLES
    (RESTRICTION: NO SHARED VARIABLES)
  - ADDRESS CLAUSES THAT SPECIFY OVERLAYS
    (RESTRICTION: NO ADDRESS CLAUSES)

SOFTECH

vg1375wo/9 3/18/85

Appendix W

ANNA

Friedreich von Henke
SRI, Inc.

Friedrich von Henke: ANNA

The talk discusses some finer points and design problems of Anna.

Example 1:

```
     (generic)
          type ITEM is private;
          ZERO: in ITEM;
          (with) function "+"(X,Y: ITEM) return ITEM is <>;
       --| axiom
       --| for all U,V,W : ITEM =>
       --|            ZERO + V = V,
       --|            U + ZERO = U,
       --|         (U + V) + W = U + (V + W);
          type VECTOR is array (POSITIVE range <>) of ITEM;

     package ON_VECTORS is
             ...
          function SUM (A,B: VECTOR) return VECTOR;
       --|   where return C: VECTOR =>
       --|      for all I: INTEGER range A'RANGE =>
       --|            C(I) = A(I) + B(I),
       --|      ... ;
             ...
     end ON_VECTORS;
```

In this example, what is being passed is something like a package, or, in the terminology that Joe Goguen is using, something like a theory or a view of a structure.

We want to have a handle on the sort of parameterization that is possible on generic packages. One would really like to write something like:

```
--|  left-zero (zero, +),  right-zero (zero, +),
--|  associative (+);
```

to have some sort of higher order predicates; and actually we would like to go one step further and say:

```
--|  monoid (ITEM, +, zero);
```

However, in ANNA we do not have the facility for saying that.

There are certain limitations imposed on ANNA, which come from the fact that one of the basic design decisions underlying ANNA is that ANNA should take an ADA program, leave it basically untouched, and add to it certain things in the form of formal comments. These formal comments are subject to rules that are basically derivatives of the ADA rules, and furthermore the specification constructs used in annotations are defined as ADA entities. For example, a function used in annotations will be defined as a virtual ADA function. In that way we gain a certain expressiveness that ADA lacks. However, this strict adherence to ADA syntax and semantics has certain drawbacks.

There is a problem of consistency of annotation illustrated by:

Example 2:

```
subtype SMALL_EVEN is INTEGER range 1 .. 100;
--|  where X: SMALL_EVEN => even(X);
```

Here the statement "3 is SMALL_EVEN" is true in ADA and false in ANNA; so we call this an ANNA error.

Another point is the treatment of equality. The Reference Manual says that equality can be explicitly defined only for limited types. We were happy to follow that rule in the design of ANNA until John Goodenough, in a note in a recent issue of ADA Letters, showed that a way can be found of defining equality for any given type. Designers of annotation languages will wonder uneasily how many other semantic bombshells still lie hidden in the programming language.

Another kind of dependence that comes up with equality is that from within the package you can refer to anything that is visible, so some equality relationship that used to be true suddenly may not be true. Fortunately ANNA would supply an appropriate default axiom that suddenly pops up if you have not thought of it.

These examples show some of the ramifications that one has to take into account when one tries to build up something that looks like a mathematical system within the context where you have the freedom that ADA provides.

Appendix X

ANNA Tools

David Luckham
Stanford University

David Luckham: ANNA Tools

I hope you got the idea from Friedrich von Henke that it is not very easy to design a language extension of ADA that is fairly consistent in its semantics and allows you to express things that you cannot say in ADA itself.

That is one of the reasons why we have no tasking specification in ANNA at present. If you ask what you would like to specify about ADA tasking before you try to write the tasks themselves, you find that the first thing you would like to do is to throw away the visibility rules and the linear elaboration of your specifications.

The attempt to extend ADA with tasking specifications is very much a more difficult thing than what we have done with ANNA which is to try just the linear sequential part of the language.

By the way we do have a task specification language, called TSL for task sequencing language, though we do not claim that it is any more than experimental.

We are at the stage where we are able to write package specifications and present them as a Parnas negotiation document. Perhaps the first family of ANNA tools that we might like to implement are things that would support this kind of negotiating before we get into much harder things like verification systems.

Our role is to encourage the use of formal specifications in the development and maintenance of correct ADA programs.

So the first thing that you can do for at least a subset of ANNA is to produce a preprocessor to a compiler, which will accept an ANNA program, and translate the annotations into ADA runtime checks, so that out the back end will come a kosher, legitimate ADA program.

Things are not that simple, because of the naming and renaming, and hiding and local scoping conventions, and the fact that variables when declared must be constrained, and so on. You have to bend over backwards to get a really good implemenation of this transformation process, and it is being worked on by various of the students currently at Stanford.

The result might be called a self-checking program. At least it is going to raise ANNA errors, if in the course of running there is an inconsistency between the ANNA annotations and the underlying ADA program. Now the ANNA annotations may of course involve execution of virtual code.

So now what can you use this for? You can run an ADA program in comparison with its formal ANNA specification. This has some obvious **possible** applications: you get a lot of runtime checks because the annotation mechanisms are powerful, and you can say a lot because they are scoped. You may be able to use it for test and debug. You may even be able to use it for permanent runtime checks in self-checking programs.

A similar kind of thing gives you more power in a hardware design language: it gives you comparative simulation of two different representations of the same piece of hardware. So we are trying to apply the same ideas to the VHDL-like language.

That is the main tool we are building. It is working quite well and is a monument to the portability of well-written ADA code. It also enables us to test the compilers and find bugs in them. There are some other tools that we are also developing, like PROLOG interpreters in ADA.

You can get comparative testing with a very simple ANNA trick: use your old program as the specification for the new version of your program: then do a comparison run of the new against the old.

We would like an optimizer for the run time checks, and certainly that optimizer has got to do a bit of reasoning, probably of the PROLOG variety. There are preliminary papers on how to parallelise the checking of these annotations.

To support the Parnassian model of the development process by negotiation and redocumentation you would like to be able to analyse your specification and to automate that analysis so you could ask intelligent questions of the specifications.

And lastly, of course, there is something we haven't addressed anywhere else at all. If we were to use ANNA for testing, we would probably need a number of standard packages, especially if we got into the testing of the timing parts of programs. However, the ANNA checking interferes with timing.

We have not tried any full-scale verifier, we don't plan to for a while, as we have not yet recovered from the PASCAL verifier. But we invite people to send one-page ADA programs, accompanied by descriptions of their purpose in English, to LUCKHAM@SAIL for annotation.

DISTRIBUTION LIST FOR M-146

	Bernard Abrams Grumman Aerospace Corporation Mail Station 001-31T Bethpage, NY 11714 (516) 575-9487	ABRAMS@USC-ECLB
*	Omar Ahmed Verdix Corporation 7655 Old Springhouse Road McLean, VA 22102 (703) 448-1980	
*	Eric R. Anderson TRW DSG (R2/1134) One Space Park Redondo Beach, CA 90278 (213) 535-5776	TRWRB!TRWSPP!ERA@BERKELEY
*	Dr. Thomas C. Antognini MITRE Corporation Mailstop B330 Burlington Road Bedford, MA 01730 (617) 271-7294	SECURITY!TCA@MITRE-BEDFORD or TCVB@MITRE-BEDFORD
	Charles Applebaum 1058 Boyurgogne Bowling Green, OH 43402 (419) 352-0777	CHA@MITRE-BEDFORD
	Krzystof Apt Thomas J. Watson Research Center P. O. Box 218 88-K01 Route 134 Yorktown Heights, NY 10598 (914) 945-2923	
	Terry Arnold Merdan Group P.O. Box 17098 San Diego, CA 92117	MERDAN@ISI
	Ted Baker Department of Computer Science Florida State University Tallahassee, FL 32306 (904) 644-2296	

David Elliot Bell   DBELL@MIT-MULTICS
Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, MD 21738
(301) 854-5889

Dan Berry
3531G Boelter Hall
Computer Science Department
School of Eng. and Appl. Science
Los Angeles, CA 90024
(213) 825-2971

Edward K. Blum   BLUM@ECLB
Mathematics Department
University of Southern California
Los Angelos, CA 90089
(213) 743-2504

* Alton L. Brintzenhoff   SCI-ADA@USC-ISI
SYSCON Corporation
3990 Sherman Street
San Diego, CA 92110
(619) 296-0085

* Dr. Dianne Britton   HELBIG@ISI
RCA Adv. Tech. Labs
ATL Building
Moorestown Corporate Center
Moorestown, NJ 08057
(609) 866-6654 or (609) 924-3253

* Dr. R. Leonard Brown   BROWN@AEROSPACE
M1/112
The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009
(213) 615-4335

Richard Chan   RCHAN@USC-ECL (bad)
Hughes Aircraft Co.
P. O. Box 33
FU-618/P115
Fullerton, CA 92634
(714) 732-7659

* Norman Cohen   NCOHEN@ECLB
SofTech, Inc.
705 Masons Mill Business Park
1800 Byberry Road
Huntingdon Valley, PA 19006
(215) 947-8880

* Paul M. Cohen PCOHEN@ECLB
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

Richard M. Cohen COHEN@UTEXAS-20
Institute for Computing Science
2100 Main Bldg.
University of Texas
Austin, Texas 78712
(512) 471-1901

Michael D. Colgate FREEMAN@FORD-COS1
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908

* Mark R. Cornwell CORNWELL@NRL-CSS
Code 7590
Naval Research Lab
Washington, D.C. 20375
(202) 767-3365

Major Terry Courtwright COURT@MITRE
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5056

* Dan Craigen CMP.CRAIGEN@UTEXAS-20
c/o I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

Steve Crocker, M1-101 CROCKER@AEROSPACE
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 92957
(213) 648-6991

John J. Daly WCOXTON@USADHQ2
USAISSAA
2461 Eisenhower Avenue
Alexandria, VA 22331-0700

Tom Dee
Boeing Commercial Airplane Co.
P. O. Box 3707
MS 77-21
Seattle, WA 98124
(206) 237-0194

Jeff Facemire FACEMIRE%TI-EG@CSNET-RELAY
Texas Instruments
P.O. Box 801
M/S 8007
2501 West University
McKinney, TX 75069
(214) 952-2137

* John C. Faust FAUST@RADC-MULTICS
RADC/COTC
Griffiss AFB, NY 13441
(315) 330-3241

Gerry Fisher
IBM Research 35-162
P. O. Box 218
Yorktown Heights, NY 10598
(914) 945-1677

Roy S. Freedman FREEDMAN@ECLB
Hazeltine Corporation
Greenlawn, NY 11740
(516) 261-7000

James W. Freeman
Ford Aerospace & Comm. Corp.
Mailstop 15A
10440 State Highway 83
Colorado Springs, CO 80908
(303) 594-1536

Mark Gerhardt MSG@MITRE-BEDFORD
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7839

Chuck Gerson
Boeing Aerospace Co.
Mailstop 8H-56
P.O. Box 3999
Seattle, WA 98124

Helen Gill
MITRE
Mailstop W459
1820 Dolly Madison Boulevard
McLean, Virginia 22102
(703) 883-7980

Kathleen A. Gilroy
Software Prod. Solutions, Inc.
P. O. Box 361697
Melbourne, FL 32936

Virgil Gligor
Department of Electrical Engineering
University of Maryland
College Park, Maryland 20742
(301) 454-8846

Donald I. Good GOOD@UTEXAS-20
2100 Main Building
The University of Texas at Austin
Austin, TX 78712
(512) 471-1901

Ronald A. Gove GOVE@MIT-MULTICS
Booz, Allen & Hamilton
4330 East West Highway
Bethesda, MD 20814
(301) 951-4624

* Inara Gravitis GRAVITIS@ECLB
SAIC
1710 Goodridge Drive
McLean, VA 22202
(703) 734-4096 or (202) 697-3749

* Col. Joseph S. Greene, Jr. JGREENE@USC-ISI
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6818

David Gries GRIES@CORNELL
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
(607) 256-4052

David Guaspari RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* J. Daniel Halpern SYTEK Corp. 1225 Charleston Road Mountain View, CA 94043 (415) 966-7300	SYTEK@SRI-UNIX or MENLO70!SYTEK!DAN@BERKELEY
* Kurt W. Hansen Dansk Datamatik Center LuudToftevej 1C DK2800 Lyngby Denmark PHONE: ++ 45 2 872622	KHANSEN@ECLB
* Scott Hansohn Honeywell Secure Comp. Tech. Center Suite 130 2855 Anthony Lane South St. Anthony, MN 55418 (612) 379-6434	HANSOHN@HI-MULTICS
* Larry Hatch DoD Computer Security Center 9800 Savage Road Fort Meade, MD 20755-6000 (301) 859-6790	HATCH@TYCHO
Linn Hatch IBM 17100 Frederick Heights Gaithersburg, MD 20879	
* Brian E. Holland DoDCSC, C3 9800 Savage Road Fort Meade, MD 20755-6000 (301) 859-6968	BRIAN@TYCHO
Ray Hookway Dept. of Computer Eng. & Science Case Institute of Technology Case Western Reserve University Cleveland, OH 44106 (216) 368-2800	HOOKWAY%CASE@CSNET-RELAY
Paul Hubbard Dept. of Computer Eng. & Science Case Institute of Technology Case Western Reserve University Cleveland, OH 44106 (216) 368-2800	HOOKWAY%CASE@CSNET-RELAY

Jim Huitema
National Security Agency
R831
Ft. Meade, MD 20755
(301) 859-6921

Larry A. Johnson — LJOHNSON@MIT-MULTICS
GTE
77 "A" Street
Needham, MA 02194
(617) 449-2000 ext. 3248

* Juern Juergens — JJURGENS@ECLB
SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254
(617) 890-6900 ext. 316

Matt Kaufmann — CMP.BARC@UTEXAS-20
Burroughs Corp.
Austin Research Center
12201 Technology Blvd.
Austin, TX 78727
(512) 258-2495

Prof. Richard A. Kemmerer — DICK@UCLA-CS
Computer Science Department
University of California
Santa Barbara, CA 93106
(805) 961-4232

John C. Knight — UVACS!JCK@SEISMO
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
(804) 924-1030

Major Al Kopp — AKOPP@ECLB
Ada Joint Program Office
OUSDRE/R&AT
Pentagon Room 3D139 (Fern Street)
Washington, DC 20301-3081
(202) 694-0211

* Thomas M. Kraly
IBM Federal Systems Division
Software Eng. & Tech. 4D08
6600 Rockledge Drive
Bethesda, MD 20817
(301) 493-1449

Dr. Jack Kramer KRAMER@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2263

Eduardo Krell
3804 Boelter Hall
UCLA
Los Angeles, CA 90024

Kathy Kucheravy
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

Dr. Kenneth Kung KKUNG@USC-ECLA
Hughes Aircraft Company
Ground Systems Group
M. S. 618/Q315
P. O. Box 3310
Fullerton, CA 92634
(714) 732-0262

* Carl Landwehr LANDWEHR@NRL-CSS
Code 7593
Naval Research Laboratory
Washington, DC 20375-5000
(202) 767-3381

* Mike Lake MLAKE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Division
Alexandria, VA 22311
(703) 845-2519

Randall E. Leonard
Army Sys. Software Support Command
ATTN: ASB-QAA
Fort Belvoir, VA 22060

Nancy Leveson
ICS Department
University of California
Irvine, CA 92717
(714) 548-7525 or (714) 856-5517

Dr. Timothy E. Lindquist LINDQUIS%ASU.CSNET@CSNET-RELAY
Computer Science Department
Arizona State University
Tempe, AZ 85287
(602) 965-2783

* Steven Litvintchouk SDL@MITRE-BEDFORD
Mail Stop A180T
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7753

* David Luckham LUCKHAM@SAIL
Stanford University
Computer Systems Lab, ERL 456
Stanford, CA 94305
(415) 497-1242

Dr. Glenn MacEwen
Computing and Information Science
Goodwin Hall
Queens University
Kingston, Ontario
K7L 3N6
(613) 547-2915 or (613) 548-4355

* John McHugh MCHUGH@UTEXAS-20
Research Triangle Institute
Box 12194
Research Triangle Park, NC 27709
(919) 541-7327

* Ann Marmor-Squires MARMOR@ISI
TRW
Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031
(703) 876-8170

Eric Marshall PAYTON@BBNG
System Development Corporation
P.O. Box 517
Paoli, PA 19301
(215) 648-7223

* Adrian R. D. Mathias RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* Terry Mayfield TMAYFIELD@ECLB
Institute for Defense Analyses
Computer & Software Division
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2479

	Rudolf W. Meijer Commission of the European Communities Info. Tech. and Telecomm. Task Force A25 9/6A Rue de la Loi 200 B-1049 Brussels, Belgium PHONE: +32 2 235 7769	RMEIJER@USC-ECLB
*	Donn Milton Verdix Corporation 7655 Old Springhouse Road McLean, VA 22102 (703) 448-1980	VRDXHQ!DRM1@SEISMO
*	Warren Monroe Hughes Aircraft Co. P.O. Box 3310 FU-618/Q315 Fullerton, CA 92634 (714) 732-2887	WMONROE@ECLA
	Mark Moriconi SRI International Computer Science Laboratory 333 Ravenswood Avenue Menlo Park, CA 94025 (415) 859-5364	MORICONI@SRI-CSL
*	LCDR Philip A. Myers Space and Naval Warfare Sys. Command SPAWAR 8141A Washington, DC 20363-5001 (202) 692-8484	MYERS@NRL-CSR
*	Karl Nyberg Verdix Corporation 7655 Old Springhouse Road McLean, VA 22102 (703) 448-1980	NYBERG@ECLB
*	Myron Obaranec U. S. Army, CECOM Fort Monmouth, NJ 07703 ATTN: AMSEL-TCS-SIO (201) 544-4962	LAKSHMI@CECOM-1
	Frank J. Oles Thomas J. Watson Research Center P.O. Box 218 88-K01 Route 134 Yorktown Heights, NY 10598 (914) 945-2012	

Mahmoud Parsian
SDI Inc.
P. O. Box 4283
Falls Church, VA 22044

Diana B. Parton   DBP@MITRE-BEDFORD
The MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7754

* Don Peters
Comm. Sec. Establishment
Dept. of Nat. Defence
101 Colonel By Drive
Ottawa KIA OK2 CANADA
(613) 998-4519

* John Peterson   PETERSON@TYCHO
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755
(301) 859-6790

* Joseph E. Pfauntsch, MS 29A   JEP@FORD-COS4
Ford Aerospace & Comm. Corp.
10440 State Highway 83
Colorado Springs, Colorado 80908
(303) 594-1326

* Richard Platek   RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Erhard Ploedereder   PLOEDEREDER@TARTAN
Tartan Labs
411 Melwood Avenue
Pittsburgh, PA 15213
(412) 621-2210

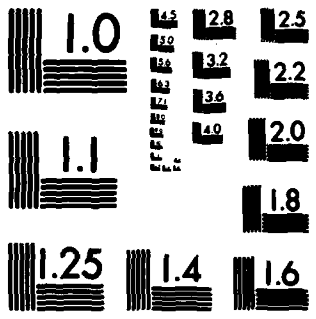
* David Preston   DPRESTON@ECLB
IITRI
5100 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711

Sri Rajeev   IHNP4!ATTUNIX!RAJEEV@BERKELEY
AT&T Bell Laboratories
Room 1-342
190 River Road
Summit, NJ 07901
(201) 522-6330

AD-A171 507 PROCEEDINGS OF THE IDA WORKSHOP ON FORMAL SPECIFICATION AND VERIFICATION O..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA C G ROBY DEC 85 IDA-M-146 5/5

UNCLASSIFIED IDA/HQ-85-30658 MDA903-84-C-0031 F/G 9/2 NL

END
DATE
FILMED
10-86


1.0
4.5
2.8
2.5
5.0
5.6
3.2
2.2
6.3
3.6
7.1
4.0
2.0
1.1
1.8
1.25
1.4
1.6

* William D. Ricker WDR@MITRE-BEDFORD
The MITRE Corporation
M/S K229
Burlington Road
Bedford, MA 01730
(617) 271-3001

Edmund Robinson
Peterhouse, Cambridge CB2 1RD
United Kingdom
(0223) 350256

* R. Max Robinson RROBINSON@USC-ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2097

W. A. Robison
30 Charles Street West
Apt. # 1811
Toronto, Ontario, CANADA
M4Y 1R5
(416) 925-0751

* Clyde G. Roby CROBY@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
Alexandria, VA 22311
(703) 845-2541

Ken Rowe
DoD Computer Security Center
9800 Savage Road
Ft. Meade, MD 20755

John Rushby - EL393 RUSHBY@SRI-CSL
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5456

* Mark Saaltink SAALTINK@MIT-MULTICS
I. P. Sharp Associates
265 Carling Avenue
Suite 600
Ottawa, Ontario, Canada K1S 2E1
(613) 236-9942

Marvin Schaefer SCHAEFER@USC-ISI
DoD Computer Security Center
9800 Savage Road
Fort Meade, MD 20755-6000
(301) 859-6880 or (301) 859-6818

* Mike Schwartz UCBVAX!HPLABS!HAO!DENELCOR!
Mailstop L0402
Martin-Marietta
Denver Aerospace
P. O. Box 179
Denver, CO 80201
(303) 977-0421

Dev Sen
STC IDEC LIMITED
Technology Division
Six Hills House
London Road
Stevenage
Hertfordshire S61 1YB ENGLAND
PHONE: 011-44-438-726161

* Jerry Shelton VRDXHQ!JHS@SEISMO
Verdix Corporation
7655 Old Springhouse Road
McLean, VA 22102
(703) 448-1980

* Brian Siritzky (212) 460-7239 SIRITZKY@NYU-ACF2 or
Dept. of Computer Science ...CMCL2!ACF2!SIRITZKY
Courant Institue of Math. Sciences
New York University
251 Mercer Street
New York, NY 10012

* Roger Smeaton SMEATON@NOSC-TECR
NOSC, Code 423
San Diego, CA 92152
(619) 225-2083

Michael Smith MKSMITH@UTEXAS
ICSCA
2100 Main Building
University of Texas
Austin, TX 78712
(512) 471-1901

* Ryan Stansifer RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* David Sutherland RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

Steve Sutkowski INCO@USC-ISID
Inco Inc.
8260 Greensboro Drive
McLean, VA 22102
(703) 883-4933

Michael Thompson
Astronautics Corporation of America
P. O. Box 523
Milwaukee, Wisconsin 53201-0523
(414) 447-8200

* Friedrich von Henke VONHENKE@SRI-CSL
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-2560

Barry Watson WATSON@ECLB
Ada Information Clearinghouse
IITRI
Room 3D139 (1211 Fern St., C-107)
The Pentagon
Washington, DC 20301
(703) 685-1477

Doug Weber RPLATEK@ECLB
Odyssey Research Associates
408 East State Street
Ithaca, NY 14850
(607) 277-2020

* Steve Welke SWELKE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2393

Col. William Whitaker WWHITAKER@ECLB
WIS/JPMO/ADT
7726 Old Springhouse Road
Washington, DC 20330-6600
(202) 285-5065

* Jim Williams JGW@MITRE-BEDFORD
MITRE Corporation
Mailstop B332
Burlington Road
Bedford, MA 01730
(617) 271-2647

Jim Wolfe JWOLFE@ECLB
Institute for Defense Analyses
Computer & Software Eng. Div.
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2109

Larry Yelowitz KLY@FORD-WDL1
Ford Aerospace and Comm. Corp.
Western Development Lab. Div.
Mailstop X-20
3939 Fabian Way
Palo Alto, CA 94303
(415) 852-4198

* Christine Youngblut CYOUNGBLUT@ECLB
Advanced Software Methods, Inc.
17021 Sioux Lane
Gaithersburg, MD 20878
(301) 948-1989

Francoise Youssefi
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

* Margie Zuk MMZ@MITRE-BEDFORD
Mailstop B321, Bldg B
MITRE Corporation
Burlington Road
Bedford, MA 01730
(617) 271-7590

Distribution List for IDA for M-146 (Final)

Sponsor

Ms. Virginia Castor (5 Copies)
Director
Ada Joint Program Office
1211 Fern St., Room C-107
Arlington, VA 22202

Other

Defense Technical Information Center (2 copies)
Cameron Station
Alexandria, VA 22314

DATE
ILMED
-8