AD-A252 477

2

RL-TR-92-16
Final Technical Report
February 1992

ROME LABORATORY

SPECIFICATION AND VERIFICATION OF SECURE CONCURRENT AND DISTRIBUTED SOFTWARE SYSTEMS

University of California

M. Archer, G. Fisher, K. Levitt, R. Olsson, J. Alves-Foss,
J. Buffenbarger, G. Fink, D. Frincke, D. Huang,
P. Windley

DTIC
ELECTE
JUN 23 1992
S A D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

92-16400

92 6 2

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-16 has been reviewed and is approved for publication.

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

FOR THE COMMANDER: John A Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3AB), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

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

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
February 1992

3. REPORT TYPE AND DATES COVERED
Final Jul 88 - Dec 89

4. TITLE AND SUBTITLE
SPECIFICATION AND VERIFICATION OF SECURE CONCURRENT AND DISTRIBUTED SOFTWARE SYSTEMS

5. FUNDING NUMBERS
C - F30602-88-D-0025
Task 13
PE - 35167G
PR - 1069
TA - 01
WU - P1

6. AUTHOR(S) M. Archer, G. Fisher, K. Levitt, R. Olsson, J. Alves-Foss, J. Buffenbarger, G. Fink, D. Frincke, D. Huang, P. Windley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California
Department of Electrical Engineering & Computer Science
Division of Computer Science
Davis CA 95616

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3AB)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-16

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Emilie J. Siarkiewicz/C3AB(315) 330-3241

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

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes an investigation of techniques to support the specification and verification of concurrent and distributed software systems, with special emphasis on issues of security. The investigation has focused on two major areas. The primary focus is a survey of existing methodologies and systems that are relevant to the specification and verification of concurrency. The secondary focus is on the initial design of a short-term workbench that embodies capabilities of existing systems together with new features that extend the current state of the art in the specification and verification of concurrency.

The introduction to the report summarizes survey results and presents overall conclusions about the current state of the art. Sections 2 and 3 of the report present the details of the methodology and system surveys respectively. The surveys include high level feature comparison tables accompanied by extended reviews. Section 4 describes a design for the short-term workbench that will support computer-aided specification and verification. Section 5 describes a set of extended examples that were developed to test the design ideas. Section 6 concludes with an overall summary and an overview of targets for future work.

14. SUBJECT TERMS
Distributed Processing, Verification, Specification, Multilevel Security

15. NUMBER OF PAGES
280

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

Contents

1 Introduction 1
- 1.1 Summary of Survey Results 1
- 1.2 Summary of Conclusions and Recommendations 3
- 1.3 Summary of Novel Research 3

2 Methodology Survey 4
- 2.1 Summary Comparison Table 4
- 2.2 Barringer and Mearns 8
  - 2.2.1 Overview 8
  - 2.2.2 Technical Discussion 8
  - 2.2.3 Critical Remarks 9
- 2.3 Chen and Hoare 9
  - 2.3.1 Overview 9
  - 2.3.2 Technical details 10
  - 2.3.3 Critical Remarks 11
- 2.4 Chen and Yeh 11
  - 2.4.1 Overview 11
  - 2.4.2 Technical Discussion 11
  - 2.4.3 Critical Remarks 12
- 2.5 Flon and Suzuki 12
  - 2.5.1 Overview 12
  - 2.5.2 Technical Discussion 13
  - 2.5.3 Critical Remarks 13
- 2.6 Halpern and Owicki 14
  - 2.6.1 Overview 14
  - 2.6.2 Technical Discussion 14
  - 2.6.3 Critical Remarks 15
- 2.7 Jones 15
  - 2.7.1 Overview 15
  - 2.7.2 Technical Discussion 15
  - 2.7.3 Critical Remarks 18
- 2.8 Lamport 18
  - 2.8.1 Overview 18
  - 2.8.2 Technical Discussion 19
  - 2.8.3 Critical Remarks 20
- 2.9 Misra and Chandy 20
  - 2.9.1 Overview 20
  - 2.9.2 Technical Discussion 21
  - 2.9.3 Critical Remarks 22

DTIC COPY INSPECTED

Accesion For
NTIS CRA&I ✓
DTIC TAB □
Unannounced □
Justification

By
Distribution /

Av

A-1

3 System Survey 23
3.1 Overview . . . 23
3.2 Summary Comparison Table . . . 23
3.3 Affirm . . . 25
3.3.1 Overview . . . 25
3.3.2 Execution and Rapid Prototype Support . . . 25
3.3.3 Abstraction Mechanisms . . . 26
3.3.4 Forms of Logic Supported . . . 27
3.3.5 Verification and Theorem Proving Support . . . 27
3.3.6 Specification Checking – Completeness and Soundness . . . 27
3.3.7 Formal system Basis – Completeness, Consistency and Soundness . . . 28
3.3.8 Qualitative Measures . . . 28
3.3.9 An Example . . . 28
3.3.10 Critical Remarks . . . 30
3.4 Enhanced HDM . . . 31
3.4.1 Overview . . . 31
3.4.2 The Enhanced HDM Methodology . . . 32
3.4.3 Revised Special Specification Language . . . 34
3.4.4 Program Verification in Enhanced HDM . . . 36
3.4.5 Hierarchical Development . . . 36
3.4.6 The Theorem Prover and How it is Used . . . 37
3.4.7 The MLS Tool . . . 38
3.4.8 The Implementation of Enhanced HDM . . . 39
3.4.9 Conclusions . . . 39
3.5 FASE . . . 41
3.5.1 Overview . . . 41
3.5.2 Execution and Rapid Prototype Support . . . 41
3.5.3 Abstraction Mechanisms . . . 42
3.5.4 Forms of Logic Supported . . . 42
3.5.5 Verification and Theorem Proving Supported . . . 42
3.5.6 Specification Checking—Completeness, Consistency, and Soundness . . . 42
3.5.7 Examples . . . 42
3.5.8 Critical Remarks . . . 44
3.6 HOL . . . 45
3.6.1 Overview . . . 45
3.6.2 The HOL Verification System . . . 45
3.6.3 Axiomatic Basis for HOL . . . 48
3.6.4 Critical Remarks . . . 56
3.7 OBJ3 . . . 63
3.7.1 Overview . . . 63
3.7.2 Abstraction Mechanisms . . . 64
3.7.3 Forms of Logic Supported . . . 64
3.7.4 Formal System Basis . . . 64
3.7.5 Specification Checking . . . 65
3.7.6 Execution and Rapid Prototype Support . . . 66

3.7.7 Verification and Theorem-Proving Support . . . 66
3.7.8 Low Water Mark Example . . . 66
3.7.9 Critical Remarks . . . 71
3.8 EVES . . . 78
3.8.1 Overview . . . 78
3.8.2 Execution and Rapid Prototype Support . . . 78
3.8.3 Abstraction Mechanisms . . . 78
3.8.4 Forms of Logic Supported . . . 78
3.8.5 Verification and Theorem Proving Supported . . . 79
3.8.6 Specification Checking—Completeness, Consistency, and Soundness . . . 79
3.8.7 Formal System Basis—Completeness, Consistency, and Soundness . . . 79
3.8.8 Examples . . . 79
3.8.9 Critical Remarks . . . 81
3.9 Gypsy . . . 83
3.9.1 Overview . . . 83
3.9.2 Execution and Rapid Prototype Support . . . 83
3.9.3 Abstraction Mechanisms . . . 83
3.9.4 Forms of Logic Supported . . . 83
3.9.5 Verification and Theorem Proving Support . . . 83
3.9.6 Specification Checking . . . 84
3.9.7 Support and/or Adaptability for Concurrency . . . 84
3.10 Nuprl . . . 85
3.10.1 Overview . . . 85
3.10.2 Execution and Rapid Prototype Support . . . 85
3.10.3 Abstraction Mechanisms . . . 85
3.10.4 Forms of Logic Supported . . . 85
3.10.5 Verification and Theorem Proving Supported . . . 86
3.10.6 Specification Checking—Completeness, Consistency, and Soundness . . . 86
3.10.7 Formal System Basis—Completeness, Consistency, and Soundness . . . 86
3.10.8 Examples . . . 86
3.10.9 Critical Remarks . . . 87
3.11 VDM . . . 90

4 Design of a Prototype Short-Term Workbench 91
4.1 Overview . . . 91
4.2 The Role of the FASE, HOL, and OBJ3 High-Level Specification Languages . . . 94
4.3 The Role of the SR Programming Language . . . 94
4.4 The Role of the Annotated-SR Verification Tool . . . 95
4.5 The Role of TED/Treemacs Theorem Proving Support Systems . . . 95

5 Algebraic Specification and Verification of Concurrency in OBJ 97
5.1 Overview of the Approach . . . 97
5.2 The OBJ Language . . . 98
5.3 The Readers/Writers Problem . . . 101
5.4 Star Operations . . . 101
5.5 Specification of Readers/Writers . . . 103

5.6 Verification of the Readers/Writers Invariant 106
5.6.1 Construction of the Proof Object 108
5.6.2 Construction of the Invariant 114
5.6.3 Induction Scheme 114
5.7 Explicitly Concurrent Specification 119
5.7.1 Message-Passing Extensions 119
5.7.2 Message-Passing Protocol 120
5.7.3 Centralized Solutions 123
5.7.4 Examples 124
5.8 Verification of Implementations 126
5.8.1 The Symbol Table Problem 129
5.8.2 Specification of the Symbol Table 129
5.8.3 Implementation of the Symbol Table 129
5.8.4 Verification of the Symbol-Table Implementation 132
5.9 Conclusions and Future Work 138
5.10 Proof Tools 140

6 Verification of Security in OBJ 143
6.1 Introduction 143
6.1.1 Using OBJ3 to Verify Security 143
6.1.2 General Observations 144
6.2 An OBJ3 Specification of A Simple Operating System 144
6.2.1 Architecture 144
6.2.2 Supervisor 146
6.2.3 User Processes 151
6.2.4 Conclusion 153
6.3 An alternate security model 153

7 Specification and Testing of Security in FASE 155
7.1 Introduction 155
7.2 A generic specification for a secure resource manager 156
7.2.1 Writing final algebra specifications: the methodology 156
7.2.2 Structure of the generic SRM specification 158
7.2.3 Specializing the template SRM specification 159
7.3 Running the specification: the Millen example 165
7.4 Advantages of the FASE system 165
7.5 The PLANNER 166
7.5.1 Detecting flows 166
7.5.2 Detailed description 167
7.5.3 Planner input vs. specification input 171
7.5.4 Examples 171
7.6 Conclusions and future directions 176
7.7 Extended Examples 178
7.7.1 Description of Millen system operations 178
7.7.2 Specification of the Millen example 180
7.8 High-level description of simple PLANNER 204

8 Axiomatic Verification of Concurrency in SR **206**
8.1 The Syntax of Annotated SR 207
8.2 The Axiomatic Semantics of SR 211

9 Working Examples and Models 220
9.1 A Secure Network Mail System Model 220
9.1.1 Introduction 220
9.1.2 Overall Description 220
9.1.3 The Model 222
9.1.4 Security Specification 223
9.1.5 Discussion 224
9.1.6 Conclusions 226
9.2 Initial Results on Specifying SDOS 227
9.2.1 High-Level Outline of an OBJC Specification of SDOS 228
9.2.2 Verifying Security Properties of the OBJC Specification of SDOS 230
9.3 A Security Kernel for Distributed Systems 234
9.3.1 Overview of Previous Research 234
9.3.2 Our Kernel 240
9.3.3 Summary 244

10 Conclusions 246

1 Introduction

This report describes an investigation of techniques to support the specification and verification of concurrent and distributed software systems, with special emphasis on issues of security. The investigation has focused on two major areas. The primary focus is a survey of existing methodologies and systems that are relevant to the specification and verification of concurrency. The secondary focus is on the initial design of a short-term workbench that embodies capabilities of existing systems together with new features that extend the current state of the art in the specification and verification of concurrency.

The remainder of the introduction to the report summarizes our survey results and presents our overall conclusions about the current state of the art. Sections 2 and 3 of the report present the details of the methodology and system surveys respectively. The surveys include high-level feature comparison tables accompanied by extended reviews. Section 4 describes our design for the short-term workbench that will support computer-aided specification and verification. Section 5 describes a set of extended examples that we have developed to test our design ideas. Section 6 concludes with an overall summary and an overview of targets for future work.

1.1 Summary of Survey Results

In the initial months of the project, we reviewed approximately 200 abstracts of papers in the following topic areas:

- Specification languages, for both sequential and concurrent programs
- Program verification methodologies and systems, for both sequential and concurrent programs
- Programming languages for concurrent and distributed programming
- Software environment support for specification and verification

From the 200 abstracts, we selected approximately thirty papers for detailed review and critique. Those papers selected for detailed review included work from the most visible and productive researchers in the field, as well as other representative work. From the members of the project team, we selected one or two individuals to be specialists in particular areas. The specialists thoroughly reviewed each of the papers in their respective areas and prepared a formal presentation on the results of the review to the rest of the project group.

In conjunction with the literature review, we have reviewed approximately two dozen systems that have features to support computer-aided specification and/or verification. Of these we have gained actual hands-on experience with ten. The systems selected for review have one or more of the following features:

- support for a formal specification language in the form of parsing, type checking, and possibly other forms of language analysis
- support for execution of formal specifications
- support for computer-aided verification
- software engineering support for specification and verification

Based on the results of the survey, we have compiled two "short lists" of methodologies and systems that we believe represent the fundamentally important concepts necessary for the tasks of specification and verification of concurrency, with special emphasis on security properties. The methodologies and systems chosen for inclusion in the short lists comprise a broadly representative set, covering the major researchers working in the area.

The methodology short list includes of the work of eight of the major research groups in the fields of specification and verification, with emphasis on concurrency. The methodologies are listed below by their associated authors with a brief synopsis of the key features:

- **Barringer and Mearns** – important technical concepts for the specification of concurrency
- **Chen and Hoare** – foundations for specification and verification of concurrency in the CSP language
- **Chen and Yeh** – foundations for event-based specification of concurrency
- **Flon and Suzuki** – important work on specification of concurrency via transformation to non-concurrent representations
- **Halpern and Owicki** – widely referenced work on modular specification and verification of concurrency based on computation histories
- **Jones** – important concepts in process-oriented specification of concurrency
- **Lamport** – foundational work on temporal logic and other basic concepts of concurrency
- **Misra and Chandy** – well-recognized work on very high-level specification of concurrency

The system short list consists of six of the most representative tools for computer-aided specification and verification:

- **Affirm** – provides basic support for algebraic specification
- **EHDM** – wide-range of support for specification and verification including software engineering support
- **FASE** – strong support for executable specification
- **HOL** – powerful and general computer-aided verification system
- **Larch** – provides basic framework for two-tiered approach to specification adopted in the design of our prototype workbench
- **OBJ3** – strong support for execution of specifications and extendible so support high-level specification of concurrency

The rationale for the selection of both of these short lists is based on a number of factors. These factors are:

- *Completeness* – those chosen comprise a complete and representative set, with no major methodology concepts or system capabilities not covered by at least one member on the lists
- *Suitability for the task at hand* – the systems should be relevant or potentially relevant to the specification of concurrency and security
- *System modifiability and stability* – for the listed systems, the source code for the systems must be public and readily available if the systems are to be modified to support concurrency
- *Familiarity* – when all other factors are equal, we prefer methodologies and systems of which we have working knowledge and experience

Based on these factors, we believe that our lists include a representative sample of methodologies and systems that afford us full coverage on the state of current research and development. The lists also form the basis for the ideas that will be embodied in the design of the short-term workbench.

1.2 Summary of Conclusions and Recommendations

Rather than wait until the end of a lengthy report to present our overall conclusions, we summarize them here based on the introduction presented thus far. We present a more technically oriented review of our conclusions in the final section of the report. The conclusions are:

- Many methodologies exist for concurrent specification and verification, including several that focus on security.
- Many systems exist that support specification and verification for *sequential* programs.
- Very few systems exist that support *concurrent* specification and verification, and certainly no single complete system exists.
- We need a discriminated union of the different features in the methodologies and systems on our short lists in order to provide a single complete system.
- Very few of the methodologies and/or systems address verification of both specification and code; we believe that code-level verification is crucial for the honest verification of security and we plan to address this issue in our workbench design.
- The design of our short-term workbench will be built on existing ideas and systems, not from scratch; this includes the reuse of existing code from one or more of the systems.

1.3 Summary of Novel Research

As we have completed the surveys, we have found that there is a large amount of novel research to be done in the areas concurrent specification and verification. In the design of our prototype workbench, we have begun some of this research, specifically in the following areas:

- The algebraic specification and verification of concurrency
- The algebraic specification of security properties
- The axiomatic semantics of the SR concurrent programming language, and its use in a methodology that will verify both specification-level as well as code-level security properties
- A model for a Secure Resource Manager that will provide the platform for high-level specification and testing of security properties
- The specification of a secure network mail handler
- A proposal for a secure distributed operating system kernel

Our research has resulted in a number of papers submitted for publication which we hope to see in print in the very near future.

2 Methodology Survey

This section of the report presents the specific details of the methodology survey that we have conducted. The survey is organized by the "short list" of methodologies presented in the previous section. This list contains eight groups of authors whose work we believe to be the most significant and representative of a larger body of relevant authors. Following the tabular comparison are extended reviews for the key authors summarized in the table.

2.1 Summary Comparison Table

Table 1 summarizes the results of the methodology survey. The table is formated with key features listed in the leftmost column, and the reviewed methodologies listed along the top row. The features are organized into seven major categories. Each of these categories is described briefly belo⁻ In the subsections which follow, detailed reviews of the summarized work are presented.

In Table 1, the features under the category "Models of Concurrency Supported" refer to the major forms of concurrency supported in specification and programming languages. These features are the union of those features used to describe concurrent execution in the current literature. "Concurrency Properties" refer to attributes of a concurrent program that should be specifiable and verifiable. Again, these features are drawn from the current literature, in particular [AS83]. The feature for "Granularity of Concurrency" refers to the lexical level of a program at which concurrency is specifiable; it should be noted that some languages support specification at more than one level. "Development Phases Supported" refers to the standard phases of the software life cycle that are supported by the systems under review. The "Forms of Logic" category includes the most common forms supported in computer-aided verification systems. HOL (for Higher-Order Logic) and Boyer-Moore (for the inventors' names) are logics particular to two computer-based logic systems. The "Quantitative" and "Qualitative Measures" categories include general features of the usage and design of the methodologies. The measures are based on our actual experience with the methodologies together with experience reported in the literature. Finally, the "Systems Features" category relates to those methodologies for which implemented system support is available; this category is not applicable to those methodologies that are not yet implemented.

	Barr & Mearns	Chen & Hoare	Chen & Yeh	Flon & Suzuki	Goguen	Jones	Lamport	Luckham	Misra & Chandy	Owicki
Models of Concurrency Supported:			[4]	[5]	[8]	[9]				
Rendesvous	X							X		
Shared Vars						X	X	X	X	
Sync Message Passing		X						X	X	X[9]
Async Message Passing			X					[12]		
RPC								[12]		
Concurrency Properties Specifiable and/or Verifiable:					[10]	[10]				
Deadlock	X	X	?	X			X	X		
Fairness		X	?				X	X		
Liveness		X	X	X				X		
Granularity of Concurrency:										
module				X			X	X[13]		
process			X[2]			X			X	
function					X[7]	X				X
statement	X	X	X[2]		X[7]		X	X		
expression										
Development Phases Supported:										
Specification		X	X		X	X	X	X	X	
Design		X	X			X		X	X	
Coding		X	X						X[15]	
Verification	X	X	X[3]			X	X	[14]	X[3]	X

Table 1: Methodologies Comparison (part 1 of 2).

	Methodology/System									
	Barr & Mearns	Chen & Hoare	Chen & Yeh	Flon & Suzuki	Goguen	Jones	Lamport	Luckham	Misra & Chandy	Owicki
Forms of Logic:										
Predicate calculus			X			X	X			
Propositional logic						X				X[16]
Temporal logic							X			X
HOL										
Boyer-Moore										
Equational					X					
Hoare logic	X	X[1]		X			X		X	
Quantitative Measures:										
Actual experience	?			little	much	some	some	some+		some
Biggest example				small	med+	small	med+	med	med-	
Qualitative Measures:										
Top-down support		X	X		weak		X		X	X
Interface to real language	CSP	CSP			OBJ3			Ada		
Ease of use:										
Specification		fair	tricky		good	fair	fair	good	fair-	hard
Verification	good	fair	tricky	good	fair	good	fair		fair-	hard
System implemented					X			X		
System Features [if implemented]										
Mechanical theorem prover support					X[8]			X[11]		
Integrated with larger software envir.	X			X	X				X	
Our experience with system					some			none		

Table 1: Methodologies Comparison (part 2 of 2).

Table 1 Notes:

1. Chen and Hoare have extended the original Hoare logic to support the concurrency features of CSP.
2. The level of concurrency in the Chen and Yeh methodology is lexically at the statement level, but semantically interaction is only enforced at statements dealing with port interaction, which means that the granularity is really more at the process level.
3. Verification in the Chen and Yeh approach focuses on processes and interprocess communication; verification of other aspects of the program are not considered but could presumably be handled with existing techniques.
4. The methodology of Flon and Susuki is based on the transformation of concurrent programs to sequential programs, after which the verification is carried out using sequential verification techniques; in this sense the methodology does not support any model of concurrency directly, though the concurrent programs that are transformed are assumed to use a shared variable model.
5. Goguen's OBJ language does not support any model of concurrency explicitly. Rather, Goguen's goal for the realisation of concurrent programs is to construct a specification which abstracts away any explicit details of concurrency, but which is stated in a functional language that is amenable to translation into a form that is concurrently executable. That is, Goguen's goal is to move all consideration of concurrent execution out of the hands of the specifier/programmer into the realm of the language translator. Our work on extensions to OBJ3 includes features that do allow the explicit representation of concurrency within the OBJ framework. Details of OBJ and our extensions to it are covered in later sections of the report.
6. See note 5.
7. Since OBJ is a fully functional language, there is no clear distinction between a function invocation, a statement, and an expression.
8. Theorem proving support is based on the rewrite execution capability of OBJ. However, there is not much in the way of other proof management facilities, such as proof management, built-in support for induction, etc. We are currently adding some of these capabilities to OBJ.
9. Although Jones' and Owicki's methodologies focus explicitly on the shared-variable level of concurrency, aspects of his methodology are applicable to other concurrency models.
10. Jones suggests that temporal logic should be incorporated into his methodology.
11. Luchham and coauthors hint at (wish for) proof rules for their Temporal Specification Language (TSL) methodology.
12. The Luckham TSL methodology is based on Ada which does not support asynchronous message passing.
13. Based on Ada, TSL granularity is at the task level.
14. TSL is based on a form of temporal logic combined with path expressions.
15. Coding is supported to the process level, but not below.
16. Owicki's methodology adds history and other auxilliary variables to the base propositional logic.
17. As described below, we have implemented some extensions within the HOL system to support a form of equational logic, and this is an area of continuing research.
18. In papers on the Larch system, the authors describe the interface from the Larch equational logic to predicate logic, but it is unclear from system descriptions if the implemented system supports machine-based predicate logic.

2.2 Barringer and Mearns

2.2.1 Overview

Barringer and Mearns propose axiomatic proof rules for the partial correctness and deadlock free in Ada tasks. The work is based on proof methods for CSP proposed by Apt, Francez and de Roever. In this approach, the correctness of a parallel program is proved in two steps. First, each process of the parallel program is proved in isolation as an individual sequential program. Then, a cooperation proof is made to ensure that the interaction between processes does not invalidate the sequential proofs.

2.2.2 Technical Discussion

The following assumptions are made about Ada programs:

1. tasks do not share global variables
2. task does not call its own entries
3. all constructs terminate normally
4. calls to subprograms have no side effects
5. assignments have no side effects
6. no two tasks have the same name

In this methodology, the correctness of a parallel program is defined in term of a global assertion GI that asserts variables from all tasks. Since tasks are assumed not to share global variables and since rendezvous is the only communication mechanism in Ada, the proof rules are centered on axioms of the rendezvous mechanism.

Barringer and Mearns define the following terms in order to facilitate their axioms:

Input/output commands (IO): they are just entry calls and accept statements. Two IO commands are said to match if one command is an entry call to the other's accept statement.

Bracketed sections (BS): these are sections of the program, delimited by '<' and '>' of the form:

$s_1; IO; s_2;$

where s_1 and s_2 are Ada constructs that do not include any IO command.

The global invariant GI needs not hold within a bracketed section BS but no variable free in GI may change outside a BS. Two bracketed sections match if they contain matching IO commands. Every IO in a program must appear in a BS and must match with at least one other IO.

Since the body of an Ada accept statement may contain entry calls and/or accept statements, a bracketed section may be nested. Barringer and Mearns propose that auxiliary variables can be used to incorporate all the "intermediate global assertions" into GI.

Co-operation: Given proofs of $\{P_i\}$ T_i $\{Q_i\}$ for all tasks T_i, $1 \leq i \leq n$, considered in isolation, the n local proofs are said to co-operate if

1. the assertions used in the local proof of $\{P_i\}$ T_i $\{Q_i\}$ have no free variables subject to change in T_j $(i \neq j)$;
2. $\{pre(BS_1) \wedge pre(BS_2) \wedge GI\}$ $BS_1 \| BS_2$ $\{post(BS_1) \wedge post(BS_2) \wedge GI\}$ holds for all matching pairs of bracketed sections BS_1 and BS_2.

The paper also discusses how to prove blocking free and deadlock free. The proof methods are similar to the methods of Owicki & Gries and Apt et al.

A summary of the proof rules involving Ada rendezvous is as following:

Entry call

$\{P\}\ T_i.E_j\ \{Q\}$

Accept statement

$\{P\}\ \mathit{accept}\ E_j(\ldots)\ \mathit{do}\ s;\ \mathit{end}\ \{Q\}$

General communication rule

$$\frac{\{P[f_{io}/a_{io}] \wedge f_{in} = a_{in}\}\ s\ \{Q\}}{\begin{array}{c}\{P\} \\ T_i.E_j(a_{in}, a_{io}, a_{out})\| \\ \mathit{accept}\ E_j(f_{in} : \mathit{in}\ldots;\ f_{io} : \mathit{in\ out}\ldots;\ f_{out} : \mathit{out}\ldots)\ \mathit{do}\ s;\ \mathit{end}; \\ \{Q[a_{io}/f_{io}, a_{out}/f_{out}]\}\end{array}}$$

Parallel composition rule

$$\frac{\text{for all tasks} T_i,\ 1 \leq i \leq n,\ \text{proofs}\ \{P_i\}\ T_i\ \{Q_i\}\ \text{cooperate}}{\{P_1 \wedge \ldots \wedge P_n \wedge GI\}\ T_1\|\ldots\|T_n\ \{Q_1 \wedge \ldots \wedge Q_n \wedge GI\}}$$

Formation rule

$$\frac{\{P\}\ s_1; s_3\ \{P_1\}, \{P_1\}\ IO_1\|IO_2\ \{Q_1\}, \{Q_1\}\ s_2; s_4\ \{Q\}}{\{P\}\ < BS_1 > \| < BS_2 >\ \{Q\}}$$

$\mathit{where} BS_1 = s_1; IO_1; s_2$ and $BS_2 = s_3; IO_2; s_4$

Transformation rule

$$\frac{\{P\}\ T'_1\|T'_2\|\ldots\|T'_n\ \{Q\}}{\{P\}\ T_1\|T_2\|\ldots\|T_n\ \{Q\}}$$

2.2.3 Critical Remarks

The methodology can be used to prove some interesting properties of Ada tasks such as blocking free, deadlock free, and termination. However, it is not clear how the method can be generalized to prove Ada tasks that communicate through shared variables. In general, Barringer and Mearns methodology is just a retrospective fitting of proofs, it cannot be used as a development tool.

2.3 Chen and Hoare

2.3.1 Overview

Zhou Chao Chen and C.A.R Hoare presented a paper entitled *Partial Correctness of Communicating Processes* which was almost immediately followed by a paper by C.A.R. Hoare, entitled *A Calculus of Total Correctness for Communicating Processes.* In these two papers they introduce a programming notation used to "...describe the behavior of groups of parallel processes, communicating with each other over a network of named channels. "

The methodology presented in these papers outlines the necessary proof rules and techniques necessary to prove the desired behaviors. Each channel of communication has an auxiliary variable associated with it. This variable (or trace) is an ordered sequence consisting of all messages communicated over the channel, direction of the message is unspecified. In Hoare's paper he also added an auxiliary variable to each process, one for each channel connected to that process. This variable is the *ready* state of the process. If the process is ready to communicate a message along the channel it contains the value to be sent, if it is ready to receive a message along the channel the variable contains the set of all possible messages (the type class) it can receive.

Proofs in this methodology deal mainly with a hierarchical development of the system. Properties (behaviors) of the outermost layer is specified in terms of sequence of values communicated over the external channels. Once the restrictions on the channels are shown to maintain these properties, we then implement the lower layer in terms of parallel communicating processes, or in actual code. In the processes case, we must specify properties about the internal communication channels. In both cases we must show that the implementation maintains the properties specified at the higher level. This allows us to work with a small group of proofs at a time and use a systematic top-down development/proof of the system.

2.3.2 Technical details

To manipulate the proofs, we need to use the following "health" rules and proof rules:

(H1) P **sat** $TRUE$

(H2) $\neg(P \textbf{ sat } FALSE)$

(H3)

$$\frac{R \Rightarrow S}{(P \textbf{ sat } R) \Rightarrow (P \textbf{ sat } R)}$$

If n is *not* a channel variable, and does *not* occur in P:

(H4) $(\forall n : N.P \textbf{ sat } R(n)) \equiv (P \textbf{ sat } (\forall n : N.R(n)))$

The following is a list of proof rules and inference rules that Hoare uses when proving properties of the processes. **(R1)** *Output*

$$\begin{aligned}((c!e \rightarrow P) \textbf{ sat } R) \quad &\equiv \quad (R[\langle\,\rangle/past, \{e\}/c.ready, 0/ready] \\ &\quad \& \ P \textbf{ sat } (R[\langle e\rangle c.past/c.past])\end{aligned}$$

Let R be an assertion *not* containing x.:

(R2) *Input*

$$\begin{aligned}((c?x : M \rightarrow P(x)) \textbf{ sat } R) \quad &\equiv \quad (R[\langle\,\rangle/past, M/c.ready, 0/ready] \\ &\quad \& \ \forall x : M.(P(x) \textbf{ sat } R[\langle x\rangle c.past/c.past]))\end{aligned}$$

Define the 'fixed point' of F to be $\mu p.F(p) = F(\mu p.F(p))$.

Also define $R \restriction n$ (R restricted to n) to be: $R \restriction n \ \underline{\triangle}\ (\#a.past + \cdots + \#z.past \geq n) \vee R)$ where $\#s$ stands for the length of the sequence s.

(R3) *Recursion*

$$\frac{(p \textbf{ sat } (R \restriction n)) \Rightarrow (F(p) \textbf{ sat } (R \restriction n+1))}{\mu p.F(p) \textbf{ sat } R}$$

(R4) *Stop*

$$(STOP_A \textbf{ sat } R) \quad \equiv \quad R[0/ready, \langle\,\rangle/past]$$

(R5) *Disjoint Parallelism*

$$\frac{(P \textbf{ sat } S) \ \$ \ (Q \textbf{ sat } T)}{(P \| Q) \textbf{ sat } (S \ \& \ T)}$$

In the review of this paper by Barringer, Barringer uses as an example a version of a Bubble Lattice Sort. Initially he specifies the properties of the *Sort* process. Then *Sort* is defined as a parallel composition of *Comp* processes with the *IN* and *OUT* channels appropriately connected. Now, using the inference rules we can prove that the above decomposition actually implement the specification of *Sort*.

Once we get to the level of code, we can use the inference rules presented by Chen and Hoare, or the modified versions of Hoare, and use these around the communication primitives in the code. (The code in this case contains CSP-style communication primitives). Other portions of the code can be verified separately using the standard axiomatic techniques of Hoare.

2.3.3 Critical Remarks

The method presented by Chen and Hoare, and later modified by Hoare, is a good hierarchical development of a proof methodology. We can work from a high-level description of the process down to the code level.

Using these techniques we can prove absence of deadlock, termination, fairness. This model does not seem to be able to be proven complete, and has yet to be proven consistent. It also contains no way of proving that a process P is *non*-deterministic. In other word if P **sat** R then there exists a *deterministic* process Q such that Q **sat** R. Hoare also states that the axiomatization of sequential composition, local variables, and assignment may cause some difficulty in this methodology.

2.4 Chen and Yeh

2.4.1 Overview

The model presented in the paper of Chen and Yeh is based on events and their relationships. An event is described as "...an *instantaneous, atomic* state transition in the computation history of a system."

Using this event-based model they can specify interrelationships between processes based on the ordering of these processes. If an ordering is not specified this implies an implicit concurrent behavior amongst the processes.

They present an Event Based Specification Language (EBS) as a tool for formally specifying systems using the event-based model. This language is based on *partial orders* and *first-order predicate calculus*. The hope is that using EBS, the developer can avoid some of the more cumbersome details involved in writing the specification when using a trace or temporal based model, yet still have the ability to express everything those models allows him to express.

2.4.2 *Technical* Discussion

To implement an ordering between events the develop an "precedes" relation denoted by "$\rightarrow$". This is a partial ordering that helps remove the system clock requirements necessary for a total ordering. If e_1 and e_2 are events in the system and $e_1 \rightarrow e_2$ then e_1 precedes e_2 by some measure of time. They use this relation and specify the transitivity, irreflexivity, and antisymmetry properties of it.

Concurrency is specified by stating that if $\neg(e_1 \rightarrow e_2) \wedge \neg(e_2 \rightarrow e_1)$ then e_1 and e_2 are concurrent.

They develop an "enables" relation which denoted by $\Rightarrow$ that states if $e_1 \Rightarrow e_2$ then the existence of event e_1 will cause the occurrence of event e_2 and some time in the future. This relation is also irreflexive, transitive, and antisymmetric.

Finally they specify the system, environment and interface ports. Where the system interacts with the environment through the interface ports. Associated with each port is a history of uniquely identified interface events.which creates a total ordering.

After presenting the above definitions and relationships they present details of EBS for specifying the behavior of a distributed system. This language consists of the following operators and precedence rules:

- **unary operators**: $\forall$ (for all), $\exists$ (there exists), and $\neg$ (logical negation);
- **relation operators**: $\in$ (belongs to), $=$ (equivalent), $\equiv$ (equals to), $\Rightarrow$ (enables), and $\rightarrow$ (precedes);

- **logical operators:** $\vee$ (logical or) and $\wedge$ (logical and);
- **implication:** $\# >$ (logical implication) and $< \# >$ (two way logical implication).

An example, given in the paper, is of a reliable transmission system with input port A and output port B. The equation specifies the property that for some message a_1 that is sent before message a_2 then it is received before a_2:

(* RT15(A,B): no out-of-order messages *)

$$\forall a_1, a_2 \in A, b_1, b_2 \in B$$
$$a_1 \Rightarrow, b_1 \wedge a_2 \Rightarrow b_2$$
$$\# > (a_1 \rightarrow a_2 \wedge b_1 \rightarrow b_2) \vee$$
$$(a_1 \equiv a_2 \wedge b_1 \equiv b_2) \vee$$
$$(a_2 \rightarrow a_1 \wedge b_2 \rightarrow b_1)$$

Now that we have specified properties of the system, we can use these as building blocks for more complicated systems. Properties of the more complicated systems can be proven using the properties specified at the lower levels.

They state that the EBS language has the following benefits:

- *Formality*: Partial ordering relations and first order predicate calculus guarantee this
- *Generality*: Safety, liveness, data-related and control-related properties can be specified and verified.
- *Accuracy*: The inherent behavior of distributed systems is represented by the lack of ordering among events; mutual exclusion is specified by the preceded operation.
- *Orthogonality*: Properties are specified separately, which makes specification minimal and extensible.

2.4.3 Critical Remarks

A trace (history sequence) of CSP can be seen as a sequence of events yet properties specified in EBS are often much easier to manipulate than using traces. This should therefore allow a greater flexibility and ease of specification compared to the other methodologies.

The examples presented in the paper seemed to be designed bottom-up although the methodology apparently can be used for a top down hierarchical design and specification of a system. Further work with this methodology and development of our canonical model using this method is recommended.

2.5 Flon and Suzuki

2.5.1 Overview

Flon and Suzuki propose that a parallel program whose processes communicate through shared memory can be converted to an equivalent non-deterministic sequential program, and that by proving some interesting properties of the non-deterministic program, we can indirectly verify the semantics on the parallel version. The properties that can be proved are invariance, potentiality, inevitability, absence of blocking, absence of deadlock, and absence of starvation. Flon and Suzuki provide a procedure and a set of inference rules to facilitate the program conversion and the proving processes. The inference rules are based on weakest precondition predicate transformation.

2.5.2 Technical Discussion

A non-deterministic sequential program is equivalent to a parallel program if for any execution sequence of the non-deterministic sequential program, there is a corresponding execution sequence in the parallel counterpart such that values of variables in the parallel version have the same history sequence. In Flon and Suzuki method, a parallel program composed from the cobegin-coend construct can be converted to a sequential program denoted by the REP construct. The REP construct is a repetitive guarded command and can only be terminated by an exit command.

$$REP ::= rep\ B_1 \rightarrow S_1 \| B_2 \rightarrow S_2 \| \ldots \| B_n \rightarrow S_n\ per$$

The conversion procedure is as following. First of all, the entire parallel program is converted to an indivisible form by introducing variables local to each process. Then, each statement in the parallel program is transformed to an equivalent statement using the rules provided by Flon and Suzuki. The rules cover the transformations of cobegin-coend construct, block construct, assignment construct, conditional construct, and loop construct.

Flon and Suzuki also provide a set of inference rules to assert the weak correctness, blocking free, deadlock free, and starvation free properties of a non-deterministic sequential program, and claim that these properties can be applied to the parallel version. The following is a summary of the inference rules:

Invariance I

$$\frac{\forall k : 1 \leq k \leq n : (I \wedge B_K \Rightarrow wlp(S_k, I)), I \Rightarrow R}{I \Rightarrow wip(REP, R)}$$

where wlp is Dijkstra's weakest liberal precondition, i.e., the construct is not required to terminate. wip is the weakest precondition for invariance

Potentiality

$$\frac{\neg Q(0), Q(m+1) \wedge \neg R \Rightarrow \exists k : 1 \leq k \leq N : (B_k \wedge wp(S_k, Q(m)))}{Q(m) \Rightarrow wpp(REP, R)}$$

where wpp is the weakest precondition for potentiality.

Inevitability

$$\frac{\neg Q(0), Q(m+1) \wedge \neg R \Rightarrow \exists i : 1 \leq i \leq N : (B_k \Rightarrow wp(S_k, Q(m)))}{Q(m) \Rightarrow wep(REP, R)}$$

where wep is the weakest precondition for inevitability.

Blocking free

$$\frac{\forall k : 1 \leq k \leq N : (I \wedge B_k \Rightarrow wp(S_k, I)), I \Rightarrow \exists k : 1 \leq k \leq N : B_k}{I \Rightarrow wbp(REP)}$$

where wbp is the weakest precondition for blocking free.

Deadlock free

$$\frac{\forall k : 1 \leq k \leq N : (I \wedge B_k \Rightarrow wp(S_k, I)), I \Rightarrow wpp(REP, B_j) \vee Q}{I \Rightarrow wdp_j(REP)}$$

where Q is the disjunction of all the guards of exit commands, and wpp is the weakest precondition for deadlock free.

Starvation free

$$\frac{\forall k : 1 \leq k \leq N : (I \wedge B_k \Rightarrow wp(S_k, I)), I \Rightarrow wep(REP, B_j) \vee Q}{I \Rightarrow wsp_j(REP)}$$

where wsp is the weakest precondition for starvation free.

2.5.3 Critical Remarks

Since the primary objective of the Flon and Suzuki methodology is to prove some interesting properties of an existing parallel program, the methodology cannot be used for the hierarchical program development, and cannot be used in conjunction with program development. The application of the proof rules is straightforward; but obtaining a good invariance and a valid metric is not a trivial task and requires a complete knowledge of behavior of other processes.

It is questionable whether the method can be applied to real examples. The parallel to sequential program conversion algorithm seems too tedious to follow, and some optimization should be performed to ease the proving process.

2.6 Halpern and Owicki

2.6.1 Overview

Halpern and Owicki's method consists of three steps: (1) Model a parallel program as a set of modules that interact by procedure calls (2) Verify module properties using standard sequential techniques (3) Compose modules into a system. Module specifications consist of an invariant, a commitment, and service specifications. The invariant refers to safety properties and the commitment refers to liveness properties (using temporal logic). The service specifications correspond to each exportable procedure and consist of preconditions and postconditions which refer to both safety and liveness properties, and of a liveness condition, which states the conditions under which procedure termination is guaranteed.

Temporal logic is an important tool in this technique. Temporal logic is an extension of ordinary logic to include time. Halpern and Owicki's notation includes:

$\Diamond P$	At some time P will be true (eventually)
$\Box P$	P will always be true (henceforth)
$\Box\Diamond P$	P will be true *infinitely often*
$A \leadsto B$	Temporal implication; if A is true, $\Box B$

In addition to temporal logic, Halpern and Owicki make use of auxiliary variables, histories, and private variables. Auxiliary variables are used in specifications and proofs; they are included only for convenience in reasoning about a program. History variables are a special class of auxiliary variables; they are unbounded sequences used for recording interactions between modules. A variable x which is 'private' in module M will have one instance for each module calling M. The notation x[C] denotes a module referring to C's version of x; x[*] denotes a module referring to its own version of x.

2.6.2 Technical Discussion

Halpern and Owicki's goal for this system is to ensure that the proof that a module meets its specifications is independent of the code of other modules in the systems. To achieve this, it is important that each module's assertions be robust (i.e., other modules cannot make them false). This is done by restricting assertions in each module M to variables local to M. Thus, a module's invariant and commitment clauses use only local variables and the service specifications use only procedure parameters and private variables of the called module. Furthermore, assertions are monotonic (i.e., once true they cannot be made false).

For verification purposes, the invariant of a simple module must hold when that module is ready to interact with another module either by accepting or initiating a procedure call. For compound modules, specifications are verified from the specifications of the components. Invariant and commitment are shown to be implied by the conjunction of the component invariants and commitments. Service specifications may be proved from the code or carried over from the component.

Note that it is always possible to assign values to the variables that are not open such that the invariant is satisfied.

2.6.3 Critical Remarks

In Halpern and Owicki's method, the proof process is simplified since proofs from code involve oniy sequential reasoning. In addition, an individual module's specification can be verified without consideration of other modules (except service specifications). Finally, composition of system specifications is based on logical reasoning, not the code that it is based upon.

2.7 Jones

2.7.1 Overview

Jones proposes a rigorous method for developing a program comprising tasks which execute in parallel and interfere with each other. A *rigorous* method relies on underlying mathematical ideas but uses these foundations for less formal arguments. The approach taken is intended to be precise without being bulky. When complete formality is required, the necessary details can be added.

In a parallel program, a task *interferes* with another by modifying a variable accessible to both or by message passing. Jones emphasizes shared-storage parallelism. There seems to be some confusion concerning the method's applicability to message-passing parallelism. However, message passing can be considered as a restricted form of shared-storage access. Indeed, sending a message to a task amounts to modifying its message queue in accordance with the intertask protocol. This protocol invariably involves semaphore sequencing.

Programs are *developed (implemented)* by the *recursive* application of *operation decomposition*. *Operation* is a noncommittal term for aprocedure, a function, or even a statement. As each high-level operation is implemented in terms of low-level operations, only the specifications of the low-level operations are necessary to justify the decisions embodied in the decomposition. This is in contrast to methods requiring a completely coded solution. At the leaves of the development tree, operations are implemented by programming-language statements. Their effects must also be specified. Thus, for a thorough specification, the language employed should enjoy a formal semantics, preferably provided in a style similar to that of Jones. Although Jones uses Ada for examples, the method is not language dependent.

Specifications are operation stubs that include a specification block. A *stub* is an operation type signature and a list of global variables that need to be accessed by the operation. A *specification block* comprises a pre-condition, post-condition, rely-condition, and guarantee-condition. Jones' main contribution is the complementary interaction of rely-conditions and guarantee-conditions.

2.7.2 Technical Discussion

Specifications are written using pre-conditions, post-conditions, rely-conditions, and guarantee-conditions. Every operation in a problem decomposition has one of each; if not, defaults apply. But before these are discussed more precisely, we need a few more definitions.

A *state* is a collection of named values. More concretely, a task's state is the collection of variable values explicitly and implicitly accessible by the task. Thus, an *operation* can be viewed as a function from one state to another. However, Jones emphasizes that while an operation can change a value within a state, it cannot change the structure (i.e., add or delete variables). Specifications employ *predicates* on states -- functions on states with Boolean range. A *relational* *predicate* on states is a function on two states with Boolean range whose arguments are recognized as initial and final states. The final state, as well as its components, are denoted by priming. For example, $p(\sigma, \sigma') \triangleq true$ is a relational predicate named p on initial state σ and final state σ' defined to be *true* (i.e., always satisfied). Although relational predicates are no more powerful than plain predicates, Jones claims their use simplifies the specification of larger problems.

A *pre-condition* is a predicate on a single state specifying over what subset of all possible states an operation must succeed. If its pre-condition is not satisfied, an operation is completely unconstrained (however, an error message would be appreciated).

A *post-condition* is a relational predicate specifying the required relationship between initial and final states. It describes the effect of an operation on a state satisfying the pre-condition. An appropriate post-condition for a nonterminating operation (e.g., an operating system) is somewhat arbitrary.

A *rely-condition* is a relational predicate specifying how other tasks can modify the state. Here, the initial state is that before modification and the final state is that after modification. A rely-condition must hold across every interfering modification to the state. Curiously, a task may violate its own rely-condition yet satisfy its specification. This phenomenon occurs, for example, when many tasks are allowed to read a global variable, but only one task is allowed to change its value. The writer relies on constancy but, of course, it may modify the variable.

A *guarantee-condition* is a relational predicate specifying how the task containing it can modify the state. Once again, the initial state is that before modification and the final state is that afterwards. A guarantee-condition must hold across all modifications to the state made by the task containing it.

Intuitively, pre-conditions and post-conditions must hold once per operation execution; the pre-condition on entrance and the post-condition on exit. In contrast, rely-conditions and guarantee-conditions must hold at (potentially) many times during operation execution. Any time a variable which is accessible to multiple tasks is modified, the guarantee-condition of the modifying operation must hold and the rely-condition of all operations having access to the variable must hold. This potential complexity is bravely represented in the proof obligations to come.

One way to view rely-conditions and guarantee-conditions is as concurrent versions of pre-conditions and post-conditions, respectively. Another is as invariants holding only at certain times.

A partial specification — one missing one or more of the four conditions — assumes default conditions. These are shown below.

$$\textit{pre-Default}(\sigma) \Leftrightarrow \textit{true}$$
$$\textit{post-Default}(\sigma, \sigma') \Leftrightarrow \textit{true}$$
$$\textit{rely-Default}(\sigma, \sigma') \Leftrightarrow \sigma = \sigma'$$
$$\textit{guar-Default}(\sigma, \sigma') \Leftrightarrow \textit{true}$$

The default pre-condition specifies an operation succeeding on any state. Such an operation is analogous to a total function. The default post-condition does not constrain an operation's modification of the state at all. Such an operation can modify every accessible variable or do nothing. The default rely-condition assumes complete noninterference, as in a sequential environment. The default guarantee-condition guarantees nothing, any interference is allowed.

Recall that Jones' method is for developing a program by decomposing a problem. The interesting decomposition steps are those involving the implementation of an operation by a set of suboperations that execute concurrently. Call the main operation OP and the suboperations $T_1, T_2, \ldots, T_n$. In order to prove that the development step is correct, several proof obligations must be satisfied. Each is discussed below.

The tasks must rely on no more than the main operation:

$rely\text{-}OP(gl, gl') \Rightarrow (\forall i : 1 \leq i \leq n : rely\text{-}T_i((gl, loc), (gl', loc)))$.

Here, *gl* is the state accessible to operations with access to *OP*. In contrast, *loc* comprises the local variables of *OP* — those shared by the suboperations. Since *loc* represents variables inaccessible to operations outside of *OP*, it cannot be modified by those operations, and the *loc* component of the final state of each task's rely-condition is implicitly guaranteed to be identical to the *loc* component of the initial state. Thus, it need not be primed.

Independently, the tasks must guarantee what the main operation guarantees:

$(\exists i : 1 \leq i \leq n : guar\text{-}T_i((gl, loc), (gl', loc'))) \Rightarrow guar\text{-}OP(gl, gl')$.

Here, *loc* is comprises variables which are accessible to the task making the guarantee. Satisfaction of this guarantee might involve modifying such a variable. Thus, in the final state *loc* is primed.

Each task's guarantee-condition must be at least as strong as every other task's rely-condition — the tasks must be able to coexist:

$(\forall i, j : 1 \leq i \neq j \leq n : guar\text{-}T_i(\sigma, \sigma') \Rightarrow rely\text{-}T_j(\sigma, \sigma'))$.

Here, σ is the conglomeration of global and local variables accessible to a task. Note that since $i \neq j$, a task can do what it relies on other tasks not to do.

The main operation must satisfy the precondition of each task:

$pre\text{-}OP(\sigma) \Rightarrow (\forall i : 1 \leq i \leq n : pre\text{-}T_i(\sigma))$.

This obligation, or more precisely its imprecision, suggests that there are an enormously large number of proof obligations — Jones explicitly identifies only some of them. We identify the others later, as a criticism of Jones' method.

The remaining proof obligations refer to a *dynamic invariant*. This is a relational predicate specifying that the computation being performed is solving the problem. A dynamic invariant is comparable to the loop invariant of Hoare Logic fame. For example, if the problem is to find the maximum element in an array, the dynamic invariant should say something along the lines of "the candidate solution is no less than all elements examined so far". Clearly, the pre-condition of the main operation must establish the dynamic invariant:

$pre\text{-}OP(\sigma) \Rightarrow dinv(\sigma, \sigma)$.

This corresponds to the basis in an induction proof.

Furthermore, the interference expected by the main operation must not violate the dynamic invariant:

$dinv(\sigma, \sigma') \wedge rely\text{-}OP(\sigma', \sigma'') \Rightarrow dinv(\sigma, \sigma'')$.

Since the tasks implementing the main operation are expected to contribute to the solution, they should preserve the dynamic invariant:

$dinv(\sigma, \sigma') \wedge (\exists i : 1 \leq i \leq n : guar\text{-}T_i(\sigma', \sigma'')) \Rightarrow dinv(\sigma, \sigma'')$.

Finally, if the dynamic invariant has survived the execution of each task and each task has terminated successfully, the main operation should terminate successfully:

$dinv(\sigma, \sigma') \wedge (\forall i : 1 \leq i \leq n : post\text{-}T_i(\sigma, \sigma')) \Rightarrow post\text{-}OP(\sigma, \sigma')$.

Actually, the preceding proof obligation does not mention termination. Therefore, a nonterminating task needs a post-condition of *true*.

In order to abbreviate specifications and simplify proofs, Jones allows access restrictions to be attached to the declaration of a variable. For example, a variable can be declared as *read-only* within a task, making post-condition and rely-condition clauses specifying this form of access unnecessary. This is a good example of Jones' philosophy of rigor versus formality. Unfortunately, he does not provide a list of allowable abbreviations — perhaps *read-only* is the only one.

2.7.3 Critical Remarks

Jones' method is primarily an extension of his sequential-program development method. This sequential method utilizes only pre-conditions and post-conditions. As with many augmented approaches, there are sharp edges. Some of them are discussed here.

While useful for documenting interference assumptions and promises, Jones' method appears to be deficient concerning task synchronization. For example, he recognizes that deadlock avoidance requires an extension to his method. He suggests some form of temporal logic or a location predicate. Perhaps a location predicate can be simulated by auxiliary variables, as is done to control interference when developing a "proof outline" for a concurrent program. If so, no extension is necessary. Unfortunately, such simulation is rather awkward.

Another difficulty is deciding where to put (i.e., conjoin) individual specification clauses. A pre-condition clause can be confused for a rely-condition clause, and vice-versa; a post-condition clause can be confused for a guarantee-condition clause, and vice versa. Moreover, a clause often needs to be in both confusable conditions. The problem seems to be that a task's pre-condition and rely-condition are independent — except at task entry. Likewise, the post-condition and guarantee-condition are independent — except at task exit.

Recall the claim that Jones explicitly identifies only some proof obligations; we elaborate this claim here. In his examples, Jones gives a specification for each task or function — no finer granularity is mentioned. In fact, however, the behavior of each statement in a program needs to be specified by a pre-condition, post-condition, rely-condition, and guarantee-condition (or defaults). This is similar to a "proof outline", where an assertion is placed before and after every statement. Such detail allows implementations to be proven correct, but the universal quantifiers in the proof obligations cause the amount of work required to explode.

A related criticism concerns a higher-level view of the development process. Recall that a major advantage of Jones' method is that correct-implementation proofs can be carried out between decomposition steps, rather than after complete decomposition. Thus, the reader is perhaps seduced by the suggested development sequence:

$$\{\text{decompose}, \text{prove}\}_{i=1}^{n}$$

resulting, after n steps, in a verified solution. Alas, experience suggests the more alarming:

$$\left\{\text{decompose}, \text{prove}^{f(i)}\right\}_{i=1}^{n}$$

where the function f grows pretty fast. The problem is that interference can occur between operations on nonadjacent levels in the development tree; this potential interference generates proof obligations. The deeper the tree gets, the more obligations there are at each step. There is, perhaps, no way to avoid this proof explosion when verifying concurrent programs.

2.8 Lamport

2.8.1 Overview

Lamport has done a great deal of work regarding the specification of concurrent programs [OL82][Lam82][Lam86]. The model used in the bulk of this work is a formal proof methodology based upon temporal logic. Temporal logic may be used to describe a concurrent program at any level of abstraction, which permits hierarchical specification and verification.

Lamport's method of verifying concurrent programs as described in [Lam86] consists of factoring a global program invariant into 'pieces' that are attached to control points in the program. Control state then an explicit part of the invariant, and is described via control predicates. Control state is necessary in Lamport's method, since the invariant assertions attached to one process may need to refer to the control state of other processes. The Owicki-Gries method uses dummy—or auxiliary—variables rather than explicit encoding of control information. An extended version of the Owicki-Gries method is described, which may be used to prove a larger class of program invariants.

2.8.2 Technical Discussion

In [OL82], Owicki and Lamport describe a method whereby liveness properties are derived using temporal logic and proof lattices. Temporal logic is a method of specifying assertions about future events. Thus, two new operators are added to propositional calculus:

$\Box P$ P is true from now on

$\Diamond P$ P will be true at some time hereafter

A third piece of notation is also useful: the assertion $P \leadsto Q$ is true iff during an execution of a program which reaches a state where P is true, Q becomes true eventually. This notation will be useful in describing proof lattices.

Proof lattices are written as finite directed acyclic graphs, where predicates are shown as nodes and a directed arc is drawn between predicates P, Q if the relation $P \leadsto Q$ holds. Each lattice has a single exit node and a single entry node. Clearly, if a proof lattice with entry point P and exit point Q exists, then $P \leadsto Q$.

A simple programming language is introduced; its only statements are: **assignment**, **while**, **cobegin**, and variable declaration. The semantics of the program consist of the set of all its possible *execution sequences*. Execution sequences describe a series of program states, each consisting of a value-assignment to each program variable and a *control component*. The control component consists of a set of *atomic actions* which are eligible for execution. Concurrency is modeled by "a nondeterministic interleaving of atomic actions from the various processes ... almost any concurrent system can be accurately modeled this way ... any safety or liveness properties proved about the model will be true of the system." An additional constraint on the model is that no process is 'infinitely faster' than any other, which provides fairness; additionally, atomic actions are assumed to terminate.

The control component may contain three different assertions: *at* A, *in* A, and *after* A (A is an executable program statement). These assertions mean: control is at the program point immediately preceding statement A, statement A is being executed, control is at the program point immediately following A.

Liveness properties are statements about control flow; thus, they may be derived from program invariants which include control flow information. Owicki and Lamport provide the following rules for this purpose:

Control Flow Rule 1 (Concatenation) *For the statement S;T*

$$\frac{\mathit{at}\ S \leadsto \mathit{after}\ S\ ,\mathit{at}\ T \leadsto \mathit{after}\ T}{\mathit{at}\ S \leadsto \mathit{after}\ T}$$

Control Flow Rule 2 (COBEGIN) *For the statement c:COBEGIN S || T COEND*

$$\frac{\mathit{at}\ S \leadsto \mathit{after}\ S\ ,\mathit{at}\ T \leadsto \mathit{after}\ T}{\mathit{at}\ c \leadsto \mathit{after}\ c}$$

Control Flow Rule 3 (Single Exit) *For any statement S:*

$\mathit{in}\ S \Rightarrow (\Box \mathit{in}\ S\ \text{or}\ \Diamond \mathit{after}\ S\ )$

Control Flow Rule 4 (Atomic Statement) *For any atomic statement* $< S >$:

$$\frac{\{P\} < S > \{Q\}, \Box(at < S > \Rightarrow P)}{< S > \leadsto (after\ < S > \wedge Q)}$$

Control Flow Rule 5 (General Statement) *For any statement S:*

$$\frac{\{P\}S\{Q\}, \Box(in\ S\ \Rightarrow P), in\ S\ \leadsto after\ S}{in\ S\ \leadsto (after\ S\ \wedge Q)}$$

Control Flow Rule 6 (while TEST) *For the statement w: while* $< B >$ *do S od*

$$at\ w\ \leadsto ((at\ S\ \wedge B)\ or\ (after\ w\ \wedge \neg B))$$

Control Flow Rule 7 (while EXIT) *For the statement w: while* $< B >$ *do S od*

$$(at\ w\ and \Box(at\ w\ \Rightarrow B)) \leadsto S$$

$$(at\ w\ and \Box(at\ w\ \Rightarrow \neg B))after\ w$$

These rules may be used to create a graphical lattice proof of liveness properties. The paper describes a liveness proof of the standard mutual exclusion problem, using the following invariant:

$$(at\ \mathrm{S}\ and \Box\Diamond\neg in\ CS_2) \Rightarrow (after\ NC_1 \leadsto in\ CS_1)$$

where CS_i is the critical section of process i and NC_i is the non-critical section of process i. This formulation states that process 2 is guaranteed access to its critical section if process 1 remains in its noncritical section. (Unfortunately, this logic cannot be used to express the more useful statement that "within a reasonable amount of time" each process will be permitted access to its critical section; it is only possible to state that 'always' a property will hold or 'eventually' a property will hold.) The proof consists of starting with the contradictory expression—that process 2 does not eventually get access to the critical section—and generating a proof lattice which leads to a contradiction.

2.8.3 Critical Remarks

Temporal logic is a useful tool for reasoning about liveness properties of concurrent programs, since it permits statements about future events. Lamport's model of temporal logic relies upon discrete events, which may reduce the model's usability for systems which cannot be represented this way. The use of a proof lattice greatly simplifies liveness proofs. However, this version of temporal logic has the usual disadvantage of not being expressive enough to place bounds on the time which it will take before an event will occur.

It is Lamport's contention that using dummy variables to represent control state limits their utility. He describes a strengthened version of the Owicki-Gries method that uses explicit control predicates, permitting a larger class of invariants to be proven.

2.9 Misra and Chandy

2.9.1 Overview

In [MC81], the authors present a proof technique for networks of processes which communicate via messages. The basic notation is similar to CSP [Hoa78] but uses explicit *channel* naming rather than explicit *process* naming for message passing. One channel connects exactly one pair of processes. Channels are directional. A channel is said to be *incident* on the processes it connects.

Processes are specified by a pair of assertions r and s, one corresponding to a precondition and the other to a postcondition. For a process h, $r|h|s$ specifies that assertion s holds initially, if r holds prior to a message transmission then "s holds at all times prior to and immediately following that message transmission." *Message transmission* refers to the sending or the receiving of a message by process h. Assertions are on traces (described later).

Processes are defined to be either a *sequential processes* or a *network of processes*. Sequential processes contain commands for message transmission. Networks consist of processes and their channels. Channels may exist between processes in separate networks: for example, if a channel is directed from process h_i to process h_j and h_i is within network H but h_j is external to H, then the channel is said to be *incident on H* and directed away from H. If both processes are within H then the channel is *internal to H*.

Misra and Chandy also make use of *traces*. An *external* trace of a process h at a particular point during computation is a sequence of tuples $< (C_1, v_1), \ldots, (C_n, v_n) >$ where the i^{th} message sent/received by h is along channel C_i incident on h, and the message has value v_i. An *internal* trace of a process h is also a sequence of tuples $< (C_1, v_1), \ldots, (C_n, v_n) >$, where in that computation the i^{th} message transmitted on all channels incident on or internal to h is transmitted along channel C_i with value v_i. All traces are initially null sequences. For proof purposes, traces may be considered auxiliary variables.

Notation for sequences Z, Z_1, Z_2:

Z	Length of Z
$Z_1 \underline{\propto} Z_2$	Z_1 is an initial substring of Z_2
$Z_1 \equiv Z_2$	Z_1, Z_2 are identical sequences
$Z_1 \| Z_2$	Sequence obtained by appending Z_2 to $Z1$
$<e_1, \ldots, e_n>$	Sequence having elements e_i in the given order.

Two inference rules and a theorem are used extensively in proving the examples given in [MC81]. They are:

Rule 1 (Network Composition) *Deduce the internal specification of a network H from the external specifications of its component processes h_i:*

$$\frac{r_i|h_i|s_i, \forall i}{\bigwedge_i r_i[H] \bigwedge_i s_i}$$

Rule 2 (Inductive Consequence) *Implication on precondition and implication on postcondition.*

$$\frac{(s \bigwedge r) \Rightarrow r',\ r'[h]s}{r[h]s}$$

$$\frac{r[h]s',\ s' \Rightarrow s}{r[h]s}$$

Theorem 1 (Hierarchy) *Use the external specifications of network H 's component processes h_i to produce H 's external specification.*

$$\frac{\forall i,\ r_i|h_i|s_i;\ (S \bigwedge R_0) \Rightarrow R,\ S \Rightarrow S_0}{R_0|H|S_0}$$

where R_0, S_0 are assertions on the external trace of H, and R, S denote $\bigwedge_i r_i$, $\bigwedge_i s_i$

2.9.2 Technical Discussion

The individual processes/networks must be specified as described above (i.e., for a process, give the $r|h|s$ assertions; for a network, additionally provide channel descriptions. Note that no network-wide or global variables may be used in the specifications, though individual processes may use their particular trace as an auxiliary variable. For example, a buffer process h might include the assertion

$true|h|Z_{out} \underline{\propto} Z_{in}$

which states that the sequence of messages output by a buffer is an initial subsequence of the sequence of messages it receives.

It must be shown that $r|h|s$ holds for each process/network.

- **Sequential Process**
  1. Assert that the trace of h is empty.
  2. Show that s holds initially.
  3. Assert r before and after each message transmission command.
  4. Annotate h.
  5. For each message transmission M show { pre(M) } M $\{s\}$

- **Networks**
  1. Component processes must be shown to fulfill $r|h|s$.
  2. The network assertion $R_0|H|S_0$ must be shown, where R_0, S_0 are assertions on the external trace of H. This is done using the Theorem of Hierarchy.

2.9.3 Critical Remarks

Misra and Chandy's system cannot be used to prove temporal properties directly (e.g., eventual deadlock, eventual termination). Additionally, [Bar85] point out that Misra and Chandy's reliance on traces make it difficult to describe finite input sequences, since it introduces a great deal of complication in the trace assertions.

3 System Survey

3.1 Overview

This section of the report presents the specific details of the systems survey that we have conducted. As with the methodologies, the systems survey is organized by the "short list" presented in Section 1. This list contains six of the most significant and representative systems for software specification and verification: Affirm, EHDM, FASE, HOL, Larch, and OBJ3. In addition to these six systems, four other systems that did not make our short list are reviewed. These four systems are:

- **Eves** – a verification system with emphasis on verification of security
- **Gypsy** – an early system for the specification and verification of concurrent programs, including support for code-level verification
- **Nuprl** – a verification system related to HOL but based on an alternate form of logic
- *VDM* – a foundational system for state-based specification, with some strong similarities to EHDM

We have included these additional reviews since the systems are significant, highly visible, and have some important relationships to the systems on our short list. However, in terms of the systems that shape the design of the prototype workbench, the six short-list systems have all of the features and concepts that we need.

Following the detailed system reviews, our system survey concludes with a subsection that compares the results of a detailed verification task carried out on the three systems with which we have the most direct experience – Affirm, OBJ3, and HOL. This more detailed comparison further supports our general conclusion that no single system has all the features necessary to specify and verify concurrent software in a natural and convenient manner. Hence, our workbench design embodies features from several existing systems.

3.2 Summary Comparison Table

Table 2 summarizes the results of our survey of existing, reasonably mature computer-based systems for specification and verification. Listed in the Table are those systems that made our "short list." For the most part, these systems support specification and verification of sequential programs, with a few features for the support of concurrency. An essential goal of our research is to meld features from the methodologies of Table 1 with the mature systems of Table 2, producing a usable tool.

As with the methodologies comparison table, key system features are listed leftmost table column, and the reviewed systems listed along the top row. The system features are organized into five major categories. Each of these categories is described briefly below. In the subsections which follow the tables, detailed reviews of the summarized work are presented.

In Table 2, the features under the category "Execution Support" refer to the major models of execution; the primary dichotomy in execution model is the whether the system supports fully functional execution (i.e., no side effects) versus execution with state memory (i.e., side effects *are* allowed). The "Forms of Logic" category includes the same entries as in Table 1 in Section 2 of the report. The "Checking" category indicates whether the system performs completeness and/or consistency checks of specifications. "Verification Support" includes a number of features useful in a computer-aided verification systems. Finally, the category of "Qualitative Measures" lists general features of the systems.

	Affirm	FASE	EHDM	HOL	Larch	OBJ3
Execution Support:						
Rewriting	X	X	X	X	X	X
Functional Evaluation	X	X	X	X		
State Memory		X	X			
Interpretation	X	X	X	X	X	X
Compilation		X				
Error Handling	fair	good	good		good	fair
Linkage to Low-Level PL	Pascal	Lisp	Pascal	ML	CLU	Lisp
Forms of Logic:						
Predicate Calculus	X		X		[2]	
Propositional Logic						
Quantification	X	X	X	X	X	
Temporal Logic						
Higher-Order Logic				X		
Boyer-Moore						
Equational	X	X	X	X[1]	X	X
Hoare Logic	X		X	[2]	X	
Checking:						
Consistency	X		X	X	X	
Completeness	X	X	X	X	X	
Verification Support:						
Rewriting	X	X	X	X	X	X
Tactics				X		
Induction	X			X	X	
Decision Procedure	X					
Proof Management	some		some	some		
Test Generation		X				
Qualitative Measures:						
Top-Down Support	X	X	X	X	X	X
Ease of Use:						
Specification	good	good	good	fair	good	good
Verification	good	N/A	good	good+	good	poor
Level of Support	none	good	good	good	fair	good
Level of Completeness	good	good-	good	good	poor	good

Table 2. Systems Comparisons

Notes:

1. As described below, we have implemented some extensions within the HOL system to support a form of equational logic, and we are continuing research in this area.
2. In papers on the Larch system, the authors describe the interface from the Larch equational logic to predicate logic, but it is unclear from system descriptins if the implemented system supports machine-based predicate logic.

3.3 Affirm

3.3.1 Overview

This section presents a overview of the capabilities of the Affirm system from ISI [TE81] as a tool for the development of abstract specifications and their verification. Although a later version exists, Affirm-85 from General Electric Company [Kem86], this review covers the version developed at ISI.

Affirm is an interactive system for the formal specification and verification of abstract data types (ADTs) and programs. It uses an algebraic specification language very similar to that presented in [GHM78]. Using this language the user can hierarchically define ADTs and a set of equational rewrite rules for them. For each ADT the user need not completely define the rewrite rules of the operations but need only define the type signatures. This allows the user to incrementally develop the specification, leaving implementation decisions until a later date.

While the specification is being developed, the user can apply some tools provided by Affirm to test the specification. Affirm will automatically check these rules for consistency using the incremental Knuth-Bendix algorithm. It also provides methods for testing completeness using the Guttag-Horning recognizable sufficient completeness test [GH78]. And, it allows testing of simple cases by actually executing the specifications.

Affirm also supports a PASCAL-like statement language for the algorithmic implementation of operations in the ADTs. This language differs from PASCAL by requiring all data types, and operations performed on them, be defined as ADTs in Affirm. It also has a few additional constructs that allow the user to add assertions about these types throughout the code. Using these assertions, Affirm can automatically generate verification conditions (propositions) ing the inductive assertion method [Flo67] or proving the code satisfies the assertions.

Using the algebraic specification language, the user can also generate equational propositions. Once the system has propositions, either user generated or as verification conditions, the user can use the underlying logic, based on propositional calculus, along with instantiation, case analysis, skolemization, normalization and unification, to prove these. The structure of these proofs in the system is maintained as a directed acyclic graph with subgraphs corresponding to the proofs of the propositions. The user can change the graph by adding, deleting or modifying nodes, with the ultimate goal of transforming the propositions, by rewrite rules, to the constant TRUE. If this transformation fails, Affirm will try to inform the user why it failed and let the user correct the proposition.

3.3.2 Execution and Rapid Prototype Support

As stated above, since Affirm is a rewrite system, any of the rules can be executed. The user can simply type the keyword **eval** followed by the expression to evaluate, and Affirm will determine the result.

This evaluation method, and the ability to develop partial specifications along with hierarchical development, make Affirm ideal as a rapid prototyping system. The user, when developing a system can mix specifications of interfaces that have yet to be implemented, with actual Pascal-like code. At all levels of development the user can define partial operations and test to make sure they satisfy the requirements. If they don't satisfy the requirements they can be modified immediately while Affirm keeps track of all the proof obligations and consequences of the change.

3.3.3 Abstraction Mechanisms

To demonstrate the Affirm abstraction Mechanisms we give a sample specification. The following is the part of the Affirm version of the abstract specification of the Object ADT for the Low Water-Mark System as given originally in [Kem86]:

```
type Object;

    { *** Declarations *** }

    needs types ElemType, SecurityLevel;

    declare x,x1:Object;
    declare e:ElemType;
    declare s,s1:SecurityLevel;

    { *** Syntax of operations *** }

    interfaces Write(e,x,s), Reset(x,s),Initial:Object;
    interface ObjLevel(x) : SecurityLevel;
    interface Read(x,s) : ElemType;
    interface ObjectInduction(x):Boolean;

    { *** Semantics *** }
    axioms
      Read(Write(e,x,s1),s) ==
        if s1 <= ObjLevel(x)
        then if s1 <= s then e else Null
        else Read(x,s),
      Read(Reset(x,s1),s) ==
        if s1 <= ObjLevel(x)
        then Null
        else Read(x,s),
      Read(Initial,s) == Null;

    define x=x1 ==
      (Read(x,syshi) = Read(x1,syshi) and
       ObjLevel(x) = ObjLevel(x1));

    schema
      ObjectInduction(x) ==
        cases(Prop(Initial),
      all x1,e,s(IH(x1) imp Prop(Write(e,x1,s))),
      all x1,s (IH(x1) imp Prop(Reset(x1,s))));

end {Object};

type ElemType;

    { *** Declarations *** }
    declare reflexive,dummy:ElemType;
    interface Null : ElemType;
```

```
        { *** Semantics *** }
        axiom reflexive = reflexive = = TRUE;

    end {ElemType};
```

The declaration section defines needed (i.e., imported objects) and variables used in the axioms. Here for the type Object we use ElemType and SecurityLevel ADTs. This mechanism allows the user to hierarchically define ADTs, where a lower level ADT may not be fully specified. The ElemType ADR is a good example of an ADT which has declarations but no semantics for the NULL operation. The syntax section defines the interface to each of the ADT's Operations.

In the Object ADT there are several operations, to permit the user to Read, Write and Reset the Object, and to determine the security level of the opject or initialize a new object. The semantics section defines the meaning of the ADT in terms of algebraic axioms about its operations. In Affirm, the axioms are specified using either the keyword axiom, informing the system to automatically apply an axiom whenever applicable in the course of a reduction, or the keyword define, informing the system that the user will specifically invoke all applications manually. This permits the user to control the flow of a proof development at the level desired. Affirm also allows the specification of schemas for user defined proof arguments.

3.3.4 Forms of Logic Supported

The Affirm system support the algebraic specification language of Guttag, Horowitz and Musser. Any propositions in the specification are simply Bolean-Valued expressions of the form

$$\mathbf{all} x_1, \ldots, x_n (\mathbf{some} y_1, \ldots, y_m (P(x_1, \ldots, x_n, y_1, \ldots, y_m)))$$

where all and some are universal and existential quantifiers.

In Affirm, all logical expressions are translated into a simplified internal IF-THEN-ELSE form and skolemized.

Also supplied in the system is an operator similar to the weakest liberal precondition [Dij76] to generate verification conditions for the Pascal-like language code segments.

3.3.5 Verification and Theorem Proving Support

The Affirm system also supports an interactive proof development environment. This is neither a proof checker nor a proof finder.

Using the rewrite rules of the data types along with the rules of propositional logic, this evironment is used to "simplify" the current proposition down to the logical constant TRUE.

The environment supplies some built in functions to allow the user to invoke definitions, apply unproven lemmas, split the proposition into subgoals, and employ data type Induction schemas that have been defined by the user.

To manipulate the proof, the system maintains the internal tree structure of the proof and allows the user to traverse it at any time to prove a current leaf or retry an existing node.

3.3.6 Specification Checking – Completeness and Soundness

Affirm automatically checks for axiom inconsistencies and applies an incremental Knuth-Bendix convergence process [KB70] to guarantee that all axioms maintain the Church-Rosser unique termination property. The user can also apply a completeness test to ensure that the axiom set is sufficiently complete using the algorithm presented in [Gut75] and [GH78]. This test also categorizes the operations into the constructors, modifiers, and selectors, which information can be useful in other aspects of ADT verification.

3.3.7 Formal system Basis – Completeness, Consistency and Soundness

As mentioned before the system will check all specifications for consistency and allows the user to determine sufficeient completeness. Based on the underlying logic of rewrite rules and the built in axiomatization of integers and booleans the system seems complete.

3.3.8 Qualitative Measures

The system is very useful and easy to manipulate the proofs and specifications of the system. It does not support any built in induction methodology other than user defined schemas. It also does not support higher order logic although it gives a large amount of useful utilities and built in techniques for the logic it does support.

3.3.9 An Example

The following is the Affirm version of the abstract specification of the Low Water-Mark System as given originally in [Kem86]:

```
type SecurityLevel;

    { *** Declarations *** }

    declare sl,sl1:SecurityLevel;

    { *** Syntax of operations *** }
    interfaces sl <= sl1, sl >= sl1 : Boolean;
    interface syshi: SecurityLevel;

    { *** Semantics *** }
    axioms
      sl <= sl == TRUE,
      sl <= syshi == TRUE;

    axiom
      sl >= sl1 == sl1 <= sl;

end {SecurityLevel};

type ElemType;

    { *** Declarations *** }
    declare reflexive,dummy:ElemType;

    { *** Semantics *** }
    interface Null : ElemType;
    axiom reflexive = reflexive = = TRUE;
```

```
end {ElemType};

type Object;

    { *** Declarations *** }
    needs types ElemType, SecurityLevel;

    declare x,x1:Object;
    declare e:ElemType;
    declare s,s1:SecurityLevel;

    { *** Syntax of operations *** }
    interfaces Write(e,x,s), Reset(x,s),Initial:Object;
    interface ObjLevel(x) : SecurityLevel;
    interface Read(x,s) : ElemType;
    interface ObjectInduction(x):Boolean;

    { *** Semantics *** }
    axioms
      ObjLevel(Write(e,x,s)) ==
        if s <= ObjLevel(x)
        then s
        else ObjLevel(x),
      ObjLevel(Reset(x,s)) ==
        if s <= ObjLevel(x)
        then syshi
        else ObjLevel(x),
      ObjLevel(Initial) = syshi;

    axioms
      Read(Write(e,x,s1),s) ==
        if s1 <= ObjLevel(x)
        then if s1 <= s
     then e
     else Null
        else Read(x,s),
      Read(Reset(x,s1),s) ==
        if s1 <= ObjLevel(x)
        then Null
        else Read(x,s),
      Read(Initial,s) == Null;

   define x=x1 ==
     (Read(x,syshi) = Read(x1,syshi) and
      ObjLevel(x) = ObjLevel(x1));

   schema
     ObjectInduction(x) ==
       cases(Prop(Initial),
     all x1,e,s(IH(x1) imp Prop(Write(e,x1,s))),
     all x1,s (IH(x1) imp Prop(Reset(x1,s))));
```

```
end {Object};

print proof;
theorem StarPreserved,      ~(s <= ObjLevel(x)) imp Write(e, x, s) = x;

proof tree:
72:!  StarPreserved
          invoke =                            /* Invoke the '=' axiom
72:   11  cases                               /* And then apply case analysis
72:-> (proven!)


97 U: print proof;
theorem SSCpreserved,      ~(ObjLevel(x) <= s) imp Read(x, s) = Null;

proof tree:
61:!->SSCpreserved
          employ ObjectInduction(x)           /* Here we employ user defined
       Initial:                               /*  induction schema
          immediate
61:    Write:                                 /* For the Write case apply case
          2   cases                           /*  analysis then invoke Induction
63:       4   invoke IH                       /*  hypothesis. Affirm then can
64:       5   put s'' = s search            /*  'search' for the proper
64:       (proven!)                           /*  instantiation
66:    Reset: SSCpreserved                  /* Same for Read
          3   cases
67:       7   invoke IH
68:       8   put s'' = s search
68:       (proven!)
```

3.3.10 Critical Remarks

In a single environment Affirm permits the user to algebraically specify ADTs, test them for consistency, completeness and execute test cases, prove propositions by transforming them using the underlying logic and rewrite rules, and verify PASCAL-like programs using the inductive assertion method.

Affirm lacks any form of parameterized typing. The current version of the system uses the "text-editor" method of parameterization. In this particular example, the lack of parameterization was not a problem. However, in examples with an ADT applied to several different types (e.g., bits, bytes, words, and n-bit values etc.) the user would have to enter the specification for each separately. If these specifications had interfaces with identical names, the user would then be prompted for the type signature at every use of each unqualified interface. Furthermore, the verification of each instance of the ADT would need to be separately verified.

3.4 Enhanced HDM

3.4.1 Overview

This section is a brief introduction to Enhanced HDM (Hierarchical Development Methodology), an interactive system for the composition and analysis of formal specifications and programs[vHR90][Rus90]. It has been under continuous development at SRI since 1983; the National Computer Security Center and NASA-Langley Research Center have been its principal sponsors. EHDM shares some concepts with SRI's previous methodology, HDM (a.k.a. old HDM), for example the capability of expressing specifications for operations (as model-based specifications) and a flow analyzer that determines if a specification is consistent with a model of multilevel security. However, Enhanced HDM is more expressive and has substantially more proof support.

The unique features of Enhanced HDM are the expressiveness of its specification language (Revised Special), certain aspects of the approach to mechanized theorem proving, its unique approach to permit the reasoning about expressions that intermix declarative specifications and imperative statements, and the support for verifying security properties of a specification.

Revised Special encourages a style of specification whereby a user can proceed from abstract specifications through more detailed specifications, the process terminating with programs written in a small but unimplemented subset of Pascal. Each step in the process entails the formulation of decisions that are verified with respect to previous decisions. EHDM does not provide support for the verification of executable and optimized programs. A system cannot be said to be verified unless executable code is verified. However, substantial assurance about a system is obtained by verifying specifications, and abstract algorithms that are easily converted into programs. EHDM encourages a user to verify properties of a system's abstract specifications and of implementation decisions described as abstract algorithms in a language that has many features of Pascal and Ada.

The justification for a system like EHDM is empirical evidence that most of the undetected (and difficult to fix) errors in delivered systems occur in what is perceived as the design stages of development. Using EHDM, the design of a system can be expressed as specifications and abstract algorithms.

Among the unique features of Revised Special are:

- Parameterizable modules, through which generic modules can be specified, verified and appropriately used.
- Support for second order predicate calculus, which, among other things, allows the user to define and reason about induction schema.
- Support for program operations as first-class objects, which allows the user to declare, specify, implement and verify operations at convenient points in the development of a system.

The goal for the EHDM theorem prover was to produce a fast and predictable response. To that end, EHDM's theorem prover is based on a collection of decision procedures. A decision procedure determine whether a predicate within its domain of decidability is true or false. EHDM includes decision procedures for propositional logic, equality over uninterpreted functions, and Presburger arithmetic (essentially linear arithmetic). However, the logic of Revised Special is much more powerful (i.e., first and second order predicate calculus), and is essential to express properties of complex software systems. In order to prepare a formula of Revised Special for proof by the theorem prover, the user can exercise combinations of the following options: (1) Cite (other formulas) premises to be used in the proof, (2) Provide substitutions for certain quantified variables, and (3) Accept the substitutions generated by EHDM's unification and resolution packages. This preprocessing will transform the Revised Special formulas into formulas in the domain of the theorem prover's decision procedures.

EHDM includes support for the verification of system security properties, mainly attempting to identify covert channels that involve the transfer of information through the operating system state. The MLS tool mechanizes the verification of Revised Special specifications, restricted to a particular style, with respect to an information flow model of security. The purpose is to demonstrate that according to the specifications there is no flow of information from a secure object to a secure object of a lower level.[1] It should be noted that use of the MLS tool cannot guarantee the absolute security of a system. In particular, the tool does not establish the correctness of executable code; furthermore, the specifications it analyzes are relatively abstract, precluding any representation decisions that would allow the multiplexing of an object among different security levels. However, the tool can detect some insecurities that originate early in the design process, often subtle ones that would be difficult to reveal by inspection of executable code.

A rugged version of the Enhanced HDM system has been completed and documented, and is available.

The remaining sections of this overview describe:

- A scenario for use of the Enhanced HDM system.
- The Revised Special specification language.
- The Pascal/Ada subset that constitutes the language for expressing verified programs.
- A hierarchical development.
- The theorem prover and how it is used to verify properties of specifications, operations, and abstract programs.
- The MLS tool.
- The implementation of Enhanced HDM.
- A summary of available documentation.
- A description of SRI's research in verification and software methodology that led to Enhanced HDM. Applications of Enhanced HDM and its predecessors are also summarized.
- Strengths and weaknesses of the current system.

3.4.2 The Enhanced HDM Methodology

As conceived by Floyd in the mid-60s, program verification is concerned with establishing the consistency between a program and its specification, the specification being a pair of assertions called the pre- and post- conditions. These conditions are usually expressed in predicate calculus or some other formal logic. Any practical mechanization of Floyd's method requires all loops of the program to be RcutS by assertions. The assertions are expressed in the same logic as the program's specifications. The program annotated by assertions can be viewed as consisting of paths, a path being defined as a sequence of program statements bracketed by assertions. Each path defines what is called a verification condition, a predicate which if true indicates that the path is correct with respect to its bracketing assertions. The program is verified if all of its verification conditions are verified.[2]

[1] As described later, the model is not restricted to a linear ordering of security levels.

[2] To be precise, this process only establishes partial correctness: the program satisfies its assertions whenever it terminates. An extension of the verification condition paradigm can be employed to establish termination.

The Floyd paradigm has been the basis for most verification systems developed over the past 15 years. However, various extensions to this paradigm have been developed and implemented, mostly in an attempt to solve the problems associated with verifying large programs that, among other things, consist of many subprograms. These extensions have led to what are called methodologies. In one form or another, all of these methodologies offer the following features:

- The capability to describe a system's behavior in terms of abstract entities, thus leading to specifications that capture the system's behavior without reference to low-level implementation concerns. The implementation concerns one would hope to avoid (or at least postpone) include concrete data representations and executable code the verification of which would have to involve machine-specific concerns. The expectation, of course, is that the abstract specifications will be easier to produce, understand and debug than ones involving implementation details.
- Hierarchical development, through which one proceeds in small steps from the abstract specifications to a concrete description of the system, such as optimized executable code. It is through hierarchical development that the proof of a large system can be considered as a collection of relatively small proofs.
- Reusability, through which generic specifications and implementations are verified and subsequently used in different applications.

Several methodologies, most notably Gypsy and the Stanford Ada-Anna methodology, started with a hierarchical programming language that was subsequently extended with a notation for specifications, typically first- order logic. Others, most notably, Affirm, FDM and the two HDMs, started with a specification language and emphasize reasoning about specifications. In some cases, the specification language was linked to a programming language. In some cases (OBJ) there is a single language (aka, a wide spectrum language) serving for both specifications and programs.

The advantages of the programming language approach are clear: executable programs are verified, and the user need learn only one language. On the other hand, it might be easier to incorporate features in support of verification and software engineering into a specification language, since it is not necessary to consider executability when it is decided what features to include. Rather than debating the merits of each approach, we present a brief scenario of how Enhanced HDM would be used in the design and verification of a system.

In developing a system with Enhanced HDM, a user might first express the overall requirements of his system, probably using one or more modules which will constitute the top-level of the system description. At an abstract level, a system is specified in terms of a collection of uninterpreted functions. Some of these functions have an obvious connection with callable programs (e.g., clos_a_file), while others are primarily present for the purpose of specification, i.e., do not correspond to any concrete representation or callable program. Examples of the latter would be functions that record the history_of_sent_messages or the set_of_failed_processors. One can call these ghost functions.

The heart of specification consists of axioms and definitions, the purpose of which are to express constraints on the functions. The constraints are to be enforced ultimately by the implementation. The developer will usually enclose these specifications in one or more modules. In constructing the specification, he also might use previously specified modules – most typically parameterized modules instantiated for current needs. For example, he might find use for a sets module, a holder for elements without regard to any ordering.

Once the top level specification is complete, the user can pose and verify properties of the specification. The goals of posing such properties are to debug the specifications and to extend the specification with even more abstract formulas. A property can take the form of a hypothesized response to real or symbolic inputs, or can be a general theorems of the specification.

Next the user will want to take steps toward an implementation by making decisions on how the functions of the top level are to be represented. First, he defines one or more next level modules, the purpose of which are to provide functions that will serve as building blocks to represent the top level functions. Then he creates the representation as a collection of mapping functions, one for each upper level function. The axioms of the top level are then verified to be theorems of the modules of the next level. The mappings will often be expressed as definitions and are quite similar to Affirm's equational implementation. Using the power of the specification language to express mappings often leads to more concise descriptions than is possible with a programming language. For example, it is usually easier to assert the existence of an element that satisfies a certain condition than to exhibit a program that explicitly identifies the element.

At any point in the development process, operations can be introduced, an operation producing a change (i.e., a side effect) to what are called state variables. The notation fo. the specification of operations is that of Hoare triples (also called Hoare sentences). An operation ca.. be expanded to a sequence of operations or, ultimately, into programs using our subset of Pascal. As we will discuss below, having operations as first-class objects in the specification language allows the user to pose lemmas about operations that will be helpful in structuring a proof.

The hierarchical development can continue until a level is reached, the specifications of which the developer is prepared to accept as correct without verification. If it is to be claimed that the implementation is verified the verification should be carried down to a level consisting of abstract programs that can be easily (with minimal chance of error) converted into executable code. However, there is no obligation on the user's part to proceed this far.

At any point in the hierarchical development process, the user can decide that one or more interfaces are to be checked with respect to the multilevel security model. Typically, an interface will be that of a resource manager (e.g., a file system) whose purpose is to enforce multilevel security for a class of objects. The user specifies two modules. One of these is in the stylized form acceptable to the MLS tool: operations on a collection of objects, each object being associated with a constant security level. The other specification can be considered as an optimized version of the first specification, allowing for objects to be multiplexed among different security levels. The first module is verified with using the MLS tool. Next, the first and second modules are shown to have identical external behavior. It is the second of the modules that is subsequently subjected to hierarchical refinement.

3.4.3 Revised Special Specification Language

Revised Special is used for expressing all specifications and proofs. Revised Special is based on multi-sorted predicate calculus and is intended to support both an axiomatic and constructive (aka model-based) style of specification. The former (also called property-oriented specification) facilitate the expression of design-level decisions; in particular, property-oriented specifications can express partial properties. The constructive style, being essentially state machine specifications, is often preferred by engineers.

Specifications are structured into modules with explicit import and export lists. Typically, a module will correspond to what is called a theory, as it expresses many (although not all) interesting properties of a mathematical theory (such as sets). Some modules, such as sets, serve primarily to provide abstract functions for the purpose of specification; they are never implemented. Other modules, (such as buffers), are refined down to concrete computational objects; see below. Module specifications may be parameterized by types and constants (including function constants), for uninterpreted – so that generic theorems can be stated and verified. Usually, the parameter will be associated with the elements that will be stored in an object (such as in stack), the elements used to index an object (such as the indices of an array), and functions defined on the indexing elements (such as ¡). Also, a parameter could correspond to an uninterpreted constant such as the length of a stack or to an error return, such as the overflow response when push is called on a full stack. Any proof about a parameterized module will be valid in all instantiations of the module. Actual parameters must, of course, match the type of their corresponding formal parameters, and in many situations it is necessary that they should satisfy additional semantic constraints (called assumptions) expressed in the body of the module specification.

Inside a module, one usually declares types, either uninterpreted types or subtypes of existing types. A function is declared by giving its name and signature. The heart of the specification gives meaning to the functions declared in the parameter section or in the main body. Through the use of the Lambda Calculus, a function (without a name) can be specified by giving it a definition which can be recursive. The other way a function can be specified is in terms of axioms, an axiom typically involving more than one function. When the theory in question is an abstract data type, the axioms will have the appearance of rewrite rules, although at present there is not special-purpose mechanization of rewrite rules. Predicate calculus can also be used in axioms. An important feature of Revised Special is is quantification over functions. This second order capability is usually used to define induction schema, but it has also been used to express abstract system requirements. For example, a specification of a Crypto system can include the assertion that there does not exist a function that can decrypt a message unless the key is known.

As indicated above, if the module contains parameters, the specification body can contain what are called assumptions on the parameters. The assumptions are properties of the parameters that must be verified to hold in any instantiation of the module. For example, an assumption might require that a parameter that indexes into an object satisfies the properties of a totally ordered set. If the instantiation associates character sequences with this indexing parameter, it must be verified that the set of character sequences is totally ordered. The proof requires axioms about lexical ordering.

The typical abstract data type specification is in terms of functions that have as its arguments one or more abstract types. Using Guttag's terms, the functions can be constructors, modifiers, or selectors. The constructors are used to generate all instances of the abstract types associated with the module. The general style is that of functional programming, a constructor taking an instance of an abstract type (among other arguments) and returning a possibly difference instance. Modifiers also return an instance of the type, but cannot generate any instances different from those producible by constructors. Selectors return a concrete object associated with particular instance of the abstract type (e.g., the top element of a stack).

Different from the functional programming model, most programming languages are based on the von Neumann computational model. To provide a link to the von Neumann model, Revised Special provides operations. Functions or variables can be declared to be state objects. An operation, similar to a function, has a signature that declares the types of its input and output arguments. However, different from a function, an operation can be declared to have a side effect on one or more state objects. An operation op is specified in terms of a Hoare triple of the form {P}op{Q}: if assertion P is true prior to op being called, Q will be true after the call. P and Q will usually be in terms of state objects, P characterizing the state expected prior to the call, and Q the state following the call. A Hoare triple can appear anywhere a boolean expression is expected in the specification. Operations can appear anywhere in a module specification, but typically are used as the development process evolves to the consideration of program-level concepts. One can use Revised Special to specify the operations of a conventional von Neumann language; in such a specification, conventional program variables will be declared as state objects, and the assignment statement (for example) is then specified in terms of its effect on particular state objects.

A module specification can also pose lemmas and theorems, properties that when verified are true of a specification. The approach to verifying these properties is discussed in a later section, but for now it suffices to say that theorems usually express fundamental properties of the specification and lemmas are employed to facilitate the verification of theorems. One interesting use of Revised Special is to pose theorems about sequence of operations that are verified with respect to specifications for the individual operations. This application is discussed in the next section.

3.4.4 Program Verification in Enhanced HDM

The user can write and verify programs written in a very small subset of Pascal (which is also a subset of Ada), the subset supporting the following statement types: assignment, if- then-else, while and repeat-until. Program variables are declared as Revised Special state variables, and procedure calls correspond to Revised Special operations with actuals substituted for formal arguments. Rather than calling this language a subset of Pascal, it has been suggested that it be called an abstract programming language; the possibility of making the abstract programming language a part of Revised Special is discussed later. Currently, there is no compiler for our subset of Pascal.

EHDM uses the Hoare-triple notation to express that a program is to be verified with respect to pre- and post-conditions. To facilitate its proof, a program can be decomposed into sections. For example, the body of a while loop is considered as a section separate from the initialization of variables that might precede the while loop. The user can separately prove the body and initialization, and then indicate that the proof of the program is to be accomplished by citing the proofs of the separate sections. The advantage of this approach are in allowing the user to divide up his program into sections as he sees best and to split off as lemmas complex properties.

3.4.5 Hierarchical Development

Hierarchical development involves the representation of a module in terms of lower level modules and the verification of the representation.

The representation is itself a module that considers the following. The user indicates that a module is to be represented in terms of one or more other modules; more generally, a set of modules can be represented in terms of a different set, the two sets perhaps overlapping. Each abstract type in the upper module set is given a representation in terms of lower module types. It might be necessary to define what equality of the upper level types means in terms of the lower level ones. For example, two stacks each represented in terms of an array and an integer pointer can be declared as equal if their pointers are equal and the arrays are equal in positions up to the value of the pointer; the remaining positions in the array are of no consequence. Finally, the representation contains a definition for each function of the upper level.

The verification of the representation is conceptually straightforward. Each of the axioms of the upper level becomes a theorem to be proved of the module set of the representation and the lower level modules. Furthermore, the user is obliged to state and verify particular proofs about the equality relation he defined, namely transitivity, reflexivity, symmetry and substitutively with respect to all upper level functions.

3.4.6 The Theorem Prover and How it is Used

The heart of the theorem prover is a collection of decision procedures for propositional logic, equality over uninterpreted functions and linear arithmetic. The linear arithmetic package reasons about equalities or inequalities that involve addition and multiplication by constants. A very limited nonlinear capability (multiplication by variable values) is supported. The key feature of the decision procedure set is that it decides automatically whether a formula in its domain is a theorem or not a theorem.

In carrying out a proof of a theorem or lemma, the user will cite as premises axioms of the specification; definitions are automatically expanded, actuals being substituted for formal. In citing the premises, the user is indicating his expectation that his theorem can be proved with respect to those axioms.

It is not possible to write specifications for real systems if one is restricted to the domains provided by the decision procedure. A Revised Special specification will usually consist of formulas in first and second order predicate calculus. In the absence of any assistance from the user, the prover will replace all variables and functions of the theorem and cited premises with arbitrary constants or constant functions. Although what results is in the domain of the decision procedure, it is usually not a theorem. What is needed to produce a provable theorem are particular substitutions for the variables and functions.

Substitutable variables are those that are universally quantified in the premises and existentially qualified in the conclusion; the situation is complicated by the interaction of universally and existentially quantified variables, but still relatively straightforward. The user is allowed to provide substitutions for substitutions for substitutable variables, doing so upon citing premises or the theorem to be proved.

Also included in the prover is a special package called the Hoare Sentence Prover (HSP). The HSP transforms expressions involving Hoare triples and user-provided substitutions into formulas of the underlying decision procedures. What the HSP generates is very close to conventional verification conditions, although it also allows for user-cited premises.

The bare-bones prover returns PROVED or UNPROVED; it does not return RfalseS since the theorem in question might be true subject to appropriate premises and substitutions. To assist the user in debugging proofs, a number of tools are provided.

The show variables feature identifies the substitutable variables and their dependencies. The proof trace indicates the theorem under proof and the premises after the substitutions have been made. The proof debugging aid allows the user to play with a failed proof. In particular, he can explore the theorem under proof and each of the premises, requesting that the proof be carried out assuming that particular expressions or subexpressions are assumed to be true. For example, he can request that the antecedent P of a premise (P implies Q) be assumed true. This way, he can determine if his ultimate proof would have succeeded if he was successful in proving P. If the answer is yes, he could then focus his attention on why the attempted proof of P was unsuccessful. Note that the proof debugging aid does not allow the user to modify substitutions; we will have more to say later about this deficiency. The proof chain checker assures that proofs have been provided for everything that the user has declared as provable, i.e., lemmas, theorems, assumptions on parameters for instantiated modulus, and axioms in modules that are subject to hierarchical refinement.

3.4.7 The MLS Tool

view char

EHDM includes support for multilevel security. The MLS tool takes as input a Revised Special module subject to a few restrictions and produces a new module containing theorems to be verified. If these theorems are successfully proved, perhaps requiring the citing of premises and substitution of variables, then it is claimed that the original module is multilevel secure.

SRI's model of multilevel security is addressed in [FLR77] (a preliminary description of the model) and [Rus84] (a description appropriate to Enhanced HDM). The model assumes a collection of security levels. Security levels are assumed to contain two components: a clearance level, taken from a totally ordered set, and a list of categories, lists ordered according to inclusion. The set of clearances typically includes the levels (in order) UNCLASSIFIED, CONFIDENTIAL, SECRET AND TOP SECRET. The set of categories includes what are called need to know rights, typically NATO, ATOMIC, etc. The set of security levels forms a lattice with TOP SECRET and all rights the topmost element and UNCLASSIFIED and no rights at the bottom. For any pair of security levels s11 and sl2 it is either the case that sl1 ¡ = sl2 or NOT (sl1 ¡= sl2), the latter corresponding to sl1J¿ sl2 or sl1 and sl2 being Rincomparable.S

Furthermore, the model assumes a collection of users, each of whom is assigned a security level. Users can invoke operations, that can cause a state change and return a value. It is assumed that users are Rsharing a systemS by allowing arbitrary interleavings of invocation of operations. Assume an arbitrary sequence of operation invocations S terminated by user U1 invoking an operation; let the value returned to U1 be V1. Consider another sequence S2, which is S with the removal of all operations invoked by users whose security levels sl2 satisfy NOT(sl2 ¡= sl1). We say the collection of operations is a multilevel secure system if the sequences S1 and S2 yield the same value to U1. The model is conceptually simple. It says that what a user can obtain from a system cannot be influenced by users whose security level is not less than or equal to his.

The SRI model cannot be mechanized directly, as it implies the need to carry out an induction over arbitrary sequences of operation invocations. A trick, similar to that developed by Floyd for program verification, is used to avoid the need for induction. The trick involves introducing objects, each of which is assigned a security level, and considers the security of operations individually. Considered abstractly, each operation is viewed in terms of the invocation level (assumed to be sl1), the objects that obtain new values as a result of the invocation, and the objects that impact the value returned to the users or those that impact the objects that obtain new values. Then, an operation is said to be multilevel security if

1. The value returned to the user depends on objects whose security levels sl2 satisfy sl2 ¡= sl1.
2. The objects that acquire new values are at security level sl2 such that sl1 ¡= sl2.
3. Consider an object at security level sl2 that acquires a new value dependent on the value of an object at security level sl1. It is required that sl1 ¡= sl2.

The MLS tools assumes a Revised Special specification where objects are partitioned accordingly to security level and where each operation is specified, essentially using a form of Hoare triples. For such a specification, the MLS tool produces three sets of theorems for each specified operation, in correspondence with the three conditions above. These theorems are contained in a module that is subject to verification similar to any theorem expressed in Revised Special. Usually, the theorems are quite simple – actually trivial, reflecting the rather simple property being verified.

The MLS tool will find application in the analysis of secure resource managers, a subsystem responsible for managing a collection of objects associated with a security level. Access to the objects is provided through a collection of operations. A secure operating system will have a number of such managers, such as a file system, a directory system, a virtual memory system, etc.

The tool assumes that each object is assigned a security level that is never changed (the tranquility principle). In practice, then, only abstract specifications can be modeled by the tool. Concrete specifications that allow an object to be multiplexed among different security levels are excluded. A buffer that is sanitized after being released by a user cannot be modeled.

Access violations are easily checked, whereby a user gains control over an object for arbitrary reads and writes. However, the tool also reveals subtle security violations that result in information flow at a relatively low rate. These flows are said to be associated with covert channels, and typically arise from operations returning error conditions. For example, an object could be locked by a user, thus denying other users access to it. In the process, information could flow in violation of the model if the locking is carried out by a user whose security level is higher than those attempting subsequent access to the object.

The MLS has the potential of exposing significant security flaws in the specifications of real systems. However, it is emphasized that the MLS tool cannot establish the security of a system. It does not address program errors. Furthermore, it only handles design decisions that are quite abstract.

3.4.8 The Implementation of Enhanced HDM

After starting up the verification system, the user with a conventional glass teletype will be talking to the EMACS editor throughout most of his session. He is free to create module specifications, each of which will occupy a file, or to retrieve some existing module. The conventional EMACS commands are available to him. Once a module is ready to be processed, the user will call on commands to parse a module, typecheck it, and to prove identified formulas. Error messages are displayed in a separate window and, where relevant, the cursor will appear near the source of the error. Version control assures that the most recent versions of modules will be used.

3.4.9 Conclusions

We view as a success the effort on Revised Special. It integrates what are recognized as the seminal ideas of specification language technology: module parameters with semantic constraints, functional in addition to operational (state-based) specifications, support for lambda definitions together with general predicate calculus, support for hierarchical refinement and support for higher-order logic. Revised Special provides features that enhance the expressiveness and reusability of specifications. Moreover, in addition to a specification language, it is a first attempt at a proof justification language. The primary weakness of Revised Special is the absence of a formal semantics (currently under development), particularly important since it combines a number of different logics. Also, not all of the features of the language are currently implemented, the language could benefit form additional predefined types (such as arrays and sequences), and additional syntactic sugar could be provided.

The goal regarding verification of code was to improve on the classical verification condition approach. A premise for the EHDM approach was the belief that users find it difficult to reason at the level of verification conditions because much of the structure of the program and of the specifications is lost when verification conditions are generated. Particularly difficult is the posing of lemmas that have a clear relationship to the program being proved; it is such lemmas that are the most powerful and the most appropriate in explaining a proof. The EHDM approach to code verification involves reasoning in terms of Hoare sentences. The program being proved together with its specifications are specified as a Hoare sentence, as are the sections of the program. Thus, the user can decompose his program as he sees best to facilitate its proof, perhaps posing lemmas as he proceeds. He then describes the proof of the program in terms of its pieces, the Hoare Sentence Prover assuring that the proof is valid. We believe that the basic idea of reasoning at the level of Hoare sentences is attractive, although the EHDM implementation does not yet provide adequate proof support. In particular, the assembling of a proof of a program from proofs of its components could be automated. Moreover, the construction of a proof of a program section often requires the user to consider numerous cases simultaneously. These cases usually correspond to paths in the programs and would yield distinct verification conditions using the classical approach. We anticipate an approach that integrates the concept of verification conditions with reasoning at the level of Hoare sentences. This approach would benefit the user by allowing him to consider the cases individually at first, and subsequently to assemble them together. No support exists for this at present.

The goal for the Theorem Prover was to move towards a Rman-machine symbiosis,S the user describing his proof and the machine checking it. Towards this goal, SRI has developed a collection of decision procedures together with a preprocessor for predicate logic that handles quantified variables and functions. It is, then, the user's responsibility to cite lemmas needed in his proof and to provide substitutions for free variables such that the formulas given to the decision procedures are indeed theorems. The underlying decision procedures are reasonably fast and the user is provided assistance in identifying the bound and substitutable variables. However, users have found that the construction of proofs can be somewhat more difficult than it should be, mostly because of the absence of automated support for the construction of proofs. A formula to be verified often consists of conjunctions in the conclusion, disjunctions in the hypotheses, implications in the hypotheses, and cases arising from case or if-then-else statements. In the EHDM system, the user must consider the formula in toto, citing axioms and producing substitutions that will handle all cases. Or, if the formula is too large and complex to consider in one shot, he can break it down by hand, proving the individual cases, and then describing a proof that assembles the individual pieces into a proof of the original formula. It would be preferable to have mechanical support for proof decomposition, particularly since there are several interactive theorem provers (e.g., HOL) that already provide it. Furthermore, it is difficult and often tedious to create and input the substitutions for free variables. Some mechanical support here would be very desirable. An additional weakness of the theorem prover is the impossibility of extending the decision procedures with use-supplied axioms. For example, it would be desirable to extend the decision procedures about arithmetic with facts about multiplication and division over variables, leaving intact the core decision procedure.

As in old Special, the MLS tool handles specifications (called MLS specifications) that associate a constant security level with objects. Several of the flaws in the previous tool have been avoided, particularly those related to nondeterministic specifications. The one weakness of the MLS tool relates to the absence of mechanical support for relating an MLS specification to the rest of the system under design. It is desirable to verify that an MLS specification has the same external behavior as one with a more concrete representation; the verification system should require such a proof and provide support for it.

The interface to Enhanced HDM, through EMACS and popup mouse accessible menus, is reasonably adequate. Ultimately, it would be desirable for the communication between the user and the theorem to be of a higher bandwidth. For example, substitutions for free variables could be effected by the user pointing to expressions.

3.5 FASE

3.5.1 Overview

The FASE system (Final Algebra Specification and Execution) was developed at Illinois by Samuel Kamin, Myla Archer, and Stanley Jefferson [Kam83, KJA83, KA84, JK86] FASE specifications are written and executed using a final algebra semantics. Final algebra specifications are similar to initial algebra specifications typified by systems such as OBJ, consist of an operation signature and operation definition rules. Final algebra operation definitions may—but are not necessarily—given in a restricted form of rewrite rules. In contrast to initial algebraic specifications, objects have abstract representations as tuples of functions. Essentially, objects are represented by their *observable behavior*. Here, observable behavior means operations that permit one object to be distinguished from another. Furthermore, the semantics of the FASE specification are those of a final algebra rather than an initial algebra: that is, the most abstract algebra with a given behavior.

3.5.2 Execution and Rapid Prototype Support

The ability to test specifications by executing them is a major feature of the FASE system. However, it is possible to write specifications that are not executable. FASE specifications are always executable if they lack quantifiers. Specifications with restricted quantification (e.g., over a finite set) are often executable.

The FASE environment is integrated with Franz Lisp permitting the user to write Lisp programs that exercise specifications. Additionally, FASE allows the user to move between specifications and implementations, thus permitting the user to develop a complete specification and then substitute Lisp implementations of some objects for their specifications.

To facilitate the concrete implementation of abstract specifications, FASE supplies a built-in random tester. This tester selects a collection of expressions and evaluates each one using both the implementation and the specification; discrepancies are reported to the user.

FASE permits the user to supply a a more 'human readable' grammar, in addition to the standard Lisp-style notation used in writing the specifications. This grammar is supplied in a separate file called the "signature" file. The grammar is parsed based on the Earley algorithm, and is somewhat more powerful than the mixed-fix syntax allowed by OBJ. In addition, placing the signature in a separate file from the body of the specification makes it possible to provide multiple grammars for each specification.

Finally, in [KA84] the following methodology is suggested for implementors starting with a specification:

- Interactive evaluation of expressions to gain understanding
- During development of the implmentation, use interactive evaluation of test cases to decide upon details
- Use user-defined syntax to test the implementation
- Extensive random testing using the built-in RandTestGen

The execution of specifications involves both lazy evaluation and finite functions. Lazy evaluation is necessary, since objects are considered to be *infinite* structures describing all future behavior, and using eager evaluation would result in infinite computations.

Since FASE specifications generally are composed of finite functions (i.e., take on a default value at all but a finite number of points) objects may be represented by sequences of (argument, value) pairs plus a default value. This greatly reduces the amount of work that must be done during execution, and prevents the deterioration of performance that generally results when an object is modified several times.

3.5.3 Abstraction Mechanisms

Because of the underlying final algebra semantics, FASE objects are described by indicating the behavior of distinguishing set operations. In initial algebra systems such as OBJ, operations are described by giving a method for calculating their values. This can bias implementations based upon such specifications, indicating that the specification is not completely abstract. FASE specifications do not have this problem, since only distinguishing behavior is described.

3.5.4 Forms of Logic Supported

Specifications may be written using first-order logic, including quantifiers. Higher-order logic is not supported. There is no direct provision for temporal logic; however, the appropriate axioms could probably be added without difficulty.

3.5.5 Verification and Theorem Proving Supported

Some work has been done to allow FASE to work with a proof tree editing system called TED. Theorems may be stated using the syntax specified in the data type's signature, and TED manages proof sequences by parsing the theorems and then sending them to the user's choice of theorem-prover and remembering the result of the proof.

3.5.6 Specification Checking—Completeness, Consistency, and Soundness

One advantage of final algebra specifications is that it is easy to determine when an object has been completely defined. This is not true for all specification systems: for example, it is very difficult to determine when an OBJ object is complete.

3.5.7 Examples

There are two major styles to FASE specifications. Both involve the *Distinguishing Set.* A FASE specification of a SET done in an (initial) algebraic style is shown in Figure 3.5.7. In this specification, the operators `empty`, `add`, `rem` are defined in terms of the distinguishing set operator `isin`. The operator `max` is defined using quantification: for integers (`n:Int`), n is in S (`isin(n,S)`) and for all integer m (`AA m:Int`) if m is in S than n is greater than or equal to m.

However, there is a second style of specification which makes the final algebra nature of the specification explicit. Such a specification is shown in Figure 3.5.7. In this specification, the operations `empty`, `add`, `rem` are all defined in terms of their future behavior with respect to the distinguishing set operation `isin`. For example, the specification for `empty` states that when `isin(n)` is applied to `empty` the observable result is always false. The specification for adding a number m to the set S states that the observable behavior for `isin(n, S)` is true if either n is in S or n was the integer just added (n = m).

```
SetofInt

empty :  -> SetofInt
add : Int SetofInt -> SetofInt
rem : Int SetofInt -> SetofInt
isin : Int SetofInt -> Bool
max : SetofInt -> Int

DISTINGUISHING SET isin ;

isin(n, empty) => false ;
isin(n, add(m,S)) => n=m | isin(n,S) ;
isin(n, rem(m,S)) => ~n=m & isin(n,S) ;
max(S) =>
     (n:Int)
       (isin(n,S) &
          ((AA m:Int)(isin(m,S)->(n>m | n=m)))
```

Figure 1: Algebraic style FASE specification for SET

```
SetofInt

empty :  -> SetofInt
add : Int SetofInt -> SetofInt
rem : Int SetofInt -> SetofInt
isin : Int SetofInt -> Bool
max : SetofInt -> Int

DISTINGUISHING SET isin ;

empty => [ <n> |-> false ] ;
add(m,S) => [ <n> |-> n=m | isin(n,S) ] ;
rem(m,S) => [ <n> |-> ~ n=m & isin(n,S) ] ;
max(S) =>
     (n:Int)
       (isin(n,S) &
          ((AA m:Int)(isin(m,S)->(n>m | n=m)))
```

Figure 2: Final Algebra specification of SET

3.5.8 Critical Remarks

As described in [KA84], one weak point of the FASE system is the inability to execute some natural specifications that include quantifiers. It is possible to rewrite these specifications so that they will become executable, but at the expense of supplying either a less abstract definition or by substituting less natural definitions.

Errors are handled in a very simple fashion; no provision is made for expressing error recovery or error messages.

It is possible to write ambiguous grammars for the user-defined signature files; this is not true for OBJ. However, the user-defined syntax permitted by the Earley algorithm is more general than that of OBJ.

3.6 HOL

3.6.1 Overview

HOL is a general theorem proving system developed at the University of Cambridge [Gor87], [CGM87] that is based on Church's higher-order logic. Church developed higher-order logic as a foundation for mathematics, but it is a promising language for describing computational systems of all kinds. HOL can be used for proving properties of anything that can be expressed in higher–order logic including hardware and software [Joy88b], [Joy88a]. This section provides a brief introduction to higher–order logic as well as its implementation in HOL.

3.6.2 The HOL Verification System

Higher-order logic is a mathematical theory; HOL is a computer program that uses higher-order logic to verify hardware and software. HOL grew out of Robin Milner's LCF theorem prover and is written in the computer language ML.

HOL has several subsystems that contribute to its use as a verification environment:

a. Several theories, including booleans, individuals, numbers, products, sums, lists, and trees. These contain the five axioms that form the basis of higher-order logic as well as a large number of theorems that follow from them.

b. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

c. Canned methods of proceeding in a goal directed fashion called *tactics*. Tactics are functions that embody knowledge about commonly used proof techniques. These tactics can be applied to goals to produce simpler goals, a number of subgoals that prove the original goal and so on. Tactics can never be used to build an incorrect proof since each of them must include a justification for the tactic in the form of an already proven HOL theorem.

   Examples of tactics include REWRITE_TAC that rewrites a goal according to some previously proven theorem or definition, GEN_TAC that removes unneeded "forall" clauses from the front of terms, and EQ_TAC which says that to show two things are equivalent, we should show that they imply each other.

d. A proof management system that keeps track of the state of a proof and manages goals and subgoals.

e. A metalanguage for programming the verification system. The metalanguage for HOL is ML, the language in which HOL is written. ML is a type polymorphic, lambda calculus–based functional language. ML is a powerful programming language in its own right and has been described in [GMW79]. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be built, and results of proofs can be made into new theories for later use. The metalanguage makes the verification system extremely flexible.

Terms There are four kinds of terms in HOL:

1. Variables
2. Constants

Operator	*Application*	*Meaning*
=	t1 = t2	t1 equals t2
,	t1,t2	the pair t1 and t2
/\	t1 /\ t2	t1 and t2
\/	t1 \/ t2	t1 or t2
==>	t1 ==> t2	t1 implies t2
<=>	t1 <=> t2	t1 if and only if t2

Table 1: HOL Infix Operators

Binder	*Application*	*Meaning*
!	!x.t	for all x, t
?	?x.t	there exists an x such that t
@	@x.t	choose an x such that t

Table 2: HOL Binders

3. Function applications
4. Abstractions

Variables and constants are denoted by any sequence of letters, digits, underlines and primes starting with a letter. Constants are distinguished in the logic and any identifier that is not a distinguished constant is taken as a variable.

Function application is denoted by juxtaposition. Thus a term of the form "t1 t2" is an application of the operator t1 to the operand t2. Its value is the result of applying t1 to t2.

An abstraction denotes a function and takes the form "\x.t". [1] An abstraction "\ x.t" has two parts: the bound variable x and the body of the abstraction t. It represents a function, f, such that "f(x) = t". For example, "\ y.2*y" denotes a function which doubles its argument.

Constants can belong to two special syntactic classes. Some constants are declared to be infix. Infix operators take two arguments and are written "rand1 op rand2" instead of "op rand1 rand2". Table 1 shows several of HOL's built-in infix operators.

Another special class that constants can belong to is the class of binders. A familiar example of a binder is ∀, written in HOL as !. If c is a binder, then the term "c x.t" (where x is a variable) is written instead of "c(\x.t)". Table 2 shows several of HOL's built-in binders.

Types HOL is strongly typed to avoid Russell's paradox. [2] Every term in HOL is typed according to the following recursive rules:

- Each constant and variable has a fixed type. letter.
- If x has type α and t has type β, the the abstraction \x.t has the type $(\alpha \rightarrow \beta)$.
- If t has the type $(\alpha \rightarrow \beta)$ and u has the type α, the the application t u has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters and digits. Thus

[1] The ASCII symbol \ is used in place of the greek letter λ which is customarily used to represent an abstraction.

[2] Russell's paradox is the case where in a high-order logic, one defines a predicate that leads to a contradiction. Specifically, suppose that we define P as P(x) = x(x) where denotes negation. P is true when its argument applied to itself is false. Applying P to itself leads to a contradiction since P(P) = P(P). This is prevented by typing since in a typed system the type of P would not allow it to be applied to itself.

Operator	*Arity*	*Meaning*
`bool`	0	booleans
`ind`	0	individuals
`num`	0	natural numbers
`(*)list`	1	lists of type *
`(*,**)prod`	2	products of * and **
`(*,**)sum`	2	coproducts of * and **
`(*,**)fun`	2	functions from * to **

Table 3: HOL Type Operators.

`*`, `***`, and `*ab2` are all valid type variables.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1, \ldots, \sigma_n$ are types and op is a type operator of arity n, the $(\sigma_1, \ldots, \sigma_n)$op is a type. [3] A type operator of arity 0 is a type constant.

HOL has several built-in type operators which are listed in Table 3. The type operators `bool`, `ind`, and `fun` are primitive. HOL has a special syntax that allows `(*,**)prod` to be written as `(* # **)`, `(*,**)sum` to be written as `(* + **)`, and `(*,**)fun` to be written as `(* -> **)`.

The Choice Operator The axiomatization of HOL uses Hilbert's choice operator `@`. This is a binder with type given by:

```
@ : (*->bool)->*
```

The idea is that if `f:ty->bool` then `"@(f)"` denotes some value `v` such that `"f(v)"` is true.

For example, `"@x:num.x*x=25"` denotes `5` (but not `-5` as the type `num` only contains the non-negative integers). An interesting side-effect of the choice operator is that all types must be non-empty since the term `"@x:t.T"` is a non-determinant member of the type `t`; this member must exist.

Sequents, Theorems, and Inference Rules The HOL system supports proof by natural deduction. Assertions in HOL are not just boolean formulae asserting some independent truth, but are of the form `(A,C)` where `A` is a set of assumptions and `C` is the conclusion. The assertion states that if all the formulae in set `A` are true then so is the formula in `C`. This forms a sort of sequent calculus and is very similar to the way proofs are carried out by humans and thus represents a natural proof environment than other systems. The form `(A,C)` is called a *sequent.*

A *theorem* is a sequent that has a *proof.* This means that the truth of the sequent has been established through *rules of inference* from other theorems. There are certain distinguished theorems called *axioms* that are taken as true without proof. Axioms are necessary as a starting point. Naturally, any good proof system will have as few axioms as possible.

A *proof* consists of a series of steps that show that the sequent to be proven can be derived from already proven theorems using the rules of inference. Most proofs in HOL take place in a *goal-directed* manner, meaning that the sequent to be proven is written down and then treated as a goal which is broken down into subgoals until the subgoals are reduced to a point that they are trivial to prove (i.e. they are already theorems themselves).

[3] Note that type operators are postfix while normal function application is prefix or infix.

There are special kinds of axioms called *definitions*. A definition is an axiom of the form c=t where c is a constant and t is a term without free variables. A constant c is said to be defined in a theory if there is only one axiom in the theory containing c and that axiom is a definition. A theory in which all the axioms are definitions is said to be definitional. This is important because definitional theories have the property that they cannot introduce any *new* inconsistencies to the system. This property is known as *conservative extension*.

All axioms, definitions, and theorems are stored relative to a *theory*. A theory is a set of type operators, constants, axioms and parent theories. Parent theories provide for a hierarchy in which the contents of an ancestor theory are available in the child theory. Higher-order logic is extended by defining new theories. To use a theory, one declares it a parent of the theory currently being drafted and then all of the components of the parent are available for use in the new theory.

Goal Directed Proofs in HOL The approach to proving theorems that is used in HOL is due to Robin Milner. He originally developed the approach for a proof system called LCF. LCF was designed for reasoning about recursively defined functions. The HOL system is a direct outgrowth of LCF.

A goal is a sequent, that is, it has a list of assumptions and a conclusion. In general, a goal that is to be turned into a theorem will not have any assumptions, and so the list of assumptions will be empty. After the goal has been proven, the sequent will have a proof, if it is indeed true, and becomes a theorem (recall that a theorem is a sequent with a proof).

In HOL, a proof is a function that turns a list of theorems into a theorem. We can obtain proofs by applying *tactics*. A tactic is a function that takes a goal as its parameter and produces a list of subgoals and a proof. When the subgoals are proven, they will each have associated theorems. Applying the proof to the list of theorems proving the subgoals produces a theorem for the original goal. For example, suppose that T is a tactic and g is a goal. Evaluating T g results in a list of subgoals and a proof, [g_1 ; g_2 ;...;g_n] and p. Eventually, after proving each of the subgoals (using tactics) we will have a list of theorems, [t_1 ; t_2 ; ... ;t_n]. If T is a valid tactic, applying p to that list of theorems achieves g and the proof is complete. Tactics will be discussed in detail in a later section.

3.6.3 Axiomatic Basis for HOL

This section discusses the mathematical foundations of HOL. Moreover, it discusses the way this foundation is implemented in HOL. Higher-order logic is based on Church's typed-λ-calculus. There are many formulation of this logic. The one used in HOL is very similar to the one discussed in [And86].

All of the data objects in HOL are classified according to their membership in sets. For example, 3 is of the type num because it belongs to the set containing all of the natural numbers. Types in higher-order logic are expressions that denote sets. All types in higher-order logic must denote non-empty sets. This is enforced by the new_type function that requires, in addition to the representation for the new type, a theorem proving that it is non-empty.

At the heart of HOL are five axioms. Axioms can neither be proven or disproven. They represent what we believe to be true about the world. A complete discussion of the axioms on which HOL is based is beyond the scope of this report, but a brief discussion should shed some light on what they mean. Here are the five axioms:

1. The boolean cases axiom

```
BOOL_CASES_AX    |- !t:bool. (t=T) \/ (t=F)
```

says that every boolean object is either true of false.

2. The implication antisymmetry axiom

```
IMP_ANTISYM_AX   |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)
```

states that if t1 implies t2 and t2 implies t1, then t1 and t2 are equal. In some sense, this axiom relates implication and equality.

3. The η–axiom

```
ETA_AX           |- !t:*->**. (\x. t x) = t
```

states, in a round–about way, that two functions are equal if they give the same results when applied to the same arguments. This property is called *extensionality*. For example, the η–axiom states that $\lambda\, x$. sin x and sin are the same function.

4. The select axiom

```
SELECT_AX        |- !P:*->bool.!x. P x ==> P(@ P)
```

states that the choice operator, when used with a predicate expressing a particular property returns an element that satisfies that predicate. This really just defines the meaning of the choice operator, as we have discussed it, in a formal manner.

5. The infinity axiom

```
INFINITY_AX      |- ?f:ind->ind. ONE_ONE f /\ ~(ONTO f)
```

states that there is a set, called :ind that has an infinite number of members.

From these five axioms, all of the thousands of theorems that make up HOL can be derived. In addition to the derived theorems, we can also make use of definitions. Recall from our brief discussion of definitions and the principal of conservative extension that definitions, while technically axioms, cannot undermine the soundness of a logical system. Thus, we can define a number of familiar logical objects using the primitive constants =, ==> and @. Some of these include:

```
T_DEF        |- T = ((\x:*.x) = (\x:*.x))

FORALL_DEF   |- ! = \P:*->bool. P=(\x.T)     (binder)

EXISTS_DEF   |- ? = \P:*->bool. P(@ P)    (binder)

AND_DEF      |- /\ = \t1 t2.!t. (t1==>t2==>t)==>t    (infix)

OR_DEF       |- \/ = \t1 t2.!t. (t1==>t)==>(t2==>t)==>t    (infix)

IFF_DEF      |- <=> = \t1 t2. (t1==>t2) /\ (t2==>t1)    (infix)

F_DEF        |- F = !t.t

NOT_DEF      |- ~ = \t. t ==> F
```

Axioms and theorems represent the data in a verification system such as HOL. In order to derive theorems from the axioms given above, we need rules of inference that say what kinds of derivations are legal. The rules of inference are a very important part of any system of logic. If derivation rules could be added to the system without regard to some formal basis, HOL would be unreliable and unsound. HOL has a set of eight primitive inference rules from which all of the other rules in the system must be derived. An inference rule has the general form:

```
A1 |- t1          A2 |- t2     ...     An |- tn
-----------------------------------------------    (conditions)
                  A |- t
```

Such a rule asserts that if the premises are proved and the conditions hold, then the conclusion may be deduced. Here are the eight inference rules:

1. ASSUME – Given a term "t", conclude t ⊢ t.
2. REFL – Given a term "t", conclude ⊢ t = t.
3. SUBST – Perform substitution for variables in a theorem
4. BETA_CONV – Perform β–reduction on a term. β–redcution substitutes a function argument into the function body: (λ x.t[x]) u = t[u].
5. ABS – Introduce abstractions into theorems.
6. INST_TYPE – Instantiate type variables.
7. DISCH – Discharge an assumption, that is, an assumption on the assumption list is moved to the conclusion as the antecedent in an implication.
8. MP – Perform modes ponens. Given ⊢ t1 and ⊢ t1 ==> t2 deduce the theorem ⊢ t2.

Inference rules in HOL are ML functions that return a theorem object. HOL's strong typing provides security in that the user cannot create arbitrary theorem objects without using a rule of inference. New rules of inference are created from the primitive inference rules through function composition.

All proofs in HOL can eventually be reduced to proofs using the primitive inference rules. This is a very powerful concept because it means that one need only look at the ML code implementing the eight primitive inference rules to trust the result from the system. In most theorem provers, there are literally thousands of lines of code that must be trusted in order to trust the result from the system. The code implementing the eight inference rules can be read and understood in few hours; as a result, it is fairly easy to convince oneself of their correctness.

Tactics and Conversions There are two ways of going about a proof. One way is to start with a set of axioms and inference rules and work forward until the desired theorem is derived. For example, given the theorems:

```
LESS_SUC = |- ! x y . x < y ==> x < (SUC y)

FIVE_LESS_SIX = |- 5 < 6
```

and the modus ponens inference rule

```
th1 |- th2    th1
-------------------
        th2
```

we can derive ⊢5 < SUC 6 using the HOL expression

```
MP LESS_SUC FIVE_LESS_&SIX
```

The second way to prove a theorem is to start with the theorem that you want to prove as a goal and then to reduce it to subgoals until the subgoals are already proven theorems or axioms. Of course, each of the reduction steps must be justified in some way. This is the way most proofs are done in HOL and we will see many examples of them.

As stated earlier, a tactic is an ML function that is applied to a goal to reduce it to subgoals. For example, a tactic called MATCH_MP_TAC could be used to prove the goal from the previous example. MATCH_MP_TAC says that a theorem of the form x ==> y can be used to reduce a goal of y to a subgoal of x. The goal in the previous example was

```
5 < SUC 6
```

Using MATCH_MP_TAC and LESS_SUC, we get a new subgoal of

```
5 < 6
```

which of course, is trivially true by the theorem FIVE_LESS_SIX.

Tactics can be described using the following notation:

```
          <goal>
========================        <tactic>
<goal> <goal> ... <goal>
```

For example, CONJ_TAC is described by

```
t1 /\ t2
=========        CONJ_TAC
t1    t2
```

Thus CONJ_TAC reduces a goal of the form (asl,"t1 / t2") to two subgoals (asl,"t1") and (asl,"t2"). The fact that the assumptions of the top-level goal are propagated unchanged to the two subgoals is indicated by the lack of mention of assumptions in the notation.

Another example is INDUCT_TAC, the tactic for doing mathematical induction on the natural numbers:

```
        !m.t[m]
======================        INDUCT_TAC
t[0]    t[m] t[SUC m]
```

INDUCT_TAC reduces a goal of the form (asl,"!m.t[m]") to a basis subgoal (asl,"t[0]") and an induction step subgoal (t[m].asl,"t[SUC m]"). Note that the induction hypothesis, "t[m]" has been appended to the list of assumptions.

Tactics provide a way of doing proofs in HOL that closely mimics the way one does proofs in mathematics. For example, it is quite common in a proof of equality, "x = y", to prove two subcases, namely "x ==> y" and "y ==> x". HOL's EQ_TAC does precisely this. The following is a brief description of some of the most commonly used tactics:

1. GEN_TAC is used to reduce goals of the form "! x . t[x]" to a goal of the form "t[x]". In an informal proof, one would commonly drop a universal quantification whose scope is the entire term since it is understood that free variables are taken to be universally quantified.
2. ASM_CASES_TAC is used to consider alternative cases. In an informal proof of a goal such as "if x then y else z" one would show that if "x" is true then "y" is true and if "x" is false then "z" is true. "ASM_CASES_TAC x" reduces a goal to two subgoals, one with "x" added to the assumption list and one with " x" added to the assumption list. When both goals are proven, the original goal is considered proven.
3. EQ_TAC, as described above, is used to reduce a goal involving equality to two subgoals, one with the implication from right to left and the other with implication from left to right.

4. REWRITE_TAC is used to rewrite a goal using known theorems. It also performs simple boolean simplification. In an informal proof one commonly says "...and now, because x = y..." meaning that the goal is simplified using the theorem ⊢x = y by algebraic manipulation. tt REWRITE_TAC takes a list of theorems and rewrites the goal with them to produce a subgoal.
5. STRIP_TAC is used with goals containing implications. If the goal has the form x ==> y we can assume x is true since if it is not, the goal is trivially true by the definition of implication. STRIP_TAC removes the antecedent from the implication and adds it to the assumption list. The consequent is the new goal.
6. EXISTS_TAC is used to pick a value for an existentially quantified variable. This is a reasonable thing to do since an existentially quantified theory merely states the existence of a single value which makes it true. Certainly if we can pick such a value for our goal, then the goal is true. EXISTS_TAC "x" reduces a goal of the form "? y.t[y]" to a goal of the form "t[y/x]".

A tactic is an ML functions that maps an argument of type goal (term list # term) to a pair consisting of a list of subgoals and a validation. A validation is a function that takes a goal list and produces a theorem. If the theorem corresponds to the original goal, then the tactic has succeeded in proving the goal.

Because the result of the validation is of type thm, it must use ML functions that return that type. We have already seen functions that return objects of type thm, they were inference rules. In some sense, tactics let the user do a goal directed proof, all the while building a large forward proof as the validation. Goal directed proof in HOL is just a way of getting the system to keep track of the details of the forward proof for you.

Because all steps taken by tactics must eventually be validated by an inference rule, tactics can never prove a false statement. Tactics can, however, look like they are working and not really be valid. They won't produce a "wrong" proof, but they will waste your time. You should be careful to ensure that tactics you write are valid.

Tacticals A tactical is an ML function that returns a tactic (or tactics) as result. Tacticals are used to combine existing tactics into new tactics. They are used extensively in HOL proofs. The most commonly used tactics are ORELSE, THEN, THENL and REPEAT which are described in the following paragraphs.

The tactical THEN corresponds to sequencing in programming flow control. The specification of THEN is

```
THEN : tactic -> tactic -> tactic
```

If T1 and T2 are tactics then T1 THEN T2 is a tactic which first applies T1 and then applies T2 to *all* the subgoals produced by T1.

The tactical ORELSE corresponds to alternation in programming flow control. The specification of ORELSE is

```
ORELSE : tactic -> tactic -> tactic
```

If T1 and T2 are tactics ORELSE T1 ORELSE T2 is a tactic which first tries T1 and then if T1 fails tries T2.

The tactical THENL corresponds to roughly to parallel execution in programming flow control. The specification of THENL is

```
THENL : tactic -> tactic list -> tactic
```

If T is a tactic which produces n subgoals and T1,...,Tn are tactics then the tactic T THENL

[T1;...;Tn] first applies T and then applies Ti to the ith subgoal produced by T. THENL is used when one wants to do different things to different subgoals.

The tactical REPEAT corresponds to looping in programming flow control. The specification of REPEAT is

```
    REPEAT : tactic -> tactic
```

If T is a tactic then REPEAT T is a tactic that repeatedly applies T until it fails.

As a simple example, the following is a tactic built using tacticals and some of the tactics that we saw in the last section:

```
(REPEAT GEN_TAC) THEN EQ_TAC THEN (EXISTS_TAC "x" ORELSE STRIP_TAC)
```

This tactic applies GEN_TAC repeatedly to the goal until all of the universal quantifiers have been stripped, breaks the equality into two subgoals using implication and then either uses EXISTS_TAC to pick a value for an existentially quantified variable of uses STRIP_TAC to strip the antecedent from the goal.

Conversions Conversions are a very important part of the HOL system. References [Pau83] and [Pau87] give a more detailed introduction to conversion.

Conversions are functions that map terms into theorems. Conversions play an important role in rewriting terms, manipulating goals and writing decision procedures. As an introduction, let's look at one of the most basic conversions REWRITE_CONV. REWRITE_CONV takes a single argument that is an equality theorem and returns a conversion for that specific theorem. Using this conversion produces a theorem tailored to the particular term given to it. For example, suppose we create a conversion called less_conv using REWRITE_CONV and LESS_THM.

```
LESS_THM = |- !(m:num) (n:num). m < (SUC n) = (m = n) \/ m < n

let less_conv = REWRITE_CONV LESS_THM;;
```

We can use this conversion to produce instantiations of LESS_THM.

```
less_conv "m < (SUC n)";;
|- m < (SUC n) = (m = n) \/ m < n

less_conv "(5+6) < (SUC (5+6))";;
|- (5 + 6) < (SUC(5 + 6)) = (5 + 6 = 5 + 6) \/ (5 + 6) < (5 + 6)

less_conv "n < m";;
evaluation failed     term_match
```

Note that, if possible, the conversion produces a theorem that is a *specialization* of the theorem given to REWRITE_CONV such that its left–hand side matches the given term. The last example shows that the conversion fails if no match is possible.

Conversionals Conversionals are to conversions as tacticals are to tactics. They are operators for putting conversions together. THENC is the sequencing operator for conversions. The expression (c1 THENC c2) t uses c1 to produce a theorem $\vdash$ t = t1 and then c2 to produce a theorem $\vdash$ t1 = t2. The overall effect is a theorem $\vdash$ t = t2 because of transitivity. The conversion fails if either of c1 or c2 fails.

ORELSEC is the alternation operator for conversions. The expression (c1 ORELSEC c2) t uses c1 to produce a theorem $\vdash$ t = t1. If that fails, then it uses c2 to produce a theorem $\vdash$ t = t2. The conversion fails if both of its arguments fail.

REPEATC repeatedly applies a conversion to a term until it fails. Here is the implementation of REPEATC

```
letrec REPEATC conv t =
    ((conv THENC (REPEATC conv)) ORELSEC ALL_CONV) t;;
```

REPEATC can be implemented as a recursive ML function that uses the conversionals THENC and ORELSEC. ALL_CONV is the identity conversion. It alway succeeds.

Converting Subexpressions None of the conversions that we have seen so far work on the subexpressions of terms. For example,

```
less_conv "(1 < (SUC n)) /\ (n < (SUC p))";;

evaluation failed     term_match
```

The conversion is applied to the top level, that is, to the conjunction, and not finding a match, fails. We need to be able to apply conversions to the subterms as well.

The function DEPTH_CONV applies a conversion to all the subterms in an expression depth first. Note that it does not retraverse a term, so the result may not be in the simplest form. Here is an example of its application to a simple expression:

```
    (DEPTH_CONV less_conv) "(1 < (SUC n)) /\ (n < (SUC p))";;
    |- 1 < (SUC n) /\ n < (SUC p) = ((1 = n) \/ 1 < n) /\
                                    ((n = p) \/ n < p)
```

The function TOP_DEPTH_CONV applies a conversion to all the subterms in an expression in a top down manner. The function retraverses the result until no further conversion can be applied. The final result of using TOP_DEPTH_CONV is always in its simplest form, but takes longer than using DEPTH_CONV.

CONV_TAC As we have seen, conversion produces a theorem from a term. That doesn't do us much good in a goal directed proof. HOL provides a tactic called CONV_TAC for using the results of a conversion in a goal directed proof. It takes a single argument, the conversion that is to be used, and produces a tactic.

```
CONV_TAC : (conv -> tactic)
```

In essence, CONV_TAC applies a conversion to the goal and then returns the right-hand side of the theorem returned by the conversion as the new subgoal. If the conversion fails, then the goal is unchanged.

Defining Conversions In order to understand how conversions work, the following example showing the definition and use of a simple conversion is presented. Suppose that we wish to prove the following goal:

```
set_goal([], "! n m . (n <= (SUC m)) = ((n <= m) \/ (n = (SUC m)))");;
```

After rewriting with theorems about the meaning of <= and what it means to be less than the successor of something, we are left with the following goal:

```
"!(n:num) (m:num).
  ((n = m) \/ n < m) \/ (n = SUC m) =
     (n < m \/ (n = m)) \/ (n = SUC m)"
```

Now, of course, this is trivially true, but we still have to do some rather unpleasant specialization of the disjunctive symmetry theorem to finish the proof. We'd like to be able to have a tactic that can show terms like this are true.

The reason that we recognize a goal such as the one is true is because we know that disjunction is associative and commutative. We can reparenthesize and reorder the terms at will. We want to write a conversion that reassociates the terms and then order them in some consistent manner. This is called "term normalization."

In our conversion, we will use the following theorems about disjunction:

```
DISJ_ASSOC = |- !a b c. a \/ b \/ c = (a \/ b) \/ c

DISJ_SYM = |- !(t1:bool) (t2:bool). t1 \/ t2 = t2 \/ t1

DISJ3_SYM = |- !a b c. (a \/ b) \/ c = (a \/ c) \/ b
```

HOL has a built-in ML function << that defines an arbitrarily, but unique total order on terms.

```
"a:bool" << "4";;
false : bool

"4" << "a:bool";;
true : bool
```

We want to define DISJ_SYM_CONV such that

```
DISJ_SYM_CONV "a \/ b" --> |- a \/ b  = b \/ a        if b << a

DISJ_SYM_CONV "(a \/ b) \/ c"  -->
                          |- (a \/ b) \/ c = (a \/ c) \/ b  if c << b
```

Here is the ML code for DISJ_SYM_CONV

```
let DISJ_SYM_CONV t = (
    let (t1,t2) = dest_disj t in
    if (not (is_disj t1) & (t2 << t1)) then
        (SPECL [t1;t2] DISJ_SYM)
    else
        let (t3,t4) = dest_disj t1 in
        if (t2 << t4) then
          (SPECL [t3;t4;t2] DISJ3_SYM)
        else fail
    ) ? failwith 'DISJ_SYM_CONV';;
```

Of course, this simple conversion only works if the goal has three or fewer subterms.

We can write a general version using DISJ_SYM_CONV that works even when the goal has more than three subterms as follows:

```
let DISJ_NORMALIZE_CONV =
  TOP_DEPTH_CONV (REWRITE_CONV DISJ_ASSOC)
  THENC TOP_DEPTH_CONV DISJ_SYM_CONV;;
```

The normalization conversion reassociates the term and then uses TOP_DEPTH_CONV to completely order the term, no matter what its size.

Now we can write the tactic. We want to normalise an expression and then prove it using REFL_TAC if possible. Note that we don't want to just fail if the two sides aren't equal, we still want the expression normalized. Also note that REFL_TAC won't work if there are universally quantified variables. The tactic should also be written such that GEN_TAC is used *only* if the goal can be solved with REFL_TAC. Here is the ML code for such a tactic.

```
let DISJ_NORMALIZE_TAC =
CONV_TAC DISJ_NORMALIZE_CONV
THEN (((REPEAT GEN_TAC) THEN REFL_TAC) ORELSE ALL_TAC);;
```

This tactic is very general. Not only does it solve goals where the disjunctive terms are equal, but it normalizes unequal terms, so that we can determine where the inequality lies.

3.6.4 Critical Remarks

This section relates our personal impressions of HOL after having used it extensively. There are several points that should be made before a detailed discussion of the strengths and weaknesses of HOL are discussed.

HOL is still a research system, not a commercial product. In addition, HOL is young in relation to many other theorem provers such as Boyer-Moore and EHDM. These points have several implications:

1. HOL is not polished. As an example, error messages, while much improved over earlier versions, are terse and sometimes not helpful. As another example, the help system is incomplete and somewhat *ad hoc*.
2. HOL is not finished. An example is the theory of numbers. While there is a large collection of theorems about numbers, they are *ad hoc*. A more careful analysis of what theorems are important might include other theorems about numbers in the base system. Another example is the set of conversions for dealing with universal and existential quantifiers. The set is by no means exhaustive and all of the conversion are not found in the same file.

Overall, HOL has great potential, but there are many deficiencies. The next sections will discuss the strengths and weaknesses on the basis of the following criteria:

1. The overall usability of the theorem prover.
2. HOL's formal foundations.
3. The soundness of the theorem prover and its trustworthiness.
4. The expressibility of HOL's specification language for describing the structure and behavior of hardware designs.
5. The suitability of HOL's specification language for expressing generic designs.
6. The suitability of HOL's specification language for describing systems composed of hardware and software.
7. The suitability of HOL's specification language for describing concurrent and distributed systems.
8. Examples for which HOL is particularly well-suited and those it does not handle well.
9. HOL's performance on large examples.
10. The extensibility of the system, including defining new types and decision procedures.
11. The proof power of the HOL environment. Consideration will be given to how automatic proofs are, the capability of the system to carry out proofs by induction, the capabilities for reusing proofs, and the ability of the user to interact with the prover when it fails to discover a proof.
12. The "software engineering" features of the environment, e.g., its capability to reason about changes to designs, its support for modularization and refinements, its capabilities for integrating design and proof.

Ease of Use This section discusses the overall usability of the HOL system. Ease of use hinges on several factors, among them the user interface, the style of proof, the proof management system, and theory and theorem management.

HOL's user interface is somewhat Spartan. As delivered, it is nothing more than a *read-eval-print loop*. Work should be done to supply a user interface that takes advantage of the features available on modern workstations such as windows, menus, expanded character sets, mouse-driven editing, etc.

The style of proof in HOL is natural and flexible. Proofs can take place in a forward, backward, or mixed mode. Tactics provide a natural way of going about goal-directed proofs. Using tactics, proofs can be done in a manner that mimics the way a human proves theorems. Indeed, one of the best ways to structure a proof in HOL is to do a hand-proof first and use the informal proof as a guide in developing the formal proof.

HOL contains a simple proof management system based on a goal stack. More extensive proof management systems exist independent of HOL that provide the capability of editing the proof tree at arbitrary points and give the user the flexibility of pursuing subgoals in a less structured manner. An example is the Treemacs proof editor from the University of Illinois [Ham88].

HOL provides a simple system for managing theories in the form of a library package based on the UNIX file system. HOL stores theories in directories of files and can be loaded from within HOL using the `load_library` command.

Another problem that arises when using HOL is that the user frequently knows which theorem should be used next, but does not know the name of the theorem. This necessitates searching through pages of listings giving the names of theorems, looking for the particular theorem needed for the job at hand. This works when the number of theorems is small, but becomes unwieldy for large collections. It is not inconceivable that a good theory of numbers would contain thousands of theorems. This problem needs to be solved, but it is not clear what the best solution is. It is possible that a database management system using pattern matching might be step in the right direction.

Perhaps one of the most frustrating areas impinging on ease of use is the quality of documentation about the system. In this area, HOL gets particularly poor marks. Users are left to discover very important points about the theorem prover by chance, or from informal contact with other users. This should change in the future since Mike Gordon has a contract to produce good documentation for the system. This documentation is to be available early in 1990.

Formal Foundations HOL's formal foundations are very strong. HOL is based on Church's typed λ-calculus (also known as higher-order logic). Typed λ-calculus was developed as a formal basis for mathematics and has received considerable attention independent of its use in theorems provers. Different logics could, of course, and are used in theorem provers. There is no way to argue that higher-order logic is the "best" logic, but one can conclude that higher-order logic imposes no fundamental limitations on HOL. Another point in its favor is that the theoretical foundation of HOL is familiar to people trained in classical logic. Most of the logic is already known to the user and the user is not forced to learn a new logic before beginning.

Higher-order logic seems to be necessary for building a trustworthy theorem prover. See the section that follows on trustworthiness for more information. In addition, higher-order logic allows generic specifications to be easily defined.

Soundness and Trustworthiness Simply put, to say that a theorem prover is sound is to say that one cannot conclude that false is true within the system. To say that a theorem prover is trustworthy is to say that it faithfully implements the logic upon which it is based.

There is little doubt that the logic upon which HOL is based is sound. One cannot reach this conclusion by proof since it is a hypothesis that can never be proven, only disproven. The more important question is whether or not HOL faithfully implements higher-order logic.

HOL is based on *5* primitive axioms. There are three ways to create a new theorem in HOL:

1. One can declare it an axiom.
2. One can make a definition.
3. One can create a theorem from existing axioms definitions, and theorems using inference rules.

While HOL allows one to declare arbitrary axioms, this practice is discouraged unlike other systems such as EHDM where this is the standard means of introducing new bases for proof. Axioms are distinguished in HOL so that they can be identified in theories, providing a measure of safety.

The principle of conservative extension assures us that as long as definitional axioms are added to the system then the system will remain sound. HOL encourages the use of definitions. Theories that are free of axioms (that is, contain only definitions and theorems) are called definitional and are sound .

An inference rule is an ML function that returns an object of type theorem. HOL contains *8* primitive inference rule. Every other inference rule is based on some functional composition of the *8* primitive inference rules.

The combination of definitional theories and a small number of axioms and inference rules make HOL extremely trustworthy. In order to be assured that the a result of *true* is correct, one need only be convinced of the correct implementation of *5* inference rules and the *8* primitive axioms. Incorrect code elsewhere might lead the theorem prover to fail to return an answer, but will not lead to an incorrect answer.

Higher-order logic seems to be essential to building a trustworthy theorems prover. Without higher-order logic, it is impossible to declare recursive axioms in a definitional manner. We have seen that the ability to use definitions rather than declaring arbitrary axioms is essential to maintaining the soundness of the theorems prover.

Another facet of trustworthiness where higher-order logic is important is induction. Without higher-order logic, the induction schema cannot be expressed in the logic and thus the induction cannot be carried out mechanically in the system but is left to for the user. Various first-order system have attempted to provide induction without higher-order logic, but none have been successful.

One deficiency of HOL in the area of trustworthiness is HOL management of theories. HOL's theories are stored in text files (which contain LISP expressions) and thus can be edited without invalidating them. In addition, when a theory is changed in the system, its descendants are not invalidated as they should be. While attempts to add this capability would probably be futile due to the insecurities of the operating system and other issues, this area should not be ignored in a system to be used for commercial proofs.

HOL currently does a good job of protecting the user from accidentally cheating and declaring something proved which is not. It does far less to guard against deliberate tampering. Before HOL is used in a commercial endeavor, one would probably also want more assurance that the results from HOL are genuine. An independent proof checker that checks a transcript of the primitive inferences taken by HOL during a proof to ensure that they are all valid and no shortcuts were taken would solve this problem since the user could not fake a correct series of inference steps without actually doing the proof. Proof checkers are simpler to write than theorem provers and would provide an added measure of security.

Trustworthiness has a downside. HOL is extremely trustworthy precisely because it requires that every theorem be derived by primitive inference. This means that many of the decision procedures used in other theorem provers for Peano arithmetic, boolean algebra, etc. cannot be used in HOL as they get their speed from doing the manipulation of terms outside the object world, that is without using primitive inferences. Research should be undertaken to find decision procedures that work by primitive inference but are fast enough to be of use.

Expressiveness In the area of expressiveness, we will deal with three topics:

1. The expressibility of HOL's specification language for describing the structure and behavior of designs.
2. The suitability of HOL's specification language for expressing generic designs.
3. The suitability of HOL's specification language for describing systems composed of hardware and software.

As was mentioned in the preceding section, HOL is based on higher-order logic. This gives HOL incredible flexibility in what it can express. Higher-order logic is Turing complete, meaning that any thing that can be expressed using any other programming language, can be expressed in HOL. This is as much as can be hoped for in any language. In addition, because of its higher-order capabilities, specifications expressed in HOL are generally more concise and clear than they might be in some other notation.

Since HOL is based on higher-order logic, any specification, at any level, can be parameterized, but very little research has been done in this area. One could easily imagine, for example, a generic ALU specification and implementation that allows each of the ALU's functions to be specified and verified separately; these separate proofs and specifications could then be automatically composed with the generic ALU implementation to form a custom ALU proof. While this has not yet been done, there does not seem to be any fundamental limitation in HOL that would prevent this.

The question of HOL's suitability for specifying mixed systems can perhaps be best addressed by example. Jeff Joyce of the University of Cambridge has specified and verified a microprocessor called *Tamarak* along with a compiler for the microprocessor. This verification shows the efficacy of HOL for specifying mixed systems. On another level, since almost all modern computers are microcoded, any verification of a microprogrammed microprocessor demonstrates HOL's capabilities in this area. There have been several large microcoded examples completed at different institutions.

Performance on Different Classes of Problems Higher-order logic is extremely expressive, but that does not mean that HOL will perform equally in all problem domains. Because HOL has been used extensively to verify hardware systems, there is a large body of knowledge about using HOL in this domain and many useful techniques have been developed and widely disseminated. Researchers, both here at the University of California, Davis and elsewhere, are just beginning to use HOL to verify interesting properties for software.

Recent work at UCD has identified several limitations in the present version of HOL which make it difficult to describe the implementation of some programs. An example of a deficiency in this area is the inability of HOL to define arbitrary recursive definitions. HOL does not allow this for good reason (soundness), but of course, any real program will allow totally recursive functions. This problem is non intractable, but is simply an example of the kinds of issues that will have to be resolved before HOL is used to verify programs.

Not surprisingly, HOL seems to do particularly well in developing theories about mathematics. Set theory, group theory, etc. can be expressed and developed quite naturally in HOL. This is important since making formal proofs tractable requires that the user of the formal system have a large body of preproven theorems about mathematics readily available.

Performance on Large Examples HOL has been used by several different researchers to specify and verify several Large Scale Integration (LSI) sized examples. Most of these are hardware, as the raw HOL logic is suited to hardware descriptions. Researchers at the University of Calgary have specified and verified a small LISP-based microprocessor (the SECD microprocessor) and are in the process of verifying another (the so-called Three Instruction Machine, or TIM). Researchers at the University of Cambridge in England, where HOL was developed, have verified three or four chips that could be classified as "large," including a network control chip and a microprocessor on the same level of complexity as a PDP-8.

HOL is being used to verify the VIPER microprocessor. It appears that VIPER represents the upper end of hardware that can be verified using current techniques. There does not appear to be any fundamental limitation in HOL that would limit its usefulness to small or medium-sized problems, but new techniques for structuring proofs, new abstraction mechanisms and more automated methods of proof need to be explored.

Abstraction Mechanisms HOL is weak in the areas of structural, behavioral and temporal abstraction. For example, HOL does not contain any built-in theories about gates, devices, specific programming languages, etc., or types specific to hardware specification such as bit-vectors, words, and so forth. It is difficult to see how built-in theories of gates would be useful without having the theorem prover linked to a CAD system and supplying the user with specification and implementations for the standard libraries of gates available in the CAD system as well as decision procedures for doing proofs with them.

Another example is the lack of a formal means for dealing with temporal issues - as would be required in proofs about concurrent programs. There are techniques to be sure, but they exist outside the system and thus there is no built-in support for state-based proofs and reasoning about temporal issues.

HOL is strong in the area of data abstraction. HOL's type definition package maintains the soundness of HOL while at the same time making the declaration of new types and functions on those types reasonably automatic.

The ability to define new types in HOL and then write specifications in terms of these new types gives HOL a powerful abstraction mechanism. Not all verification environments are so powerful. Some limit the specification to terms involving boolean logic, so that it is impossible, for example, to say abstractly that an ALU should sum its inputs.

HOL provides no formal support for reasoning about changes to a design. This means that a small change to the implementation of a design invariably necessitates redoing the proof. Depending on how the proof was done, this may be trivial, or may be a major undertaking. Many systems are structured enough that with the appropriate abstraction mechanisms, mechanisms for reasoning about changes to a design should be possible. This represents a fertile area of possible research.

Extending HOL HOL's extensibility is one of its strongest points. Indeed, one could argue that to use HOL is to extend it since one cannot prove a theorem without adding it to a theory. User's can add new theories, types, tactics, conversions, and so forth to tailor the system to their needs.

In addition to being extended by proving theorems, ML, the metalanguage of HOL, provides a mechanism for programming the theorem prover and defining new functions. HOL is a system that can be tailored to a particular problem domain. One could imagine, for example, a hardware verification environment built on top of HOL, linked to a CAD system.

HOL is extensible without the user having to modify the internal code. In addition, HOL protects the user from making extensions that will make the system unsound. For example, we discussed how tactics are validated. A user can write a new tactic, but if the validation does not work, then the tactic cannot be used to mistakenly conclude that an HOL term is true.

Proving Theorems in HOL HOL is a interactive theorem prover. It does not attempt to carry out proofs without user intervention. Proof systems that attempt to automatically generate proofs almost always fail, leaving the theorem prover in an unknown state. The user then adds another lemma and tries again. HOL does not require user assistance for every inference, but the user is expected to guide the proof. As an example, the proof script of an n-bit ALU we carried out is about 600 lines long, but produces a proof which takes over 220,000 primitive inference steps.

HOL has powerful induction schema that include both induction over natural numbers and structural induction over both built-in and user defined data types. HOL's data abstraction package includes a facility for automatically generating induction tactics for user defined types.

HOL needs to be expanded to include decision procedures for terms involving common types such as numbers and booleans. One often gets to a point in a proof that the answer is obvious, but there are hours of term manipulation left to be done. There are already existing algorithms for dealing with many of these situations and they should be included in HOL so that this kind of tedious work can be avoided where possible.

Software Engineering Features HOL has no built-in capability to reason about changes to completed proofs. This is an area of active research and new ideas could be easily tested in HOL due to its powerful metalanguage, ML.

HOL supports the modularization of proofs through the creation of *theories*. A theory is a collection of related theorems, types, definitions, and axioms. For example, the type definitions and theorems about natural numbers are collected in a theory called :num. HOL maintains a hierarchical dependency graph showing which theories depend on others. This hierarchy is used to provide modular proofs of correctness that limit the effect of changes to a system being verified.

HOL's type checker is strong and is used to maintain soundness. The typechecker has very good inferential capabilities, freeing the user from worrying about type declarations where they can be inferred from context.

HOL provides no means of executing even simple expressions. This can be very frustrating since one defines functions and cannot execute them over several example to get a feel for their correctness before starting on a long verification effort. Many times, one defines a function, only to discover part way through a proof that it is incorrect. This usually invalidates the proof so far and the user is obliged to start over. On a more general level, an execution facility can be used to give the specification writer feedback that the specification is correct. A formal specification that does not match the requirements is useless.

Concluding Remarks Overall, HOL is a very capable system that has proven itself through use in several large proofs.

There are tradeoffs, of course, in the design and implementation of theorem provers. HOL is extensible and programmable; there is only a small amount of code that must be trusted. EHDM and, to a greater extent, Boyer-Moore are handcrafted for performance; they attempt to automatically prove theorems. HOL is perhaps less focused and, in some situations, more difficult to use, but much more flexible and in the end stronger.

3.7 OBJ3

3.7.1 Overview

OBJ3 is an equational programming language designed for the specification and implementation of abstract data types. Recent literature, however, demonstrates that OBJ3 has many applications. In particular, the language and its interpreter support testing, debugging, and rapid prototyping of algebraic specifications of software systems [GW88]. More importantly, specifications can be mechanically verified, by using the interpreter as theorem prover [Gog88].

Because OBJ3 can be used for all phases of a development effort, it is a wide-spectrum language. OBJ3 is also a functional language, since user-defined operations cannot cause side-effects.

OBJ3's encapsulation unit is the *object*. The object construct corresponds roughly to the module or package constructs found in other programming languages.

An object has a name and can contain several kinds of components. These components define sorts and operations that are exported by the object. Sorts are analogous to data types; operations are analogous to functions. Specifically, an object can contain: importing instantiations of other objects; sort, subsort, variable, and operation declarations; and equations. Before continuing, however, some definitions are necessary.

The fundamental data structure in OBJ3 is the *term*. A term is defined recursively: It is either a variable, a constant (an operation with no arguments), or an operation with terms as arguments. A term containing no variables is a *ground term*. An OBJ3 computation is the *reduction* of a ground term. A reduction is the simplification of a term according to a set of equations. Operationally, equations are treated as unidirectional rewrite rules.

An object is *imported* when its sorts and operations are needed by the importer. OBJ3 supports several importation mechanisms — they are extremely flexible. These mechanisms promote top-down, modular design. They are discussed in more detail, later in this section.

A *sort* declaration is used to construct a user-defined sort. A *subsort* declaration allows an element of one sort to be an element of another sort.

A *variable* declaration allows variables to be used in equations. Each variable is declared to be of a particular sort. An equation containing a variable is an abbreviation for the set of equations obtained by replacing the variable with each element of its sort. Of course, an equation may contain more than one variable.

An *operation* declaration specifies the name, arity, and coarity of an operation. An operation can be overloaded, since its arity and coarity are part of its name. For a mixfix operation, the argument positions are declared by embedding underbars in the operation name (e.g., _+_). Various operation attributes can also be defined. These include: associativity, commutativity, a precedence, and an evaluation strategy. An evaluation strategy specifies the order in which the operation's arguments are reduced.

An *equation* specifies a relationship between two or more operations. Thus it specifies how a term can be reduced. In fact, an equation is simply an ordered pair of terms. If, after variable binding, the left-hand side of an equation is equal to a subterm, that subterm can be replaced by the equation's right-hand side, using the variable binding. Conditional equations are also supported. A conditional equation is applicable only when its associated boolean condition (a term that reduces to true or false) reduces to true.

3.7.2 Abstraction Mechanisms

OBJ3 supports a program-development paradigm called parameterized programming [Gog84]. When a parameterized object is instantiated, an actual parameter is bound to each formal parameter. An actual parameter is also an object (i.e., it provides sorts and operations). OBJ3's parameterization mechanism is far more powerful than Ada's generic-package mechanism.

Each actual parameter must satisfy the *theory* associated with its corresponding formal parameter. A theory is an object-like construct that specifies the syntactic and semantic requirements of an actual parameter.

In general, an actual parameter may satisfy a theory in more than one way. The way in which an actual parameter does satisfy a theory is specified by an object-like construct called a *view*. A view is a mapping from sorts and operations in the theory to sorts and operations in the actual parameter. Often, explicit views are unnecessary; OBJ3 is quite adept at determining default views.

3.7.3 Forms of Logic Supported

Strictly speaking, an OBJ3 specification is written in equational logic. Each equation specifies that one or more pairs of terms are equal, and therefore interchangeable. Practically speaking, however, OBJ3 provides several built-in objects that make writing a specification far easier. In addition, more complex objects can be imported from a library supplied with the interpreter.

The built-in objects define boolean, integer, rational, real, tuple, and string operations. Most are implemented by Lisp code for efficiency.

The library objects define sets, lists, arrays, complex numbers, quaternions, categories, and unification. A decision procedure for propositional calculus and a bubble-sort algorithm are also specified in OBJ3.

3.7.4 Formal System Basis

For a specification language to support verification, it must have a formal semantics. Otherwise, theorems cannot be legitimately proven. OBJ3 has both an operational semantics and a mathematical semantics.

The operational semantics of OBJ3 is based on *order sorted term rewriting*. Order sorted term rewriting defines the behavior of the interpreter's rewrite-rule engine.

The mathematical semantics of OBJ3 is based on *order sorted equational logic*. Order sorted equational logic is a restricted form of equational logic (a system with no sorts) and a generalized form of many sorted equational logic (a system with no subsorts). Equational logic is seen as too permissive, whereas many sorted equational logic is seen as too restrictive. This compromise has several benefits:

> Among the advantages of "strong typing", which of course we call strong sorting, are: to catch meaningless expressions before they are executed; to separate logically and intuitively distinct concepts; to enhance readability by documenting these distinctions; and, when the notion of subsort is added, to support overloading, coercions, multiple representations, and error handling, without the confusion found in many programming languages [GW88].

Order sorted equational logic is a rigorous mathematical theory. Of primary importance, however, is the way in which the sort of an operation is determined. When an operation name is parsed, its sort is chosen to be the smallest (i.e., lowest in the hierarchy) of those possible. But this choice may be changed to a larger sort (i.e., higher in the hierarchy), in order to apply an equation. Subsorting sometimes allows an operation to be defined as total, when it would otherwise be necessary to define it as partial; this greatly simplifies the operation's specification because it does not have to detect argument values that are outside of the intended domain. Thus, error handling is often automatic.

The formal semantics of an OBJ3 object is an *algebra*. An algebra is a collection of sets with functions among them. An object's sorts correspond to the algebra's sets, its constant operations of each sort correspond to the elements of each set, and its other operations correspond to the functions. Since OBJ3 is based on order sorted equational logic, the proof theory of that formal system applies directly to OBJ3 specifications.

For an object's operational and formal semantics to agree, its formal semantics must be an *initial algebra* [GM83]. An initial algebra is one where every element of every set can be named using the given functions (i.e., there is no "junk") and all true equations can be proven to be true using the given equations (i.e., there is no "confusion").

An object's formal semantics is an initial algebra if the object is *canonical*. An object is canonical if it is *confluent* and *terminating*. An object is confluent if the order of equation application does not effect the result. An object is terminating if every reduction terminates.

For many examples of interest, the rules of equational deduction are both sound and complete. In particular, this is true when every sort contains at least one constant. However, there are important examples without this property [GM83].

3.7.5 Specification Checking

An OBJ3 specification can be analyzed in (at least) three ways. It can be shown to be free of syntactic errors, it can be shown to have desirable application-independent properties, and it can be shown to have desirable application-specific properties. The first two classes of properties can be established by general-purpose tools, which are discussed here. The last class is established by verification, which is discussed later.

As the interpreter processes an object, it parses and sort-checks each equation according to the sort, subsort, variable, and operation declarations. Thus, it detects all syntactic and even some obvious semantic errors. Unfortunately, this is the only automatic analysis currently available.

An important feature that is missing from OBJ3 is a tool for determining if an object is canonical. In general, this question is undecidable, but in practice such a tool often succeeds. More specifically, two tools are necessary. The first tries to prove that an object is terminating; the second that a terminating object is confluent. Usually, termination is obvious and requires no proof. Confluence can be demonstrated by the Knuth-Bendix completion procedure [KB70]. This procedure can even generate new equations that make an object confluent. Unfortunately, the Knuth-Bendix procedure does not always terminate.

The Knuth-Bendix procedure has been implemented for a predecessor of OBJ3 named OBJ [Gog80]. Unfortunately, these two languages are substantially different. Perhaps a future version of OBJ3 will also provide tools to help check for termination and confluence.

Another useful tool that is not provided is an automatic test generator [Jal89]. Such a tool takes an object as input and produces as output an arbitrarily large set of terms. The terms are produced in order of increasing size so that all operations are covered. These terms can then be reduced by the interpreter to test the object.

3.7.6 Execution and Rapid Prototype Support

For a specification language to support testing, debugging, and rapid prototyping, it must be executable. An interpreter for OBJ3 specifications is available from SRI International; it is written in Kyoto Common Lisp.

The interpreter maintains a collection of currently defined *modules*. A module is either a parameterized object, an instantiated object, a theory, or a view. Initially, only the built-in modules are defined. The built-in modules provide arithmetic, logic, and string operations. By default, a reduction is performed in the context of the most recently instantiated object, but the interpreter can be instructed to reduce in the context of any defined object

Commands are given to the interpreter by typing them at a prompt. The in command causes a file of commands to be executed. Such a file can also contain in commands. The show command pretty-prints a module or displays system-parameter values. These parameters primarily control the interpreter's verbosity (e.g., reduction tracing); they are changed by the set command. The reduce command reduces a term. When it finishes, it displays the result and the number of rewrites performed.

In OBJ3, input is the term to reduce and output is the result of its reduction. This is the only form of input/output that is provided. Thus, rapid prototyping of computational algorithms is supported, but a rapid prototype cannot demonstrate a realistic user interface.

3.7.7 Verification and Theorem-Proving Support

There are three levels to consider when using OBJ3 as a theorem prover. Only the activities of first level are supported by the interpreter. Activities of the other two levels must be performed manually.

At the lowest level of verification, equations are applied to reduce terms. For a person, this work is tedious and error-prone. Fortunately, machines thrive on term substitution. Thus OBJ3's rewrite-rule engine can execute proof scores.

At the middle level of verification, these proof scores are constructed. A proof score contains module definitions and reductions that prove a theorem. More precisely, the theorem is only proven if all the reductions reduce to true. For example, a proof score might employ induction to prove that a property of a particular specification always holds.

Automatic construction of proof scores is an attractive prospect, especially if their correctness is guaranteed. One approach is to supply a library of generic proof scores that can be instantiated as needed. Another approach is to employ "proof management" software that understands proof tactics but relies on OBJ3's interpreter to do the real work.

At the highest level of verification, the correctness of proof scores is justified. Such justifications are usually very abstract and cannot possibly be done mechanically. The language employed is mathematics (e.g., that of initial algebras).

3.7.8 Low Water Mark Example

As a concrete demonstration of the OBJ3 language, this section presents a specification of the Low Water Mark security problem [CGHM81]. Following this specification, the security constraints are formulated as an invariant and an inductive proof is given that verifies that the specification establishes and maintains the invariant.

The problem is to control access to a shared database containing a single item of information.

```
theory ITEM is
    sort Item .
    op notfound : -> Item .
endth
```

Figure 1: The Requirements for a Database Item.

In accordance with access constraints, a user can read the data item, write a new item into the database (possibly lowering the database security level), or reset the database security level to its highest possible value. A reader's security level must not be lower than the database's level and a writer's or resetter's security level must not be higher than the database's level.

Rather than designing a specification for a database containing a particular sort of data, a parameterized specification is constructed. Only one parameter is needed, but when the specification is instantiated the parameter must supply a sort and an element of that sort to return as the result of a successful read of an initialized database. These requirements are specified by the theory in Figure 1. The theory requires an instantiating object to supply a sort named Item and a constant of sort Item named notfound.

The Low Water Mark specification is shown in Figure 2. Capitalization clearly counts and follows the recommended conventions: Module and variable names are all uppercase. Sort names are uppercase and lowercase. Operation names and keywords are all lowercase. Comments begin with three asterisks and continue to the end of the line.

The first line of LWM declares a parameter I that must satisfy the ITEM theory. The object then declares two sorts named Database and Level, imports a built-in object INT, and declares that the integers are a subset of the levels. Thus, integers can be used as levels. Sort Int is declared in object INT. The names, arities, and coarities of the operations provided by LWM are then declared. Operation new constructs an initialized database with a security level of ten, the highest possible value for this specification; read, write, and reset do what their names imply; rd-done, wr-done, and rs-done return the results of read, write, and reset (resp.); lwm bundles a security level with a data item to form a database; high is the highest security level; and none is the item returned to writers, resetters, and unauthorized readers. Following this "signature" of LWM, several variables are declared (variable I is distinct from parameter I). Finally, equations specify the behavior of the declared operations. The first two construct an initialized database; the next three do the actual reading, writing, and resetting according to the security constraints; and the last nine allow nested invocations to interact appropriately.

In order to test the specification in Figure 2, it must be instantiated. First, an object must be chosen as the parameter to LWM. Then, a view must be used to specify how that object satisfies the ITEM theory. And then, another object must be constructed that imports and instantiates LWM.

A simple example is a database of natural numbers. Figure 3 specifies how the built-in object NAT satisfies ITEM. Sort Item is satisfied by (mapped to) sort Nat and constant notfound is satisfied by (mapped to) constant 0. The view ITEM-TO-NAT implicitly imports ITEM and NAT. Figure 4 shows LWM being instantiated according to ITEM-TO-NAT. Sample reductions demonstrate the specification's operations.

There are three main steps in verifying that LWM correctly enforces the Low Water Mark security constraints. First, a *proof object* is constructed from the *specification object* in Figure 2. The proof object is then instantiated and augmented with a predicate on sort Database that expresses the security constraints. This predicate is then used with a form of structural induction [GHM78] to prove that the constraints are never violated.

```
object LWM[I :: ITEM] is
    sort Database .
    sort Level .
    protecting INT .
    subsort Int < Level .

    op  new : -> Database .
    op  read : Level Database -> Database .
    op  write : Level Item Database -> Database .
    op  reset : Level Database -> Database .

    op  rd-done : Item Database -> Database .
    op  wr-done : Item Database -> Database .
    op  rs-done : Item Database -> Database .

    op  lwm : Level Item -> Database .
    op  high : -> Level .
    op  none : -> Item .

    var L L1 : Level .
    var I I1 I2 : Item .

*** Create a new database.
    eq  new = lwm(high,notfound) .
    eq  high = 10 .

*** read requires ProcLev >= ObjLev; it does not change ObjLev
    eq  read(L,lwm(L1,I1)) =
        if L >= L1 then
            rd-done(I1,lwm(L1,I1))
        else
            rd-done(none,lwm(L1,I1))
        fi .

*** write requires ProcLev <= ObjLev; it sets ObjLev to ProcLev
    eq  write(L,I,lwm(L1,I1)) =
        if L <= L1 then
            wr-done(none,lwm(L,I))
        else
            wr-done(none,lwm(L1,I1))
        fi .

*** reset requires ProcLev <= ObjLev; it sets ObjLev to high
    eq  reset(L,lwm(L1,I1)) =
        if L <= L1 then
            rs-done(none,lwm(high,I1))
        else
            rs-done(none,lwm(L1,I1))
        fi .
```

Figure 2: The Low Water Mark Specification Object (1 of 2).

```
*** Eliminate completed invocations.
    eq  read(L,rd-done(I2,lwm(L1,I1))) = read(L,lwm(L1,I1)) .
    eq  read(L,wr-done(I2,lwm(L1,I1))) = read(L,lwm(L1,I1)) .
    eq  read(L,rs-done(I2,lwm(L1,I1))) = read(L,lwm(L1,I1)) .
    eq  write(L,I,rd-done(I2,lwm(L1,I1))) = write(L,I,lwm(L1,I1)) .
    eq  write(L,I,wr-done(I2,lwm(L1,I1))) = write(L,I,lwm(L1,I1)) .
    eq  write(L,I,rs-done(I2,lwm(L1,I1))) = write(L,I,lwm(L1,I1)) .
    eq  reset(L,rd-done(I2,lwm(L1,I1))) = reset(L,lwm(L1,I1)) .
    eq  reset(L,wr-done(I2,lwm(L1,I1))) = reset(L,lwm(L1,I1)) .
    eq  reset(L,rs-done(I2,lwm(L1,I1))) = reset(L,lwm(L1,I1)) .

endo
```

Figure 2: The Low Water Mark Specification Object (2 of 2).

```
view ITEM-TO-NAT from ITEM to NAT is
  sort Item to Nat .
  op notfound to 0 .
endv
```

Figure 3: Satisfying the Item Requirements.

```
object LWM1 is
    protecting LWM[ITEM-TO-NAT] .        *** instantiate
endo

reduce read(6,write(5,1111,new)) .
reduce read(4,write(5,1111,new)) .
reduce read(7,write(6,2222,read(9,write(8,1111,new)))) .
reduce read(5,write(6,2222,read(9,write(8,1111,new)))) .
reduce read(7,reset(6,write(5,1111,new))) .
reduce read(7,reset(4,write(5,1111,new))) .
```

(a): Input.

```
----------------------------------------------------------
reduce in LWM1 : read(6,write(5,1111,new))
rewrites: 9
result Database: rd-done(1111,lwm(5,1111))
----------------------------------------------------------
reduce in LWM1 : read(4,write(5,1111,new))
rewrites: 9
result Database: rd-done(none,lwm(5,1111))
----------------------------------------------------------
reduce in LWM1 : read(7,write(6,2222,read(9,write(8,1111,new))))
rev-ites: 17
result Database: rd-done(2222,lwm(6,2222))
----------------------------------------------------------
reduce in LWM1 : read(5,write(6,2222,read(9,write(8,1111,new))))
rewrites: 17
result Database: rd-done(none,lwm(6,2222))
----------------------------------------------------------
reduce in LWM1 : read(7,reset(6,write(5,1111,new)))
rewrites: 13
result Database: rd-done(1111,lwm(5,1111))
----------------------------------------------------------
reduce in LWM1 : read(7,reset(4,write(5,1111,new)))
rewrites: 14
result Database: rd-done(none,lwm(10,1111))
```

(b): Output.

Figure 4: Instantiating and Testing the Specification Object.

```
object PROOF-BOOL is
    protecting TRUTH-VALUE .
    protecting BOOL .

    op  eif_then_else_fi : Bool Bool Bool -> Bool
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Bool Bool -> Bool [strategy (1 2 0) prec 51] .

    ⋮
endo

object PROOF-TRUTH[X :: TRIV] is
    protecting TRUTH-VALUE .
    protecting PROOF-BOOL .

    op  eif_then_else_fi : Bool Elt Elt -> Elt
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Elt Elt -> Bool [strategy (1 2 0) prec 51] .

    ⋮
endo

object PROOF-REL[X :: TRIV] is
    protecting TRUTH-VALUE .
    protecting PROOF-TRUTH[X] .

    op  _<s_ : Elt Elt -> Bool [prec 51] .
    op  _<s=_ : Elt Elt -> Bool [prec 51] .
    op  _>s_ : Elt Elt -> Bool [prec 51] .
    op  _>s=_ : Elt Elt -> Bool [prec 51] .

    ⋮
endo
```

Figure 5: Signatures of Three Proof Tools.

An alternative approach is to verify the parameterized specification; thus any instantiation is also verified. Although this technique has been shown to be mathematically sound [Gog88], OBJ3's interpreter cannot perform reductions in the context of a parameterized object. Therefore, such a verification cannot currently be automated.

In order to formulate a predicate that captures the Low Water Mark security constraints, the proof object (and conceptually, the specification object) is modified to record a small amount of historical information. In particular, three pieces of information are saved as part of the database: the type of access made by the last successful invocation, the security level of that invocation, and the security level of the database just before that invocation succeeded. With this historical information available, the predicate can be formulated as an invariant of all constructable databases.

Aside from application-dependent reasons, like the history maintenance described above, the transformation from specification object to proof object is necessary because OBJ3's interpreter can only reduce ground terms. The trick is to replace variables with new operations called *variable operations* [GHM78]. Unfortunately, a term containing variable operations does not always reduce as expected. The problems are caused by the built-in conditional and relational operations. However, these can be replaced by the operations from PROOF-BOOL, PROOF-TRUTH, and PROOF-REL. The signatures of these objects are shown in Figure 5.

The new conditional operation is "eager" whereas the original is "lazy"; its evaluation strategy causes all of its arguments to be reduced before the boolean condition is tested. The new relational operations are "symbolic" whereas the originals are "literal"; their equations can reduce them to true but never to false.

Figure 6 shows the proof object. Operations rd, wr, and rs record history information as the last argument to lwm. Operations avar, lvar, and ivar are variable operations that can replace variables of sorts matching their coarities. Each variable operation has a single integer argument. This makes an infinite supply of variable replacements available (e.g., lvar(0), lvar(1), lvar(2), ...). The proof tools are then instantiated for each necessary sort. The imported operations are used in the equations as replacements for the built-in operations.

The proof object is just as executable as the specification object. In Figure 7, the proof object is instantiated for natural numbers and some sample terms are reduced. These tests are the same as those in Figure 4. This time, however, the results contain history information.

The Low Water Mark security constraints are embodied by operation invariant in object INVARIANT shown in Figure 8. After instantiating the proof object, INVARIANT defines invariant as a predicate on databases and instantiates a case-analysis rule for it. Case analysis is a second-order axiom [GHM78] that is instantiated as a *module expression*. This particular expression imports PROOF-CASE from Figure 9 and renames sort Sort to Database and operation p to invariant. The three equations for invariant cover each of the three possible types of database access. Variable P represents the process (user) security level. Variable O represents to object (database) security level.

With the invariant specified, a proof score employing structural induction can be constructed. The score is shown in Figure 10. The basis proves that the invariant holds for an initialized database. An object is then used to make an inductive hypothesis. Object INDUCTION imports INVARIANT and (transitively) the instantiated proof object. An equation then assumes that the invariant holds for a particular database. Variable operations are required here. With variables, the equation assumes that the invariant holds for all databases, but that is the theorem to prove. The new equations for rd-done, wr-done, and rs-done eliminate completed invocations, thereby allowing invariant to be evaluated. Three reductions are necessary for the inductive step. Each reduction proves that the invariant is preserved across a particular kind of access, whether it succeeds or not. The detailed trace of each reduction is about two pages long. The traces demonstrate that the automatic proof proceeds essentially as a manual proof would.

3.7.9 Critical Remarks

The previous sections simply describe OBJ3, this section tries to gauge its suitability and effectiveness as a specification and verification system. Both the language and its interpreter are considered. Language deficiencies are probably permanent, since correcting them probably requires changing the underlying formal system. Interpreter deficiencies are not so serious; fixing them just takes time.

Perhaps the most important characteristic of OBJ3 specifications is that they are easy to design, understand, and modify. To some extent, this is a characteristic of specifications written in any equational language. This clarity results from the simplicity of equational logic, where there are no sophisticated built-in operations or complicated models of computation. There is only one rule: a term may be replaced by an equal term. Of course, it is a waste of time to start every specification from scratch; libraries of specifications are developed and imported as needed.

Equational languages are functional languages. As such, they lack what programmers who use conventional (imperative) languages take for granted — and have a hard time giving up. Namely, a function can only return a single value, there is no global state, and there are no assignment statements. But it is exactly these familiar "features" that are responsible for most software

```
object LWM[I :: ITEM] is
    sort Database .
    sort Level .
    sort Action .
    protecting INT .
    subsort Int < Level .

    op  new : -> Database .
    op  read : Level Database -> Database .
    op  write : Level Item Database -> Database .
    op  reset : Level Database -> Database .

    op  rd-done : Item Database -> Database .
    op  wr-done : Item Database -> Database .
    op  rs-done : Item Database -> Database .

    op  lwm : Level Item Action -> Database .
    op  high : -> Level .
    op  none : -> Item .

    ops rd wr rs : Level Level -> Action .

*** Variable operations.
    op  avar : Int -> Action .
    op  lvar : Int -> Level .
    op  ivar : Int -> Item .

*** Proof tools.
    protecting PROOF-TRUTH[Database] .
    protecting PROOF-TRUTH[Level] .
    protecting PROOF-REL[Level] .
    protecting PROOF-TRUTH[Action] .

    var L L1 : Level .
    var I I1 I2 : Item .
    var A : Action .

*** Create a new database.
    eq  new = lwm(high,notfound,rs(high,high)) .
    eq  high = 10 .

*** read requires ProcLev >= ObjLev; it does not change ObjLev
    eq  read(L,lwm(L1,I1,A)) =
        cif L >= L1 then
            rd-done(I1,lwm(L1,I1,rd(L,L1)))
        else
            rd-done(none,lwm(L1,I1,A))
        fi .

*** write requires ProcLev <= ObjLev; it sets ObjLev to ProcLev
```

Figure 6: The Low Water Mark Proof Object (1 of 2).

```
    eq  write(L,I,lwm(L1,I1,A)) =
        eif L <s= L1 then
            wr-done(none,lwm(L,I,wr(L,L1)))
        else
            wr-done(none,lwm(L1,I1,A))
        fi .

*** reset requires ProcLev <= ObjLev; it sets ObjLev to high
    eq  reset(L,lwm(L1,I1,A)) =
        eif L <s= L1 then
            rs-done(none,lwm(high,I1,rs(L,L1)))
        else
            rs-done(none,lwm(L1,I1,A))
        fi .

*** Eliminate completed invocations.
    eq  read(L,rd-done(I2,lwm(L1,I1,A))) = read(L,lwm(L1,I1,A)) .
    eq  read(L,wr-done(I2,lwm(L1,I1,A))) = read(L,lwm(L1,I1,A)) .
    eq  read(L,rs-done(I2,lwm(L1,I1,A))) = read(L,lwm(L1,I1,A)) .
    eq  write(L,I,rd-done(I2,lwm(L1,I1,A))) = write(L,I,lwm(L1,I1,A)) .
    eq  write(L,I,wr-done(I2,lwm(L1,I1,A))) = write(L,I,lwm(L1,I1,A)) .
    eq  write(L,I,rs-done(I2,lwm(L1,I1,A))) = write(L,I,lwm(L1,I1,A)) .
    eq  reset(L,rd-done(I2,lwm(L1,I1,A))) = reset(L,lwm(L1,I1,A)) .
    eq  reset(L,wr-done(I2,lwm(L1,I1,A))) = reset(L,lwm(L1,I1,A)) .
    eq  reset(L,rs-done(I2,lwm(L1,I1,A))) = reset(L,lwm(L1,I1,A)) .

endo
```

Figure 6: The Low Water Mark Proof Object (2 of 2).

```
object LWM1 is
   protecting LWM[ITEM-TO-NAT] .        *** instantiate
endo

reduce read(6,write(5,1111,new)) .
reduce read(4,write(5,1111,new)) .
reduce read(7,write(6,2222,read(9,write(8,1111,new)))) .
reduce read(5,write(6,2222,read(9,write(8,1111,new)))) .
reduce read(7,reset(6,write(5,1111,new))) .
reduce read(7,reset(4,write(5,1111,new))) .        (a): Input.
```

```
reduce in LWM1 : read(6,write(5,1111,new))
rewrites: 11
result Database: rd-done(1111,lwm(5,1111,rd(6,5)))
```

```
reduce in LWM1 : read(4,write(5,1111,new))
rewrites: 11
result Database: rd-done(none,lwm(5,1111,wr(5,10)))
```

```
reduce in LWM1 : read(7,write(6,2222,read(9,write(8,1111,new))))
rewrites: 19
result Database: rd-done(2222,lwm(6,2222,rd(7,6)))
```

```
reduce in LWM1 : read(5,write(6,2222,read(9,write(8,1111,new))))
rewrites: 19
result Database: rd-done(none,lwm(6,2222,wr(6,8)))
```

```
reduce in LWM1 : read(7,reset(6,write(5,1111,new)))
rewrites: 15
result Database: rd-done(1111,lwm(5,1111,rd(7,5)))
```

```
reduce in LWM1 : read(7,reset(4,write(5,1111,new)))
rewrites: 16
result Database: rd-done(none,lwm(10,1111,rs(4,5)(b): Output.
```

Figure 7: Instantiating and Testing the Proof Object.

```
object INVARIANT is
    using LWM[ITEM-TO-NAT] .

*** Proof tools.
    op  invariant : Database -> Bool .
    using PROOF-CASE * (sort Sort to Database, op p to invariant) .

*** Low water mark invariant.
    var I : Nat .
    var L P O : Level .

    eq  invariant(lwm(L,I,rd(P,O))) = (P >= O and L == O) == true .
    eq  invariant(lwm(L,I,wr(P,O))) = (P <= O and L == P) == true .
    eq  invariant(lwm(L,I,rs(P,O))) = (P <= O and L == high) == true .

endo
```

Figure 8: The Low Water Mark Invariant.

```
object PROOF-CASE is
    sort Sort .

    op  p : Sort -> Bool .
    op  eif_then_else_fi : Bool Sort Sort -> Sort .
    op  eif_then_else_fi : Bool Bool Bool -> Bool .

    var B : Bool .
    var S1 S2 : Sort .

    eq  p(eif B then S1 else S2 fi) =
        eif B then
            p(S1)
        else
            p(S2)
        fi .
endo
```

Figure 9: The Case-Analysis Proof Tool.

```
*** Basis

reduce invariant(new) .

*** Inductive hypothesis

object INDUCTION is
    using INVARIANT .
    var I : Nat .
    var D : Database .
    eq  invariant(lwm(lvar(1),ivar(1),avar(0))) = true .
    eq  rd-done(I,D) = D .
    eq  wr-done(I,D) = D .
    eq  rs-done(I,D) = D .
endo

*** Inductive steps

reduce invariant(read(lvar(0),lwm(lvar(1),ivar(1),avar(0)))) .
reduce invariant(write(lvar(0),ivar(0),lwm(lvar(1),ivar(1),avar(0)))) .
reduce invariant(reset(lvar(0),lwm(lvar(1),ivar(1),avar(0)))) .
```

(a): Input.

```
==========================================
reduce in INVARIANT : invariant(new)
rewrites: 10
result Bool: true
==========================================
object INDUCTION
==========================================
reduce in INDUCTION : invariant(read(lvar(0),lwm(lvar(1),ivar(1),avar(
    0))))
rewrites: 11
result Bool: true
==========================================
reduce in INDUCTION : invariant(write(lvar(0),ivar(0),lwm(lvar(1),ivar(
    1),avar(0))))
rewrites: 11
result Bool: true
==========================================
reduce in INDUCTION : invariant(reset(lvar(0),lwm(lvar(1),ivar(1),avar(
    0))))
rewrites: 13
result Bool: true
```

(b): Output.

Figure 10: Verification of the Invariant by Induction.

defects.

As advertised, OBJ3's sort system is a realistic compromise between anarchy and tyranny. Probably the best way to make a specification understandable is to use mnemonic sort names from the problem domain. A strict sort system also helps to promote a disciplined style of specification. Nevertheless, subsorting, overloading, and coercion relax many of the unnecessary restrictions found in typical systems. Overloading is also heavily relied upon for defining proof tools for the example verification.

OBJ3's parameterization mechanisms are more powerful than those found in any other language. Theories, views, and module expressions allow very generic specification libraries to be developed and reused. Even on its own, this sublanguage is worthy of merit. The sublanguage, and the programming paradigm it supports, promotes and enforces hierarchical and modular designs.

Even though OBJ3 does not come with a conventional theorem prover, its interpreter can do much of what a theorem prover is supposed to do. This leads to an interactive style of proof development that is pragmatic, intuitive, and subject to user control. In addition, a failed proof attempt often suggests corrections or lemmas by providing a readable trace of the failed attempt. Reading such a trace really is the best way to debug a proof; provers that report only "proved" or "unproved" provide no such help. In contrast to this interactive approach, fully automatic provers are sometimes difficult to steer, while fully manual provers require a user to perform too much tedious work.

As is, the interpreter provides only rudimentary verification support. Therefore, several improvements should be incorporated. Some of these improvements are already available in the Affirm specification and verification system [TE81]. Alternatively, Affirm would benefit by integrating some of OBJ3's features.

The rewrite-rule engine should be modified to enable it to reduce terms containing variables. These variables could be treated (internally) as variable operations. Hence, full unification is unnecessary. Affirm provides this capability.

The rewrite-rule engine should also be modified to accept recursion-limiting commands. These commands are often needed during verification because recursive equations cause nonterminating reductions during a proof. Such equations occur naturally, but they also result when a conditional equation is transformed into a regular equation. This transformation is also necessary for verification and should be automated. Affirm provides a restricted form of recursion limiting. Namely, it offers one-level rewriting.

A verification interface should also be added to the system. This could be a separate tool for constructing proof scores, or a shell that understands proof strategies and manages a proof attempt. Affirm allows executable "proof schemas" to be defined and provides a tree-based proof manager.

Tools that help prove that an object is canonical are also needed. Thus, an order-sorted version of the Knuth-Bendix completion procedure should be implemented. This should be provided as a separate tool. Affirm includes an implementation of the procedure, but insists that all specifications be canonical. It generates and adds completion equations while a specification is being input. This approach seems clumsy and dogmatic.

All in all, OBJ3's interpreter is quite efficient and relatively bug free. Error messages are generally informative, but sometimes they try to be more helpful than they are. For example, if the parser cannot decide between two parses it tries to print both possibilities, but they are often indistinguishable. At least it tries to be helpful. The interpreter also provides easy access to the underlying Lisp interpreter. OBJ3 purists may shudder, but this is a very important feature.

Even though the interpreter is intended to be run in an Emacs [Sta87] window, the user interface could be improved. For example, command-history and command-completion mechanisms would be convenient. There should also be a command to change what the interpreter considers as the "current directory" for in commands. It should also change the current directory while reading a file, so that nested in commands work as expected.

The reference manual is complete and well written. It describes the language, its extensive and interesting history, its operational and mathematical semantics, and explains many examples. A large bibliography is also provided. The Lisp code itself is also well organized and well documented. The value of this latter feature should not be underestimated.

3.8 EVES

3.8.1 Overview

EVES is an environment for coding verified programs being developed at I.P. Sharp Associates Limited, Ottawa. The aim of the project is a production quality verification system for programs satisfying NCSC A1+ requirements. The system was also required to have a rigorous proof of soundness, the designer's contention being that many theorem provers in existence embodied primitive and unsound concepts.

The method of proof consists in incrementally entering definitions of symbols and satisfying certain *proof obligations* indicated by the system.

The research can be divided into two main components: Verdi which is the language for specification and implementation (embodying the mathematical basis) and NEVER, the theorem prover developed for constructs in Verdi.

The approach has been to construct a prototype system – m-EVES, divided into m-Verdi and m-NEVER – and augment it to achieve the necessary power of EVES. Thus m-EVES provides an environment for research and instruction. It has been completed at the end of 1987, and work has continued since then towards the production quality EVES system. A fairly large proof regarding the Low Water Mark example has been published in [CKM*88].

3.8.2 Execution and Rapid Prototype Support

The language for specification and implementation (m-Verdi) includes many common programming features, based on Pascal-like constructs. It is a strongly-typed language, and presents constructs for annotations.

Communication with the devices which affect the program and which may be affected by it is provided by the *environment*, which introduces symbols and captures concepts forming an axiomatic basis for the program.

A compiler for m-Verdi already exists; it has been developed using the Karlsruhe Code Generator Synthesis System. Programs in m-Verdi are compiled into code for VAX machines. The compilerincorporates several optimizations, and it is also easily retargetable to other systems.

3.8.3 Abstraction Mechanisms

Abstraction and information hiding is provided in m-Verdi through the use of a *package* construct. Packages collect together a sequence of declarations. Some symbols may be hidden from the rest of the program, as well as the body of the package.

Packages are interesting in that they allow proofs to be deferred: while for other entities the proof obligations must be satisfied at the time they are being declared, a package header may be declared and used while the body of the package may be added later, at which time the proof obligations must be satisfied. To avoid circularity, proofs for must use the state existing immediately after the declaration is given.

3.8.4 Forms of Logic Supported

The logic behind EVES is based on Predicate Calculus, with extensions to allow for definition of recursive functions and introduction of new symbols.

Proofs in EVES consist of extensions to a *theory* – a set of symbols (the *vocabulary*) along with a set of axioms relating the symbols. Extensions to a theory are conservative, in the sense that meanings of symbols existing prior to the extension do not change. This is explained in more detain in [Cra87].

3.8.5 Verification and Theorem Proving Supported

The theorem proving style is based on the following components of the system:

- a simplifier: uses tautology checking, congruence closure, and linear programming techniques;
- a rewriter: supports conditional rewriting with backchaining, forward rules, and rules allowing permuted parameters;
- an invoker: conditionally expands function definitions based on heuristics;
- induction: based on the Boyer-Moore method for automatic induction.

The m-NEVER theorem prover allows for interactive development of proofs coupled to powerful automatic capabilities. The system allows easy transition from a proof-checking mode, with user input guiding the proof, to a largely automatic method, by invoking commands which allow the system to use more and more heuristics. Large proof steps can thus be obtained.

3.8.6 Specification Checking—Completeness, Consistency, and Soundness

Specifications in m-Verdi are based on the Floyd-Hoare style, using pre- and post-assertions. Heuristic information can be linked to prepositions.

Any symbol must be declared before being used. Proof obligations must be satisfied before a symbol can become part of the vocabulary. This ensures soundness.

All declarations are syntactically and semantically checked by the system. Recursive procedures must be well defined, and procedures' verification conditions must be proved. A procedure must be proved to terminate as well as to satisfy the post-condition.

3.8.7 Formal System Basis—Completeness, Consistency, and Soundness

The basis of EVES is the many-sorted predicate calculus, which provides a rigorous mathematical characterization. The semantics of the m-Verdi language is based on Denotational Semantics. The logic has been shown to be sound relative to the formal semantics.

Note that this does not imply that any proof obtained using the m-EVES system is correct, as there is no proof yet of the correctness of the implementation of m-EVES. There is still work to be done in obtaining a proven "proof-checker" for proofs obtained through m-EVES.

3.8.8 Examples

As a demonstration of the use of m-EVES, we will show parts of the verification of a mathematical algorithm: exponentiation using the fact that $x^y = (x^2)^{y/2}$ for even y. This proof has been obtained from [Cra88].

The declaration of the procedure which embodies this algorithm is given in Fig.1. Before this procedure can be added to the theory, the auxiliary function EXP must be defined, and several properties of DIV must be axiomatized.

```
!procedure FAST_EXPONENTIATION (pvar X,
                                pvar Y, pvar RESULT) =
  initial (X'0 = X, Y'0 = Y)
  post RESULT = EXP (X'0, Y'0)
  begin
    RESULT := 1
    loop
      invariant AND (INT'GE (Y, 0),
                     TIMES (RESULT, EXP (X, Y)) = EXP (X'0, Y,0))
      measure ORDINAL'VAL (Y)
        exit when Y = 0
        if MOD (Y, 2) = 0
          then X := TIMES (X, X)
               Y := DIV (Y, 2)
          else RESULT := TIMES (X, RESULT)
               Y := MINUS (Y, 1)
        end if
    end loop
  end FAST_EXPONENTIATION;
```

Figure 1: Procedure for exponentiation.

```
!function EXP (X, Y): INT =
  pre INT'GE (Y, 0)
  measure ORDINAL'VAL (Y)
  begin
    if Y = 0 then 1
             else TIMES (X, EXP (X, MINUS (Y, 1)))
      end if
  end EXP;
```

Figure 2: Function describing exponentiation.

```
Beginning proof of EXP...
IMPLIES (AND (Y <> 0,
              INT'GE (Y, 0)),
         ORDINAL'LT (ORDINAL'VAL (MINUS (Y, 1)),
                     ORDINAL'VAL (Y)))

!reduce;

Which simplifies
when rewriting with ORDINAL'LT_6 to ...
TRUE
```

Figure 3: Proof of function EXP.

The body of EXP is shown in Fig.2. As this is a recursive function, EVES requires a proof that it terminates. This is done by showing that the measure expression (in this case ORDINAL'VAL (Y) strictly decreases.

When the body of EXP is entered, the system shows what proof obligations should be fulfilled, as shown in Fig.3. The only resulting obligation is completed by the simple rule reduce.

For the proof of FAST_EXPONENTIATION it is useful to add INT'GE (DIV (X, Y), 0) as an assumption; this can be obtained by applying a forward rule (FRULE) to a proposition which contains DIV (X,Y). The declaration of this axiom, as well as the satisfaction of its obligations, is shown in Fig.4.

Note that, as with the Boyer-Moore prover, this system attempts to complete the proof through simplifications before invoking induction, and it finds the induction scheme automatically. The transcript has been simplified to show only salient points.

A few more axioms have been used for the proof; these will not be shown here. When the declaration of FAST_EXPONENTIATION is finally added, the proof obligation is satisfied by a simple application of reduction.

3.8.9 Critical Remarks

The EVES system has achieved substantial success in obtaining verified and executable programs. The notation used is rich enough to express several non-trivial programs and to write relevant specifications and propositions. Because of the origins of m-Verdi in programming languages such as Pascal, expressing programs in this language is quite natural.

The language however does not have any constructs to express concurrency, and neither is there an inclusion of polymorphism or higher-order functions (though these might be included in the final version of EVES).

It is also important to remember that the correctness of the implementation has not been proved. This correctness is quite essential in a system of the size of EVES, which incorporates many different rewrite tools and heuristic methods (i.e. there is a lot of code which must be trusted).

```
!axiom DIV_NONNEGATIVE (X, Y) =
  FRULE
  triggers (DIV (X, Y))
  begin IMPLIES (AND (INT'GE (X, 0),
                      INT'GE (Y, 0)),
                 INT'GE (DIV (X, Y), 0))
  end DIV_NONNEGATIVE;

Beginning proof of NONNEGATIVE ...
IMPLIES (AND (INT'GE (X, 0),
              INT'GE (Y, 0)),
         INT'GE (DIV (X, Y), 0))

!prove by induction;

Which simplifies to ...
   .
   .
   .
Returning to :
Beginning proof of DIV_NONNEGATIVE ...
Inducting using the following scheme ...
   .
   .
   .
produces ...
   .
   .
   .
Which simplifies
with invocation of DIV to ...
TRUE
```

Figure 4: Proof of an axiom about DIV.

3.9 Gypsy

3.9.1 Overview

Gypsy is an integrated system of methods for formal program specification, implementation and verification. It provides precise means of expressing a program throughout all stages of its design from initial formal specification, implementation, verification, to subsequent evolution. Gypsy also provides modularity, incremental development, abstract data types, and support for process concurrency which includes synchronizing process communication and real-time dependencies.

3.9.2 Execution and Rapid Prototype Support

The environment for Gypsy is called the Gypsy Verification Environment (GVE). This environment is currently run on Symbolics 3600 and MULTICS. The two general services of the environment are maintaining library and providing tools. It maintains in database a library of Gypsy program, specification, and verification condition, etc.. It also provides the tools for implementing the specification, programming, and verification methods.

The GVE contains several major subsystems to support the development and verification of programs : a Gypsy database manager, a parser, an edit interface, a verification condition generator, a theorem prover, a program optimizer, a Bliss translator, and an Ada translator. In addition to all these, Gypsy has an overall top level executive which monitors the verification status of all units in the database and guides the user through the verification process.

3.9.3 Abstraction Mechanisms

Gypsy is derived from Pascal. However, some of the rules in Pascal has been changed in order to simplify the verification processes. Gypsy does not allow nested routines, variable parameters, or routines as parameters to other routines. This eliminates any side effects that may happen, thus simplifies verification and increases the potential for optimization of expression evaluation.

Gypsy has the features that support unit-by-unit manipulation, increase unit independence, and isolated unit interactions. The Independent Principle in Gypsy says that "the proof of a routine may only depend upon its own specifications and implementation, and upon the external specifications of the routines to which it textually refers". The Independent Principle insures that the implementation of one routine cannot interfere with the proof of any other routine.

3.9.4 Forms of Logic Supported

The verification in Gypsy can be done by formal proof, by run time validation, or they may simply be assumed. Specification that are proved or assumed need not be evaluated at run time, and therefore they are permitted to contain special operations and types that could not otherwise be permitted.

3.9.5 Verification and Theorem Proving Support

There is a Theorem Prover in the Gypsy verification system which interactively assists in proving the verification conditions. The Theorem Prover is mostly interactive and has commands that attempt to complete a proof automatically. It is a tool that is easy to guide through a proof and easy to follow. It can also display a "proof tree" which shows the steps of the proof.

3.9.6 Specification Checking

In Gypsy, it is possible to give formal, mathematical proofs that a program satisfies its specifications. The Verification Condition Generator in Gypsy is designed so that it is always possible to construct automatically a set of theorems (verification conditions), that are sufficient to prove the consistency of a program's implementation and its specifications. If these formulas can be proved, then whenever the program runs, its implementation causes an effect that satisfies its specifications.

3.9.7 Support and/or Adaptability for Concurrency

Gypsy also allows both the specification and the coding of concurrent processes. The process in Gypsy is the same as the procedure except that it only allows message buffer as parameter.

Message buffer is a finite length queue in which there are only two operations: send (enqueue) and receive (dequeue). The buffer uses a straight first in first out algorithm and all the operations are mutually excluded in time.

The communication between the processes in Gypsy is done straightly through message buffer. Buffer in Gypsy is a predefined structure, it may be declared locally or passed as parameter. Gypsy keeps a history of all the transactions that are performed to a buffer. The send and receive statements append the transaction to the appropriate local history with the time of transaction stamped on the record. Therefore, the complete history of process interactions can be analyzed by examining the histories of message traffic among processes.

3.10 Nuprl

3.10.1 Overview

The Nuprl system is the result of several experiments using Proof Refinement Logics at Cornell University. It has been used primarily as a tool to study constructive type theory as applied to mathematics.

Nuprl represents a different approach towards formally verifiedprograms. It is based on constructive logic: to prove that there exists an entity with a certain property one shows how such an entity may be obtained. Propositions in this logic therefore have a computational content. More details of the constructive approach czn be obtained in [Con85].

The system is quite powerful as a result of the use of the ML metalanguage. This language, which also serves as the basis of other systems such as HOL and LCF-LSM, is very adequate for expressing proof generating programs. What differentiates Nuprl from these other systems is its reliance on constructive type theory.

3.10.2 Execution and Rapid Prototype Support

The Nuprl language allows for an explicit reference to the computational content of propositions in its logic. This is given by using the `term-of` operator. When applied to a proof outline (or a complete proof) it extracts the information needed for execution. In Nuprl the theorem prover behaves more like a compiler than an evaluator.

The computational content arises from the *extraction form* associated with every Nuprl proof rule. Nuprl terms obtained from extraction terms have constructs corresponding to those in standard programming languages.

The process of *evaluating* a term consists in taking the *noncanonical* form of an extracted term and *reducing* it to obtain a term closer to a canonical form.

3.10.3 Abstraction Mechanisms

There is no explicit notion of abstraction, as the language is a very simple functional notation. On the other hand the computational content of the proof – the *extract form* – is hidden from the user, and manipulated mechanically by the system. This is not rigid however – termss can be explicitle manipulated by the system.

3.10.4 Forms of Logic Supported

The type theory forming the basis of Nuprl is derived from typed lambda calculus, a functional notation supporting higher order logic. The lambda calculus has been extended to model a richer type structure, including dependent types, propositions as types, and layering of type universes into *large types.*

The semantics of Nuprl is therefore linked to the concept of types – equality of types, membership in a type, equality within a type, and so on.

The power of type theory allows us to use it to model several other logics. The type constructors of Nuprl – disjoint union, cartesian products, etc – can be used to express logical connectives in classic logic.

Nuprl also has built in definitions for integers and lists, along with appropriate induction forms; this allows for easy modeling of number theory. A set constructor is also implemented primitively in the logic.

3.10.5 Verification and Theorem Proving Supported

The methodology used for proofs in Nuprl is top-down, based on the concepts of refinements and *tactics*. These tactics are metalanguage procedures which manipulate terms by embodying the rules of inference available. Tactics can be used to manipulate proofs, obtain subterms, and perform many other operations on the object language entities.

Refinement is provided by applying a rule to a goal and obtaining subgoals, which will be the next objects to which rules must be applied. Rules may also completely prove a goal. The proof is therefore tree-structured.

The form of theorem proving relates to the concept of *propositions as types* – the proof must show that the type specified by the proposition is inhabited. This is done by using types which are known to be inhabited – say the integer type – and using type constructors to obtain the type characterizing the proposition. The user must show that the type is inhabited by some element; this element turns out to be the proof itself.

The rules which manipulate goals are specified in three ways: parsep 0in

- *intro* rules: break down the conclusion of goals into subgoals;
- *elim* rules: apply a specified hypothesis;
- *reduce* rules: rules for computation.

3.10.6 Specification Checking—Completeness, Consistency, and Soundness

Problem statements are written in the typed lambda calculus notation. Specifications constitute the top node of the proof trees. All the nodes of this tree have a *sequent* – a goal and an associated list of hypotheses.

The soundness of definitions in Nuprl follows from the soundness of type theory. Types defined by users must be proved to be well formed.

3.10.7 Formal System Basis—Completeness, Consistency, and Soundness

The logic of Nuprl is based on Intuitionistic logic – classic logic without the law of the excluded middle. The absence of this law as a proof method implies that proofs based on contradiction are not allowed. Any function definable in this Intuitionistic logic has been proved by Kleene to be Turing computable.

Type theory as such is a result of a large amount of mathematical and philosophical research aimed at a logic which is powerful enough to represent large amounts of formal knowledge and yet providddess soundness and consistency.

3.10.8 Examples

The first example we show here is from [C*86] and involves a proof of the law of the excluded middle, that is that for a proposition P either P or P is true, but not both. This exlusive-or is represented by the Nuprl operator *vel*.

Figure 5shows the beginning of this proof. the subgoals have been formed by the system. Goals which are labeled by a # indicate incomplete proofs, while those marked with a * indicate goals which have been proved. Sequents are represented as hypotheses separated from goals by ">>".

```
# top
>> all P:U1. P vel ~P

BY intro at U2

1# 1.  P:(U1)
   >> (P vel ~P)

2* >> (U1) in U2
```

Figure 5: Proof of excluded middle

```
#  top 1
1. P:(U1)
>>  (P vel ~P)

BY intro at  U1

1# 1. P:(U1)
   2. ~P&~~P
   >> void

2* 1. P:(U1)
   >> ~P&~~P in U1
```

Figure 6: Proof of the law of excluded middle, cont.

The first tactic applied consists in generalizing the type of P to indicate an arbitrary propositional variable. Application of the intro rule results in a second subgoal of type membership, which is usually solved automatically.

The empty type is indicated by void; the proposition shown in Figure 6therefore says that ~P&~~P denotes an empty type.

The rest of the proof is straightforward manipulation, and is shown in Figure 7.

The previous proof did not provide any computational content, it is simply a proof about the logic. In Figure 8we show a proposition and the evaluation of its extract term. The body of the proof is not being shown for conciseness. The proposition indicates that for any two integers, either they are equal or there exists another integer which added to one of them results in the other. The computational content of this is a procedure which gives the difference beetwen two integers.

The evaluator has an ML interface, thus the notation looks slightly different from the proof environment. Functions are indicated by a lambda calculus notation using \ as an approximation to λ. The word axiom refers to the leaves of the proof with no computational content. We will not explain details of the functional notation used. The expression thm refeers to the theorem just proved.

The Nuprl system uses windowing environments, and these transcripts are just an approximation of the actual interaction.

3.10.9 Critical Remarks

This system has evolved mostly as a tool for reasoning about mathematics andd incorporating several interesting reults in typee theory. As such there hasn't been much concern with producing

```
# top 1 1
1. P:(U1)
2. ~P&~~P
>> void

BY elim 2

1# 1. P:(U1)
   2. ~P&~~P
   3. ~P
   4. ~~P
   >> void

BY elim 4

1* 1. P:(U1)
   2. ~P&~~P
   3. ~P
   4. ~~P
   >> ~P

2* 1. P:(U1)
   2. ~P&~~P
   3. ~P
   4. ~~P
   5. void
   >> void
```

Figure 7: Proof of the law of excluded middle, cont.

```
*  top
>> all x,y:int.. (x=y in int)|some x:int. ~(z=0 in int)#(x+z=y in int)

....

eval

>term_of(thm) ;;
\x.\y. (\E.decide(E;l.inl(l);
          r.inr(<y-x,<\f.(\v0.any(v0))(r(axiom)),axiom>>)))
         (int_eq(x;y ;inl(axiom);inr(axiom)))
>let d = \y..decide(y;u.p1(u);u.p1(u)) ;;
\y..decide(y;u.p1(u);u.p1(u))
>d(term_of(thm)(7)(10)) ;;
3
```

Figure 8: Proposition and its extract term

production quality software. The main aim instead has been to explore methodologies.

On the other hand it has been applied to verification of hardware, of transformations, and programs, without using the elegant concepts described above, for example considering propositions as types. Without these the interaction with Nuprl becomes very similar to that with proof checkeers such as HOL.

The use of Nuprl for verification of real programs of significant complexity is yet to be achieved. Instead it provides a different paradigm for reasoning about problems and structuring information logically.

3.11 VDM

The Vienna Development Method (VDM) is a well-established representative of the model-based approach to specification [Bjo78] [Jon78] [Bjo80] [Jon86] [Jon80] [CHJ86] [Jon83]. Its evolution began in 1966 with the Universal Language Description language, an attempt to formally define PL/I. In 1969, the Vienna Development Language was developed as a method for formally defining abstract interpreters. Then in 1974, VDM was developed as a denotational approach to specifying software systems; its metalanguage, Meta-IV, was also defined at this time.

With a model-based approach, a specification is a model constructed from well-defined primitives that maintain a state. A specification's state is a composite data object that is only visible to the specification's operations. An operation can be invoked by a user to effect changes to the state.

These primitives can be abstract or concrete. Using concrete primitives results in a rather operational specification (i.e., a specification can look like code).

VDM's metalanguage, META-IV, is considered to be "enriched" logic. Theorem proving is done after an algorithmic transformation to raw logic.

VDM's built-in data types and operations make it very similar to the denotational approach to specifying programming-language semantics. Data objects are defined by abstract mathematical types. Operations are defined either implicitly or explicitly. A specification is refined by making data-object representations more concrete. This is called data reification. Operations are then redefined for the reified data structures. The relationship between a specification and an implementation is recorded in a retrieve function and invariants are proven by structural induction.

VDM provides a rich variety of built-in data types and associated operators. Scalars are the simple boolean, numeric, and enumerated types found in most programming languages. Sets, tuples, maps, and trees are the structured types. A tuple is a cartesian product; it can model a pair, a list, or a record. A map is a finite-domain function; it can model an array or a hash table.

A specification can employ implicit or explicit operations. Implicitly defined operations use applicative constructs. For example, recursive functions can define predicates that appear as preconditions and postconditions. Explicitly defined operations use imperative constructs. VDM's explicit operations include: assignment with `:=`, selection with `if-then-else`, iteration with `while-do` and `for-all-do`, nondeterminism with a "parallel" statement, and sequencing with `;`.

4 Design of a Prototype Short-Term Workbench

4.1 Overview

Based on the knowledge gained from our survey, we have begun the design and implementation of a prototype short-term workbench. The goal for the workbench is to provide computer-based support for:

- The high-level specification of a concurrent/distributed software system
- The verification of the high-level specification, with emphasis on the verification of security properties
- The implementation-level specification of a concurrent/distributed software system
- Verification that the implementation-level specification is correct with respect to the high-level specification
- Implementation of the specified software in an efficient, concurrent/distributed programming language
- Verification that the implementation is correct with respect to the implementation-level specification (and thereby correct with respect to the high-level specification)

For the overall architecture we have chosen a two-tiered approach to specification, as developed originally in the Larch system [Win87], [Hor85], [GHW85]. Since Larch was developed solely for specification of sequential programs, a major component of our research is to add capabilities to specify concurrent and distributed programs.

Figure 1 is a high level diagram of the major conceptual levels of the workbench. The top two levels are the separate tiers of the specification, and the bottom level is the concrete, executable program. The algebraic tier of the specification expresses the highest levels of software abstraction. In general, the algebraic tier presents specifications in an object-oriented style, in which the basic unit of specification is an object description. Accompanying each object specification is a set of basic operations on that object, plus a set of equations that formally specify how the operations behave.

The axiomatic tier of the two-tier specification provides the interface between the high-level algebraic abstractions and the concrete programming language. In our Larch-inspired approach, the axiomatic tier is represented concretely by an axiomatic semantic description of the SR programming language.

The annotations shown in italics in Figure 1 describe the framework for formal verification in a two-tiered system. Specifically, the objects and operations are verified to be complete and consistent at the algebraic tier. Such verification is based on known techniques from the study of algebraic semantics. Once the operations have been validated at the abstract algebraic level, they can be used to specify pre- and post-conditions to be satisfied by the (SR) program at the axiomatic level.

A significant advantage of the two-tier approach is that critical properties of the software system can be stated at the appropriate level of abstraction. The algebraic tier of a specification is the level at which high-level properties of a software system and stated and verified. For example, abstract properties of security can be stated and verified at the algebraic level. Once these properties are established and verified at the abstract algebraic level, more concrete properties of the program can be stated and verified at the axiomatic tier. For example, one would state and verify that the program correctly implements the security properties that were verified abstractly at the algebraic level.

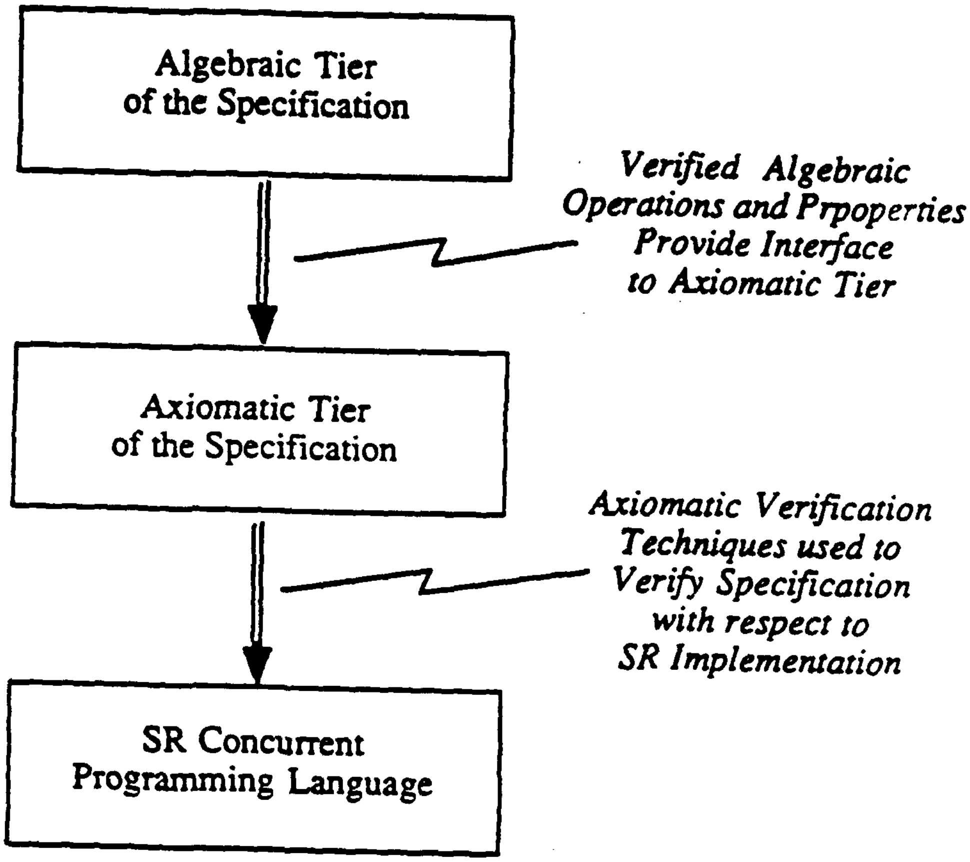

Figure 1: Three Conceptual Levels of the Specification and Verification Framework

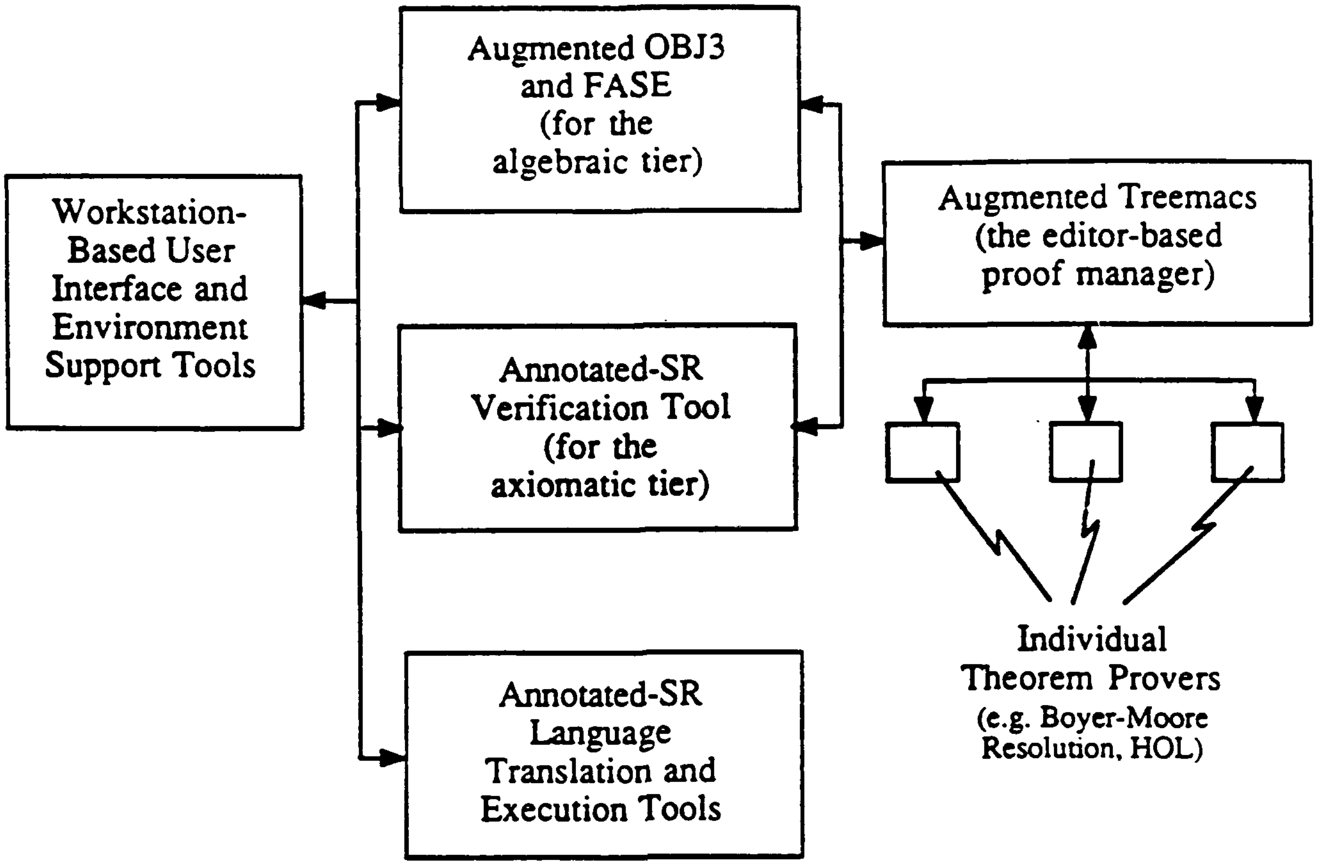
Augmented OBJ3 and FASE (for the algebraic tier)
Workstation-Based User Interface and Environment Support Tools
Augmented Treemacs (the editor-based proof manager)
Annotated-SR Verification Tool (for the axiomatic tier)
Individual Theorem Provers
(e.g. Boyer-Moore Resolution, HOL)
Annotated-SR Language Translation and Execution Tools

Figure 2: Tool Components of the Prototype System

Based on our two-tiered framework, Figure 2 shows the tool components which comprise our initial design of a prototype specification and verification system. The tool to support the algebraic tier will be a system such as OBJ3, augmented with features to support the specification of concurrency. The SR-based verification tool will assist in the management of the axiomatic specification of the SR program. "Annotated-SR" refers to a version of the language with syntactic enhancements to support the use of verification annotations attached directly to the program. Such annotations include pre- and post-conditions, module invariants, and other axiom-based assertions.

High-level control of the specification and verification are provided by the editor-based proof manager, and a set of general software development tools common to an advanced, workstation-based software environment. It should be noted that while the Treemacs proof manager is shown as a physically separate component in Figure 2, it is in fact a well-integrated component of the general environment tools. In particular, as the name "Treemacs" suggests, the proof manager is based on the same general-purpose Emacs editor that is used for all other forms of specification and program editing. This form of uniform, editor-based interface is common in most advanced software development environments today.

The roles of the key components shown in Figure 2 are introduced briefly just below. Then in sections 4.2 through 4.5 we present extended descriptions of our research on key components of the workbench.

4.2 The Role of the FASE, HOL, and OBJ3 High-Level Specification Languages

In our two-tiered approach, the top tier consists of an object-oriented, high-level specification language. There are several such languages available for the specification of sequential programs. At present we are pursuing a multi-language approach for the high-level specification tier. The languages we are using are FASE, HOL, and OBJ.

There is much recent support in the software engineering literature for mixed-language software development environments. The advantages of a mixed-language system stem from difficulty in finding a a single, fully general-purpose language that satisfies all users needs. Rather, it is more likely that a family of more specialized languages can be put together in common environment, allowing users to choose the language most suited to a particular problem and/or the language with which a user has the most experience.

Most of the work reported in the literature discusses mixed-language environments at the level of the concrete programming language. Here we are extending the mixed-language concept to higher-level specification languages. Hence, in our workbench we will allow user the freedom to choose from a set of specification languages, having the environment supply a well-defined interface between components of a specification written in different languages. One of the areas of research to be performed is how and at what level to define the intra-language interface. We will conduct this research based on our experience with mixed-language programming environments.

It is possible as the development of the workbench progresses, we will find that the mixed-language approach is not satisfactory, either because of the excess overhead to requires to manage mixed-languages specifications or because technical difficulties arise in developing the intra-language interface. At present, however, the mixed-language approach is advantageous and we plan to pursue as long as possible.

One of the best known is the OBJ3 language, developed at SRI by Goguen and his associates. We have been gaining experience with OBJ3, and find it a useful vehicle for high-level specification. We are not yet as firmly committed to OBJ3 as we are to SR, since there are several other algebraic languages that have been implemented that we plan to evaluate thoroughly before making our final decision. We can say that our initial experience with OBJ3 has been positive.

Another main component of our research is the extension of a high-level specification language to support the specification of concurrent programs. Section 4.2 below describes the details of our ongoing work with OBJ3, including extensions to handle specification of concurrency in an algebraic framework. While the current work is in the concrete framework of OBJ3, it is conceptually applicable to other algebraic languages, including potentially FASE and HOL.

4.3 The Role of the SR Programming Language

SR (Synchronizing Resources) is a language for writing concurrent programs. It provides more flexibility in the way processes synchronize and communicate than do other languages, including Ada. For example, SR provides both synchronous and asynchronous operation invocation whereas Ada provides only synchronous; having the asynchronous form simplifies many programs.

There are two primary rationale for choosing SR as our base programming language. The first and most important rationale is purely technical - we feel that SR provides one of the most conceptually sound and complete vehicles in which to express concurrent and distributed programs. While there are other languages that express certain features of concurrency well, SR is reasonably unique in its support of a full range of concurrency features in a conceptually uniform manner.

The second rationale for the choice of SR is pragmatic. Namely, several of our team members have considerable experience with the language and its implementation. Since UC Davis is one of the primary development sites for the language, we are in a position to design language extensions that may be necessary to fit our evolving specification and verification methodology. No other language affords us as clear an opportunity for evolving the language to fit our methodological needs.

4.4 The Role of the Annotated-SR Verification Tool

This is the component of our prototype methodology that is the least well-developed at this point. Tool support for axiomatic verification has been developed almost exclusively for sequential as opposed to concurrent programming languages. By tool support in this context, we mean those tools that produce theorems (actually conjectures, pending their proof) corresponding to properties the user proposes to verify about the program. These tools are conventionally called verification condition generators. Special purpose verification condition generators have been developed for particular properties of sequential programs, such as information flow related to multilevel security. For the verification of concurrent programs, the verification condition generator must consider all possible interleavings of operations. This can lead to formidable theorems, the management and simplification of which will be addressed in our research.

The initial necessary step to an axiomatic verification support tool is the development of the formal axiomatic semantics of our base language – namely SR. This is a substantial research effort in its own right, details of which are covered in section 4.5 below. Once our work on the semantics has developed, we will begin to consider the problem of verification condition generation for concurrent programs. This research will involve a blending of the work of several current researchers who have addressed different aspects of the axiomatic verification of concurrency.

4.5 The Role of TED/Treemacs Theorem Proving Support Systems

The role of the theorem prover is to verify the verification conditions. Numerous mechanical theorem provers are now available and we have been gaining practical experience with several. Also available is the TED/Treemacs [1] system that supports the management of the theorem proving process. TED/Treemacs is a tree-editor-based system that can provide a uniform interface to a variety of theorem provers and allows users to conveniently structure and carry out proofs.

In the context of program verification, it is convenient to view four classes of theorem provers:

- first-order logic-based, using resolution as the primary rule of inference (there are many of these available, the best supported being those developed at Argonne and the University of Illinois)
- first-order logic-based, using natural deduction (e.g., the Bledsoe theorem prover as used in the Affirm and Gypsy verification systems)
- first-order based, with user-supplied instantiations (e.g., Muse and the theorem prover of the Enhanced HDM verification system)

[1] TED and Treemacs are based on fundamentally the same principles; Treemacs is the successor to TED, containing some updated capabilities and better integration with the current release of the Emacs editor.

- recursive function-based (e.g., Boyer-Moore and LCF theorem provers).

As our methodology for concurrent verification evolves, we will consider how each of classes can and should be used in the verification task.

As noted above, we have not yet determined what role temporal logic will play in the methodology we develop. It seems reasonably clear at this point that one or two fundamental properties of temporal logic are necessary to reason completely about concurrent programs at the axiomatic specification level. We are not sure at all what role, if any, temporal logic may play at the algebraic specification level. If some form of temporal logic or the equivalent is necessary at some level of specification, this will have a definite impact on the theorem proving support required.

5 Algebraic Specification and Verification of Concurrency in OBJ

As described in the introduction, we have begun initial experimentation with algebraic specifications for concurrency within the framework of the OBJ3 system. Specifications at the algebraic level have several appealing properties, including:

1. The algebraic semantics are founded on fully formal mathematical principles

2. Algebraic specifications are executable via rewrite rule interpretation

3. Support for formal verification is provided, also in the form of rewrite rule application

4. A layered approach to specification is possible within the general object-oriented framework of algebraic specification; this layering fosters the definition of understandable specifications, and helps subdivide the refinement and verification of specifications into manageable, logical steps

5.1 Overview of the Approach

A major goal of research in the verification methodology is the development of mathematical techniques for demonstrating that a program is correct. These techniques are even more valuable when they can be applied to the verification of a concurrent program. In this section, we investigate how an equational language can be used to specify concurrency and how its interpreter can be used as a theorem prover to demonstrate correctness.

At first glance, automatic program verification appears to offer a tremendous benefit: bug-free software. Upon closer examination, however, verification techniques can only detect a small class of the many ways in which a program may fail to execute as desired [Gog88]. Nevertheless, this class is important. In particular, it usually contains the class of failures that are detectable by the programmer's test suite.[1] Testing a sequential program is not always easy; thoroughly testing a concurrent program is often very difficult. Therefore, the study of automatic verification techniques for concurrent programs is a worthwhile pursuit.

There are several approaches to specification; a popular dichotomy is the axiomatic approach and the algebraic approach [GM86a]. Among the algebraic methods, those with first-order equations as their assertion language have the simplest semantics. Such specifications are easy to write and understand; they are also relatively easy to interpret. Thus an equational specification is executable. In practice, this is very important. Notice that an executable specification is essentially a program. Equational languages are often advertised, therefore, as "very high-level" or "wide-spectrum" languages.

OBJ3 is just such a language. Its advantages over other equational languages are: module parameterization, operation overloading, subsorting, and availability. These are all important software-engineering features. The language is described in Section 5.2.

The main body of this section contains the results of experiments that use OBJ3 as a verification tool. Section 5.4 describes a technique for modeling concurrency in OBJ3, Sections 5.3 and 5.5 employ the technique to develop a specification for the classic Readers/Writers problem. Section 5.6 verifies an important property of this specification.

[1] Experience shows that a user is more prone to program abuse than the programmer.

Section 5.7 proposes extensions to OBJ3 that provide a message-passing mechanism reminiscent of that found in CSP. A message-passing version of the Readers/Writers problem is also presented.

A concurrent program is composed of atomic actions, which can each be proven correct by existing sequential methods. Section 5.8 revisits an early paper on sequential-program verification and demonstrates that OBJ3 can automate many of the steps. Here, the verification is somewhat different: an implementation is proven correct with respect to a specification.

Section 5.9 recognizes that the results presented here are preliminary and, therefore, identifies future research directions. These include implementing the extensions proposed in Section 5.7 and attempting larger and more complex examples.

5.2 The OBJ Language

This section provides a very brief overview of OBJ3. If a seemingly esoteric feature is described here, it is probably used in a later section. For more details, consult the reference manual [GW88]. OBJ3 is an equational programming language designed for the specification and implementation of abstract data types. Its denotational semantics are based on order sorted algebra, its operational semantics are based on order sorted term rewriting. An interpreter, written in LISP, is available from SRI International.

OBJ3's encapsulation unit is the *object*. An object can contain: importing instantiations of other objects; sort,[2] subsort, operation, and variable declarations; and equations. An object can be parameterized.

An operation declaration specifies the name, arity, and coarity of an operation. For a mixfix operation, the argument positions are declared by embedding underbars in the operation name (e.g , `_+_`). Various operation attributes can also be defined. These include associativity, commutativity, a precedence, and an evaluation strategy. An evaluation strategy specifies the order in which arguments are evaluated; it allows operations to be "eager" (the default) or "lazy". For example, the evaluation strategy for `if_then_else_fi` is `strategy (1 0)`. The numbers in parentheses correspond to argument positions. Thus, the first argument is reduced first; the 0 then causes equations for `if_then_else_fi` to be applied. Since there is no 2 or 3, the `then` and `else` subterms are not reduced at this time.

An equation specifies how a term can be reduced. That is, it specifies a relationship between two or more operations. If, after variable binding, the LHS of an equation is equal to a subterm, that subterm can be replaced by the equation's RHS using the variable bindings. Conditional equations are also supported. A conditional equation is applicable only when its associated boolean condition is true.

Figure 1 contains an object that defines a stack of integers; it also shows some sample reductions. The keyword `protecting` is a form of import. `INT` is a built-in object that provides sort `Int` and the expected operations on integers. OBJ3 contains various built-in objects defining booleans, naturals, integers, reals, and alphanumeric identifiers. Polymorphic equality (`_==_`), inequality (`_=/=_`), and conditional (`if_then_else_fi`) operations are also provided. The `op` keyword introduces an operation declaration. For example, `new` is a constant operation that represents the empty stack and `top` is a function that takes a single argument of sort `Stack` and returns an integer. Variables are declared with the `var` keyword. The `eq` keyword introduces an equation; `cq` is used for a conditional equation. Here, the equations define stack-operation semantics, including (pathetic)

[2] Algebraists prefer "sort" over "type".

```
object STACK-OF-INT is sort Stack .
    protecting INT .
    op new : -> Stack .
    op push : Int Stack -> Stack .
    op pop : Stack -> Stack .
    op top : Stack -> Int .
    var S : Stack .
    var I : Int .
    eq pop(new) = new .
    eq pop(push(I,S)) = S .
    eq top(new) = 0 .
    eq top(push(I,S)) = I .
endo
```

(a): Object.

```
reduce in STACK-OF-INT : top(pop(push(1,push(2,new))))
rewrites: 2
result NzNat: 2
```

```
reduce in STACK-OF-INT : top(pop(push(1,new)))
rewrites: 2
result Zero: 0
```

```
reduce in STACK-OF-INT : pop(push(3,push(2,push(1,new))))
rewrites: 1
result Stack: push(2,push(1,new))
```

(b): Reductions.

Figure 1: A Stack of Integers.

```
object NUM-HIERARCHY is
    sort Int .
    sort Nat .
    subsort Nat < Int .
endo
```

Figure 2: An Example of Subsorting.

error handling. Two sample reductions demonstrate the operations' behavior. Incidentally, OBJ3 has two kinds of comment, both extending to the end of the line. Those introduced by ***> print during parsing, those introduced by *** do not.

OBJ3 is a strongly sorted language; however, it allows subsorts to be declared and permits operation overloading. Subsorts are useful when items of one sort are also items of another sort. For example, Figure 2 might be part of a number hierarchy. This allows an operation that requires an argument from the supersort (the integers) to be applied to an argument from the subsort (the naturals). Subsorts make OBJ3 much more flexible and they simplify the specification of error and exception handling. Operation overloading is useful when a single operation name is used for more than one operation. The intended operation is determined by the (lowest) sort of each argument; it can also depend on the required result sort. For example, Figure 3 contains an overloaded infix addition operation. Overloading allows an equation to be applicable only when the arguments are of the appropriate sort. Here (evidently), the overhead of complex addition is incurred only when an argument is actually complex.

OBJ3 supports parameterized programming. When a parameterized object is instantiated, an actual parameter (an object) is bound to each formal parameter. Each actual parameter must satisfy the *theory* associated with its corresponding formal parameter. A theory defines the requirements of an actual parameter. The way in which the actual parameter satisfies the theory is specified by a *view*.[3] A view is a mapping from sorts and operations in the theory to sorts and

[3] OBJ3 is quite clever in deducing default views.

```
object COMPLEX-ADDITION is
    sorts Complex Real .
    op _+_ : Complex Complex -> Complex .
    op _+_ : Complex Real -> Complex .
    op _+_ : Real Complex -> Complex .
    op _+_ : Real Real -> Real .
    op ccadd : Complex Complex -> Complex .
    op cradd : Complex Real -> Complex .
    op rradd : Real Real -> Real .
    var C : Complex .
    var R : Real .
    eq C + C = ccadd(C,C) .
    eq C + R = cradd(C,R) .
    eq R + C = cradd(C,R) .
    eq R + R = rradd(R,R) .
endo
```

Figure 3: An Example of Overloading.

```
theory ELEM is sort Elem .
    op undefined : -> Elem .
endth

object STACK-OF-ELEM[E :: ELEM] is sort Stack .
    op new : -> Stack .
    op push : Elem Stack -> Stack .
    op pop : Stack -> Stack .
    op top : Stack -> Elem .
    var S : Stack .
    var E : Elem .
    eq pop(new) = new .
    eq pop(push(E,S)) = S .
    eq top(new) = undefined .
    eq top(push(E,S)) = E .
endo

view ELEM-TO-INT from ELEM to INT is
    sort Elem to Int .
    op undefined to 0 .
endv

make STACK-OF-INT is STACK-OF-ELEM[ELEM-TO-INT] endm
```

Figure 4: Another Stack of Integers.

operations in the actual parameter. Figure 4 defines a stack of integers as before, but by instantiating a generic stack object with the INT object. ELEM is a theory requiring the parameter E[4] to provide a sort named Elem and a constant of sort Elem named undefined. ELEM-TO-INT is a view describing how INT satisfies ELEM. The sort mapping is the obvious one; zero is designated as the top of an empty stack, as before. Finally, the make construct instantiates a stack of integers; it is equivalent to the object in Figure 5.

[4] Parameter E is distinct from variable E declared later in the object. In fact, the parameter is not explicitly referenced in the object. However, if it was necessary to instantiate, say, a list of elements within the object, the import would be protecting LIST[E], where E is the parameter.

```
object STACK-OF-INT is
    protecting STACK-OF-ELEM[ELEM-TO-INT] .
endo
```

Figure 5: An Alternative Method of Instantiation.

```
object SUMINC is protecting INT .
    op f : Int Int -> Int .
    op g : Int -> Int .
    var I J : Int .
    eq f(I,J) = I + J .
    eq g(I) = I + 1 .
endo
```

(a): Object.

```
reduce in SUMINC : f(g(1),g(2))
rewrites: 6
result NzNat: 5
```

(b): Reduction.

Figure 6: An Opportunity for Concurrent Term Rewriting.

5.3 The Readers/Writers Problem

A good example of a small concurrent programming problem is the Readers/Writers problem [Fin86]. The problem is to provide an interface to a shared database. A solution must arbitrate access to this database between competing processes in such a way that the integrity of the data is maintained. There are two kinds of processes: readers traverse but do not modify the database, writers traverse and modify the database. Since the internal structure of the database is unspecified, data integrity is maintained by coarse access restriction. Specifically, if nr and nw are the number of reader and writer processes currently accessing the database (resp.), the invariant in Equation 1 must always be true.

$$0 \leq nr \wedge 0 \leq nw \leq 1 \wedge (nr = 0 \vee nw = 0) \tag{1}$$

The key point is that a writer requires exclusive access to the database. Notice that the scheduling method for blocked processes is unspecified; we use first-come/first-serve (FCFS) scheduling.

5.4 Star Operations

This section describes how OBJ3 can be used to specify the behavior of a concurrent system. Some clumsiness should be expected since we are using a language that has no explicit concurrency mechanisms to explicitly model concurrency. This is in contrast to th: concurrent term rewriting approach [GKM87], where implicit opportunities for concurrency are detected within a sequential specification. These opportunities are easy to detect in OBJ3 because it is a functional language (i.e., operations cannot cause side-effects). Such an opportunity is shown in Figure 6. In the reduction, the equation for g can be applied concurrently to reduce the two arguments to f, but the equation for f cannot be applied concurrently with the equation for g. Concurrent term rewriting is an alternative to the Von Neumann model of computation because it avoids the program-counter bottleneck. Incidentally, the approach can also be used to increase the execution speed of a concurrent specification developed using the techniques described below.

The first obstacle to overcome is related to process naming. As in other sequential languages, all OBJ3 operations are invoked in the context of a single anonymous process. However, we want to reduce terms representing operation invocations from multiple processes. An important observation is that we are not interested in the result returned by every invocation; rather, we are interested in the result returned by a particular invocation, as affected by the other invocations. Therefore, we need some way to identify the invocation we are interested in. This is done by invoking a *star operation*, one with the same name except for a prefixed asterisk.[5] Generally speaking, each

[5] There is nothing special about the naming convention, any scheme could be used, but a prefixed asterisk makes the correspondence obvious and it stands out nicely in a term. As far as OBJ3 is concerned, however, a star operation

```
object SEMAPHORE is sort Semaphore .
    protecting INT .
    subsort Int < Semaphore .
    ops *p p *v v : Semaphore -> Semaphore .
    ops done outside : Semaphore -> Semaphore .
    var S : Semaphore .
    var I : Int .
    cq *p(I) = done(I - 1) if I > 0 .
    eq *v(I) = done(I + 1) .
    cq p(I) = I - 1 if I > 0 .
    eq v(I) = I + 1 .
    eq v(*p(S)) = *p(v(S)) .
    eq *v(p(S)) = p(*v(S)) .
    eq v(p(S)) = p(v(S)) .
    eq p(done(S)) = done(S) .
    eq v(done(S)) = done(S) .
    eq outside(done(I)) = I .
endo
```

(a): Object.

```
reduce in SEMAPHORE : outside(v(*p(0)))
rewrites: 10
result Zero: 0
```

```
reduce in SEMAPHORE : outside(v(v(p(*p(p(0))))))
rewrites: 23
result Zero: 0
```

```
reduce in SEMAPHORE : outside(*p(v(v(v(p(0))))))
rewrites: 19
result NzNat: 1
```

```
reduce in SEMAPHORE : outside(v(*p(p(0))))
rewrites: 11
result Semaphore: outside(*p(0))
```

```
reduce in SEMAPHORE : outside(*v(p(p(p(0)))))
rewrites: 10
result NzNat: 1
```

(b): Reductions.

Figure 7: A FCFS Semaphore.

"exported" operation[6] provided by an object requires a corresponding star operation; its definition is often similar and sometimes identical to that of its nonstarred namesake.

Unfortunately, there are complications. For example, when an exported operation relies on an auxiliary operation, an auxiliary star operation is often necessary to ensure that intermediate results from other invocations do not interfere with the result we are interested in. Furthermore, other operations are usually necessary to reduce a term containing the result of an invocation of a star operation to that result. That is, the intermediate results of other invocations must be eliminated. Specifically, the result must be moved toward the outside of a term where, it can participate in the final reduction step.

Figure 7 contains an object that uses star operations to model the behavior of a FCFS semaphore; it also shows some sample reductions. SEMAPHORE provides the usual P and V operations; but because an OBJ3 operation must always return something, they return the semaphore's value.

is no different from any other.

[6] Unfortunately, OBJ3 provides no way of declaring that an operation is inaccessible to other objects. Thus, all operations are exported. Only some operations, however, are intended for outside use, others are auxiliary in nature and should not be invoked from outside the object. Note that an earlier version of OBJ3 did support hidden operations [GM87].

```
theory KEY is
    sort Key .
endth

theory ITEM is
    sort Item .
    op notfound : -> Item .
endth
```

Figure 8: The Requirements for Database Parameters.

Remember, this is only a model; as such it requires an interpretation. The remainder of this section informally discusses this interpretation.

For a sequential specification, nested operations in a term model the history of invocations made by a single process; that is, they model the time-ordering of invocations. With multiple processes, however, a natural extension is to interpret nested operations in a term as the history of concurrent invocations, exactly one of which is a star operation. Thus in the first reduction in Figure 7, the *p operation models a *P* operation that blocks on a zero semaphore until a *V* operation is subsequently invoked by some other process. The term then reduces to a value that models the abstract semaphore's value just before the *P* would have returned. A *v operation behaves analogously. Notice how the term is reduced to the return value; the interaction of done and outside strongly resembles exception handling in Lisp. The value returned by an invocation of a p or v operation is immaterial; what is important is the invocation's effect on the semaphore's value.

Although the object in Figure 7 is somewhat muddied by the star operation's and their support, the important semantics are still there. The conditional equations for *p and p ensure a nonnegative semaphore; the equations allowing *v and v invocations to "pass by" *p and p invocations model *V* operations that execute while *P* operations are blocked; in contrast, the lack of reordering equations for *p and p invocations enforces the FCFS protocol.

Finally, consider atomicity. A semaphore implementation usually employs an atomic test-and-set instruction. A fact for OBJ3, and a natural assumption for any equational language, is that equation application is atomic. This is necessary to ensure that terms remain well formed.

5.5 Specification of Readers/Writers

The technique used for the SEMAPHORE object of Figure 7 can be generalized to specify the Readers/Writers problem. For concreteness, the database proper is specified as an associative list; however, it is strongly sorted. For reusability, the key and item sorts are specified by the object's parameters. They are described by the theories in Figure 8. The first theory requires a KEY parameter to provide only a sort Key whereas the second requires an ITEM parameter to provide a sort Item and a constant notfound. Item notfound is returned by an unsuccessful read operation.

Figure 9 is a parameterized specification for the Readers/Writers problem. Its various parts are described in the following paragraphs.

Operations *new and new simply construct an empty database, shown below.

```
rw(0,0,eod)
```

The rw operation encapsulates two semaphorelike counters and the database proper. The counters record the number of readers and writers (resp.) currently accessing the database proper; eod abbreviates "end-of-database". Operation rec stores database records. When a record is written to the database, its key and item components are stored as the first two arguments (resp.) to rec;

```
object READERS-WRITERS[K :: KEY, I :: ITEM] is
    sort Database .
    protecting INT .
    subsort Item < Database .

    op  *new : -> Database .
    op  *rd : Key Database -> Database .
    op  *wr : Key Item Database -> Database .

    op  new : -> Database .
    op  rd : Key Database -> Database .
    op  wr : Key Item Database -> Database .

    op  rw : Int Int Database -> Database .
    op  eod : -> Database .
    op  rec : Key Item Database -> Database .

    op  *rdout : Item Database -> Database .
    op  *wrout : Item Database -> Database .
    op  rdout : Item Database -> Database .
    op  wrout : Item Database -> Database .

    op  outside : Database -> Item .

    var D : Database .
    var K K1 : Key .
    var I I1 : Item .
    var NR : Int .      *** number of readers
    var NW : Int .      *** number of writers

*** Create a new database.
    eq  *new = rw(0,0,eod) .
    eq  new = rw(0,0,eod) .

*** Enter the database.
    cq  *rd(K,rw(NR,NW,D)) = rw(NR + 1,NW,*rd(K,D))
        if NW == 0 .
    cq  *wr(K,I,rw(NR,NW,D)) = rw(NR,NW + 1,*wr(K,I,D))
        if (NR == 0) and (NW == 0) .
    cq  rd(K,rw(NR,NW,D)) = rw(NR + 1,NW,rd(K,D))
        if NW == 0 .
    cq  wr(K,I,rw(NR,NW,D)) = rw(NR,NW + 1,wr(K,I,D))
        if (NR == 0) and (NW == 0) .

*** Exit the database.
    eq  rw(NR,NW,*rdout(I,D)) = *rdout(I,rw(NR - 1,NW,D)) .
    eq  rw(NR,NW,*wrout(I,D)) = *wrout(I,rw(NR,NW - 1,D)) .
    eq  rw(NR,NW,rdout(I,D)) = rw(NR - 1,NW,D) .
    eq  rw(NR,NW,wrout(I,D)) = rw(NR,NW - 1,D) .
```

Figure 9: The Readers/Writers Specification Object (1 of 3).

```
*** Return results.
    eq  outside(*rdout(I,D)) = I .
    eq  outside(*wrout(I,D)) = I .

*** Read traversal algorithm.
    eq  *rd(K,eod) = *rdout(notfound,eod) .
    eq  *rd(K,rec(K1,I1,D)) =
        if K == K1 then
            *rdout(I1,rec(K1,I1,D))
        else
            rec(K1,I1,*rd(K,D))
        fi .
    eq  rd(K,eod) = rdout(notfound,eod) .
    eq  rd(K,rec(K1,I1,D)) =
        if K == K1 then
            rdout(I1,rec(K1,I1,D))
        else
            rec(K1,I1,rd(K,D))
        fi .

*** Write traversal algorithm.
    eq  *wr(K,I,eod) = *wrout(I,rec(K,I,eod)) .
    eq  *wr(K,I,rec(K1,I1,D)) =
        if K == K1 then
            *wrout(I,rec(K,I,D))
        else
            rec(K1,I1,*wr(K,I,D))
        fi .
    eq  wr(K,I,eod) = wrout(I,rec(K,I,eod)) .
    eq  wr(K,I,rec(K1,I1,D)) =
        if K == K1 then
            wrout(I,rec(K,I,D))
        else
            rec(K1,I1,wr(K,I,D))
        fi .

*** Passing equations.
    eq  rec(K1,I1,*rdout(I,D)) = *rdout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,*wrout(I,D)) = *wrout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,rdout(I,D)) = rdout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,wrout(I,D)) = wrout(I,rec(K1,I1,D)) .

    eq  *rd(K1,rdout(I,D)) = rdout(I,*rd(K1,D)) .
    eq  *rd(K1,wrout(I,D)) = wrout(I,*rd(K1,D)) .
    eq  *wr(K1,I1,rdout(I,D)) = rdout(I,*wr(K1,I1,D)) .
    eq  *wr(K1,I1,wrout(I,D)) = wrout(I,*wr(K1,I1,D)) .

    eq  rd(K1,*rdout(I,D)) = *rdout(I,rd(K1,D)) .
    eq  rd(K1,*wrout(I,D)) = *wrout(I,rd(K1,D)) .
    eq  rd(K1,rdout(I,D)) = rdout(I,rd(K1,D)) .
```

Figure 9: The Readers/Writers Specification Object (2 of 3).

```
    eq  rd(K1,wrout(I,D)) = wrout(I,rd(K1,D)) .
    eq  wr(K1,I1,*rdout(I,D)) = *rdout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,*wrout(I,D)) = *wrout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,rdout(I,D)) = rdout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,wrout(I,D)) = wrout(I,wr(K1,I1,D)) .

endo
```

Figure 9: The Readers/Writers Specification Object (3 of 3).

the third argument is the rest of the database. An eod terminates the "list" of records.

The *rd, *wr, rd, and wr operations do what their names suggest; their uniform coarities are discussed later. The *rdout, *wrout, rdout, and wrout operations are necessary for decrementing the rw counters upon database exit. Specifically, *rd, *wr, rd, and wr "bubble into" the database until the desired key or eod is found, whereupon they reduce to *rdout, *wrout, rdout, and wrout (resp.) and bubble back out of the database, decrementing the appropriate counter as they pass rw.

The use of outside here is analogous to its use in the SEMAPHORE object from Figure 7.

READERS-WRITERS defines a rather curious subsort relation. Specifically, the object declares a sort Database as a supersort of Item. To understand why this is necessary, consider the coarities of the operations.

Intuitively, a read should return an item and a write should return nothing (a write modifies the database). However, an OBJ3 operation must return something, so both *rd and *wr (hence *rdout *wrout) return items; *rd returns the item read and *wr returns the item written. In contrast, both rd and wr (hence rdout wrout) must return a database. This is because the result returned by an inconsequential reader or writer — a rd or wr — must be accessed by other readers and writers.[7] Clearly then, rd, wr, rdout, and wrout all require a coarity of Database. And even though the coarity of *rd, *wr, *rdout, and *wrout is Database, the subsort relation Item < Database allows them to return items.

The four conditional equations enforce database-entry synchronization; compare them to Equation 1 on page 101. They also ensure that the appropriate counter is incremented. Conversely, the next four equations decrement a counter when an invocation leaves the database. Notice how a rdout or wrout disappears after it passes rw. However, a *rdout or *wrout eventually bubbles to the outside of the term where one of the next two equations — those for outside — finish the reduction. Next come equations that implement the typical traversal algorithms for a read or write operation. The remainder of the equations allow certain operations to pass by each other. If more sophisticated scheduling is desired, additional equations of a similar nature could be used (e.g., to reorder read and write invocations). Alternatively, rw could be given an additional argument — a list of blocked invocations — that would be ordered according to the scheduling discipline.

The number of equations would be greatly reduced if OBJ3 supported second-order equations. The OBJ3 alternative — parameterization and instantiation — is hardly worth the effort here.

In order to test a READERS-WRITERS database, key and item views are needed. The views in Figure 10 are for a database whose keys are alphanumeric identifiers[8] and whose items are natural numbers (zero represents a nonexistent item). An instantiation and some sample reductions are also shown.

5.6 Verification of the Readers/Writers Invariant

In the last section, OBJ3 was used to specify a concurrent programming problem and to test the specification on a few samples of input data. Although bugs found during testing are guaranteed to be bugs, testing is not guaranteed to find all the bugs. Program verification strives to be more thorough. Accordingly, this section describes OBJ3's use as a verification tool. In particular, we prove that the operations provided by the READERS-WRITERS object of Figure 9 maintain the truth of Equation 1 on page 101.

[7] With regard to the result of a reduction, a rd is a no-op, but a wr may have an effect.

[8] QIDL provides "qualified long identifiers" (viz., LISP symbols).

```
view KEY-TO-QIDL from KEY to QIDL is
  sort Key to Id .
endv

view ITEM-TO-NAT from ITEM to NAT is
  sort Item to Nat .
  op notfound to 0 .
endv
make RW is READERS-WRITERS[KEY-TO-QIDL,ITEM-TO-NAT]
```

(a): Object.

```
reduce in RW : outside(•rd('a,wr('a,1,new)))
rewrites: 22
result NzNat: 1

reduce in RW : outside(•rd('a,wr('b,1,new)))
rewrites: 24
result Zero: 0

reduce in RW : outside(•rd('a,wr('a,2,wr('a,1,new))))
rewrites: 34
result NzNat: 2

reduce in RW : outside(•rd('a,wr('a,2,wr('b,1,new))))
rewrites: 40
result NzNat: 2

reduce in RW : outside(•rd('b,wr('a,2,wr('b,1,new))))
rewrites: 36
result NzNat: 1

reduce in RW : outside(•wr('b,3,wr('a,2,wr('b,1,new))))
rewrites: 38
result NzNat: 3

reduce in RW : outside(wr('b,3,wr('a,2,•wr('b,1,new))))
rewrites: 40
result NzNat: 1

reduce in RW : outside(wr('b,2,•rd('b,wr('b,1,new))))
rewrites: 35
result NzNat: 1

reduce in RW : outside(wr('b,2,rd('b,•wr('b,1,new))))
rewrites: 36
result NzNat: 1
```

(b): Reductions.

Figure 10: Instantiating and Testing a Database.

```
object DIVIDE is
    protecting RAT .
    ops divide div : Rat Rat -> Rat .
    op  undefined : -> Rat .
    var X Y : Rat .
    ops xvar yvar : -> Rat .     *** Variable operations
    eq  divide(X,Y) =
        if Y == 0 then
            undefined
        else
            div(X,Y)
        fi .
    eq  div(X,Y) = X / Y .
endo
```

(a): Object

```
reduce in DIVIDE : divide(xvar,yvar)
rewrites: 4
result Rat: xvar / r:Rat>NzRat(yvar)
```

(b): Reduction.

Figure 11: A First Attempt at Using Variable Operations.

There are three main steps in the verification. First, a *proof object* is constructed from the *specification object* in Figure 9. This proof object is then instantiated[9] and augmented with a predicate on sort Database expressing the Readers/Writers invariant. This predicate is then used with a form of structural induction to show that serviced invocations maintain the invariant.

5.6.1 Construction of the Proof Object

The transformation from specification object to proof object is necessary because the OBJ3 interpreter was not designed to be a theorem prover; modifications that eliminate the need for this step are considered later. The fundamental problem is that the interpreter can only reduce variable-free terms (i.e., ground terms). There exist more powerful reductions systems — unification systems — that can reduce terms containing variables [Lel88], but such flexibility is actually unnecessary here. The trick is to replace variables in a term with *variable operations* [GHM78]. A variable operation is a new constant operation whose coarity is the sort of the variable it replaces. Alternatively, a variable operation can be given an integer argument thereby allowing it to replace any number of variables of that sort.

Unfortunately, a term containing variable operations does not always reduce as expected. Consider the rudimentary proof object and reduction in Figure 11, where variables X and Y are replaced by operations xvar and yvar.[10] The problem is that OBJ3's built-in equality operation, _==_, returns false because yvar is not syntactically equal to 0. The solution is to replace the built-in equality operation with a *symbolic equality* operation, _=s=_. The symbolic equality operation reduces to true if its arguments are equal, but does not reduce to false (i.e., it does not reduce at all) if its arguments are unequal.

Unfortunately, the symbolic equality operation causes yet another problem, as demonstrated in Figure 12. This time, the culprit is the lazy if_then_else_fi operation. Its evaluation strategy directs the interpreter to reduce the if subterm first. If it reduces to true, the operation reduces to the then subterm; if it reduces to false, the operation reduces to the else subterm; but if it

[9] Verification of a parameterised object offers the additional advantage of proof reusability. Furthermore, it has been shown to be mathematically sound [Gog88]. Unfortunately, the OBJ3 interpreter cannot perform reductions in the context of a parameterised object. Therefore, automating such a verification seems difficult.

[10] The strange looking r:Rat>NzRat operation is a *retract*, part of OBJ3's subsorting mechanism. Ignore it.

```
object DIVIDE is
    protecting RAT + PROOF-TRUTH[RAT] .
    ops divide div : Rat Rat -> Rat .
    op  undefined : -> Rat .
    var X Y : Rat .
    ops xvar yvar : -> Rat .      *** Variable operations
    eq  divide(X,Y) =
        if Y =s= 0 then
            undefined
        else
            div(X,Y)
        fi .
    eq  div(X,Y) = X / Y .
endo
```

(a): Object

```
reduce in DIVIDE : divide(xvar,yvar)
rewrites: 1
result Rat: if yvar =s= 0 then undefined else div(xvar,yvar) fi
```

(b): Reduction.

Figure 12: Adding a Symbolic Equality Operation.

```
object DIVIDE is
    protecting RAT + PROOF-TRUTH[RAT] .
    ops divide div : Rat Rat -> Rat .
    op  undefined : -> Rat .
    var X Y : Rat .
    ops xvar yvar : -> Rat .      *** Variable operations
    eq  divide(X,Y) =
        eif Y =s= 0 then
            undefined
        else
            div(X,Y)
        fi .
    eq  div(X,Y) = X / Y .
endo
```

(a): Object

```
reduce in DIVIDE : divide(xvar,yvar)
rewrites: 2
result Rat: eif yvar =s= 0 then undefined else xvar / r:Rat>NzRat(yvar) fi
```

(b): Reduction.

Figure 13: Adding an Eager Conditional Operation.

reduces to neither `true` nor `false` (e.g., `yvar =s= 0`), the `if_then_else_fi` operation remains in the term. In the first two cases, the selected subterm is subsequently reduced; but in the latter case, neither subterm is reduced, even though both may be reducible. The solution is to replace the built-in conditional operation with an eager *conditional* operation, `eif_then_else_fi`. The eager conditional operation has an evaluation strategy that directs the interpreter to reduce all three arguments[11] before attempting to select a subterm.

The final version of the proof object is shown in Figure 13. Symbolic equality and eager conditional operations produce a more satisfying result.

Syntax and semantics for `_=s=_` and `eif_then_else_fi` are provided by PROOF-BOOL and PROOF-TRUTH. These are objects that roughly correspond to OBJ3's built-in BOOL and TRUTH objects. Figure 14 shows the signature of both objects, Section 5.10 lists them in their entirety. PROOF-TRUTH is parameterised; it can be instantiated for comparison and selection of elements of any sort. PROOF-BOOL is similar to — but more sophisticated than — `PROOF-TRUTH[BOOL]`. Notice how important overloading is. Without it, a different operation name would be needed for each

[11] Arguments are reduced left-to-right, but since an OBJ3 operation cannot cause side-effects, the evaluation order is unimportant.

```
object PROOF-BOOL is
    protecting TRUTH-VALUE .
    protecting BOOL .

    op  eif_then_else_fi : Bool Bool Bool -> Bool
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Bool Bool -> Bool [strategy (1 2 0) prec 51] .
    :
endo

object PROOF-TRUTH[X :: TRIV] is
    protecting TRUTH-VALUE .
    protecting PROOF-BOOL .

    op  eif_then_else_fi : Bool Elt Elt -> Elt
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Elt Elt -> Bool [strategy (1 2 0) prec 51] .
    :
endo
```

Figure 14: Signatures of PROOF-BOOL and PROOF-TRUTH.

sort. Also noteworthy are the evaluation strategy for eif_then_else_fi and the single equation for _=s=_. In comparison, the built-in TRUTH object gives if_then_else_fi the attribute strategy (1 0) and uses a built-in equation (i.e., LISP code) to always reduce a _==_ to true or false.

The elided portions of PROOF-BOOL and PROOF-TRUTH are primarily the familiar tautologies. Unfortunately however, some boolean simplification rules are schematic or higher order. They represent an infinite number of equations [GHM78]. For example, the "logical-substitution" rule says that all occurrences of an if subterm can immediately be reduced to true or false, if the occurrence is in a corresponding then or else subterm (resp.) — regardless of how deeply nested the occurrence is. In addition, since the equation sets are logically incomplete, special-purpose equations are often necessary. Consequently, equations are added as needed.

The final step in transforming a specification object into a proof object is to remove conditional equations. This is necessary because the boolean condition associated with a conditional equation probably uses _==_, which means it will not reduce properly in the presence of variable operations. A conditional equation of the form

    l = r if b

can be transformed into a regular equation of the form

    l = if b then r else l fi

allowing variable operations and the symbolic equality and eager conditional operations to be used as before. The transformed equation is recursive. Operationally, it implies a nonterminating reduction sequence. This is especially evident considering the eager evaluation strategy for eif_then_else_fi. A proof tool that prevents this unpleasant behavior is described later.

Figure 15 is the proof object that results from applying the abovementioned transformations to the specification object in Figure 9. Notice the systematic naming convention for variable operations, the multiple instantiations of PROOF-TRUTH, the equations using operations from PROOF-TRUTH, and the erstwhile conditional equations. Also, remember that the object is still uninstantiated.

```
object READERS-WRITERS[K :: KEY, I :: ITEM] is
    sort Database .
    protecting INT .
    subsort Item < Database .

    op  *new : -> Database .
    op  *rd : Key Database -> Database .
    op  *wr : Key Item Database -> Database .

    op  new : -> Database .
    op  rd : Key Database -> Database .
    op  wr : Key Item Database -> Database .

    op  rw : Int Int Database -> Database .
    op  eod : -> Database .
    op  rec : Key Item Database -> Database .

    op  *rdout : Item Database -> Database .
    op  *wrout : Item Database -> Database .
    op  rdout : Item Database -> Database .
    op  wrout : Item Database -> Database .

    op  outside : Database -> Item .

*** Variable operations.
    op  dvar : -> Database .
    op  kvar : -> Key .
    op  ivar : -> Item .
    op  nrvar : -> Int .
    op  nwvar : -> Int .

*** Proof tools.
    protecting PROOF-TRUTH[Database] .
    protecting PROOF-TRUTH[Key] .
    protecting PROOF-TRUTH[Item] .
    protecting PROOF-TRUTH[Int] .

    var D : Database .
    var K K1 : Key .
    var I I1 : Item .
    var NR : Int .      *** number of readers
    var NW : Int .      *** number of writers

*** Create a new database.
    eq  *new = rw(0,0,eod) .
    eq  new = rw(0,0,eod) .

*** Enter the database.
    eq  *rd(K,rw(NR,NW,D)) =
        eif NW == 0 then
```

Figure 15: The Readers/Writers Proof Object (1 of 3).

```
            rw(NR + 1,NW,*rd(K,D))
        else
            *rd(K,rw(NR,NW,D))
        fi .
    eq  *wr(K,I,rw(NR,NW,D)) =
        eif (NR =s= 0) and (NW =s= 0) then
            rw(NR,NW + 1,*wr(K,I,D))
        else
            *wr(K,I,rw(NR,NW,D))
        fi .
    eq  rd(K,rw(NR,NW,D)) =
        eif NW =s= 0 then
            rw(NR + 1,NW,rd(K,D))
        else
            rd(K,rw(NR,NW,D))
        fi .
    eq  wr(K,I,rw(NR,NW,D)) =
        eif (NR =s= 0) and (NW =s= 0) then
            rw(NR,NW + 1,wr(K,I,D))
        else
            wr(K,I,rw(NR,NW,D))
        fi .

*** Exit the database.
    eq  rw(NR,NW,*rdout(I,D)) = *rdout(I,rw(NR - 1,NW,D)) .
    eq  rw(NR,NW,*wrout(I,D)) = *wrout(I,rw(NR,NW - 1,D)) .
    eq  rw(NR,NW,rdout(I,D)) = rw(NR - 1,NW,D) .
    eq  rw(NR,NW,wrout(I,D)) = rw(NR,NW - 1,D) .

*** Return results.
    eq  outside(*rdout(I,D)) = I .
    eq  outside(*wrout(I,D)) = I .

*** Read traversal algorithm.
    eq  *rd(K,eod) = *rdout(notfound,eod) .
    eq  *rd(K,rec(K1,I1,D)) =
        eif K =s= K1 then
            *rdout(I1,rec(K1,I1,D))
        else
            rec(K1,I1,*rd(K,D))
        fi .
    eq  rd(K,eod) = rdout(notfound,eod) .
    eq  rd(K,rec(K1,I1,D)) =
        eif K =s= K1 then
            rdout(I1,rec(K1,I1,D))
        else
            rec(K1,I1,rd(K,D))
        fi .

*** Write traversal algorithm.
```

Figure 15: The Readers/Writers Proof Object (2 of 3).

```
    eq  *wr(K,I,eod) = *wrout(I,rec(K,I,eod)) .
    eq  *wr(K,I,rec(K1,I1,D)) =
        eif K =s= K1 then
            *wrout(I,rec(K,I,D))
        else
            rec(K1,I1,*wr(K,I,D))
        fi .
    eq  wr(K,I,eod) = wrout(I,rec(K,I,eod)) .
    eq  wr(K,I,rec(K1,I1,D)) =
        eif K =s= K1 then
            wrout(I,rec(K,I,D))
        else
            rec(K1,I1,wr(K,I,D))
        fi .

*** Passing equations.
    eq  rec(K1,I1,*rdout(I,D)) = *rdout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,*wrout(I,D)) = *wrout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,rdout(I,D)) = rdout(I,rec(K1,I1,D)) .
    eq  rec(K1,I1,wrout(I,D)) = wrout(I,rec(K1,I1,D)) .

    eq  *rd(K1,rdout(I,D)) = rdout(I,*rd(K1,D)) .
    eq  *rd(K1,wrout(I,D)) = wrout(I,*rd(K1,D)) .
    eq  *wr(K1,I1,rdout(I,D)) = rdout(I,*wr(K1,I1,D)) .
    eq  *wr(K1,I1,wrout(I,D)) = wrout(I,*wr(K1,I1,D)) .

    eq  rd(K1,*rdout(I,D)) = *rdout(I,rd(K1,D)) .
    eq  rd(K1,*wrout(I,D)) = *wrout(I,rd(K1,D)) .
    eq  rd(K1,rdout(I,D)) = rdout(I,rd(K1,D)) .
    eq  rd(K1,wrout(I,D)) = wrout(I,rd(K1,D)) .
    eq  wr(K1,I1,*rdout(I,D)) = *rdout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,*wrout(I,D)) = *wrout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,rdout(I,D)) = rdout(I,wr(K1,I1,D)) .
    eq  wr(K1,I1,wrout(I,D)) = wrout(I,wr(K1,I1,D)) .

endo
```

Figure 15: The Readers/Writers Proof Object (3 of 3).

```
object RW is
    using READERS-WRITERS[KEY-TO-QIDL,ITEM-TO-NAT] .

*** Proof tools.
    op  inv : Database -> Bool .
    using PROOF-INT .
    using PROOF-CASE * (sort Sort to Database, op p to inv) .

*** Readers/Writers invariant.
    var D : Database .
    var NR NW : Int .
    eq  inv(rw(NR,NW,D)) =
        ((NR >s= 0) and (NW >s= 0) and (NW <s= 1) and
        ((NR =s= 0) or (NW =s= 0))) =s= true .
endo
```

Figure 16: The Readers/Writers Invariant Property.

5.6.2 Construction of the Invariant

The next step in the verification is to translate Equation 1 on page 101 into an OBJ3 equation. This equation is part of the *predicate object* RW in Figure 16. As in Figure 10, RW is an instantiated database mapping identifiers to natural numbers. However, the source of the instantiation is the proof object from Figure 15. Operation inv is a predicate on sort Database; its equation defines it to be true iff the invariant is true. RW also imports PROOF-INT and PROOF-CASE. The signatures of these proof tools are shown in Figure 17; Appendix 5.10 lists both in their entirety. PROOF-INT provides symbolic relational operations on integers that are similar in purpose to _=s=_. Likewise, it contains some familiar and some special-purpose tautologies. PROOF-CASE provides a "case-analysis" rule [GHM78]. This is an example of a second-order rule; it is instantiated by sort and operation renaming.[12] This is one of OBJ3's module-expression constructs.

5.6.3 Induction Scheme

Having constructed the proof and predicate objects, terms whose reductions verify the invariant can now be formulated. The approach resembles structural induction, a generalization of induction over the integers [GHM78]. First, we prove that the invariant holds for a newly constructed database (the basis). Next, we assume properties of an existing database (an inductive hypothesis) and prove that an invocation trying to "enter" the database preserves the invariant (an inductive step). Finally, we assume properties of an existing database (another inductive hypothesis) and prove that an invocation trying to "exit" the database preserves the invariant (another inductive step). Both star and nonstar operations must be considered. This doubles the number of reductions in the proof, but half of them are essentially reruns. Before starting, however, one more proof tool is necessary.

Recall that the transformation from a conditional equation to a regular equation introduced tail recursion. Also, notice that some of the traversal equations from Figure 9, hence Figure 15, are recursive. In order to avoid infinite reduction sequences, this recursion must be controlled. The solution is to use a modified version of the interpreter that provides a user-callable LISP function recursion-limit, shown in Figure 18. Arguments are bound to parameters according to LISP's "by keyword" parameter-passing mechanism. Some defaults apply. If the :obj keyword/argument pair is omitted, all operations with the specified name are recursion limited. If the :op pair is

[12] Conventional parameterisation and instantiation did not seem to work here.

```
object PROOF-INT is
    protecting TRUTH-VALUE .
    protecting INT .
    protecting PROOF-TRUTH[INT] .

    op  _<s_ : Int Int -> Bool [prec 51] .
    op  _<s=_ : Int Int -> Bool [prec 51] .
    op  _>s_ : Int Int -> Bool [prec 51] .
    op  _>s=_ : Int Int -> Bool [prec 51] .
        .
        .
        .
endo

object PROOF-CASE is
    sort Sort .

    op  p : Sort -> Bool .
    op  eif_then_else_fi : Bool Sort Sort -> Sort .
    op  eif_then_else_fi : Bool Bool Bool -> Bool .

    var B : Bool .
    var S1 S2 : Sort .

    eq  p(eif B then S1 else S2 fi) =
        eif B then
            p(S1)
        else
            p(S2)
        fi .
endo
```

Figure 17: Signatures of PROOF-INT and PROOF-CASE.

```
(recursion-limit
  :obj <obj-name>     ;string
  :op <op-name>       ;string
  :limit <limit>)     ;integer
```

Figure 18: Recursion Limiting.

```
reduce inv(*new) .
reduce inv(new) .
```

(a): Reductions.

```
==========================================
reduce in RW : inv(*new)
rewrites: 12
result Bool: true
==========================================
reduce in RW : inv(new)
rewrites: 12
result Bool: true
```

(b): Results.

Figure 19: The Bases Proofs.

omitted, all operations in the specified object are recursion limited. If the :limit pair is omitted, the limit is removed.

The modified interpreter maintains a list of recursion-limited operation names. Just before each reduction step, the top operation of the subterm to be reduced is compared to those on the list. If a match is found, the current recursion depth for that operation is compared to its limit. If the limit has been reached, the subterm is marked as reduced; otherwise, the current recursion depth for that operation is incremented and subterm reduction proceeds.

Figure 19 shows two reductions proving that the invariant holds for a newly constructed database. No recursion limiting is necessary here. The equation for *new (or new) is applied, then the equation for inv. The rest of the reduction sequence is straightforward simplification — a process called *jittering*.

Figure 20 shows an inductive hypothesis and four reductions proving that the invariant is preserved by an invocation trying to enter the database. OBJ3's ev keyword causes the LISP expression following it to be evaluated. This and recursion-limit are used to avoid the recursion problems discussed earlier. For tidiness, the limits are removed when they are no longer needed. RW1 imports the predicate object RW from Figure 16 and adds the inductive-hypothesis equations. The assumption is that the current state of the database satisfies the invariant. In other words,

```
inv(rw(arvar,awvar,dvar))
```

is true. Now what we want to do is show that a *rd, rd, *wr, or wr invocation on the database preserves the invariant. The four terms to be reduced are constructed accordingly. In the reductions, the equation allowing the invocation to pass by rw is applied first, leaving a large eif_then_else_fi subterm surrounded by inv. Case analysis then distributes inv to the then and else subterms. The rest is jittering.

Notice that none of these terms reduce to true, as before. This is because a counter's current value may prevent an invocation from entering the database. In essence, the synchronization provided by conditional equations in a specification object translates to equational polling in a proof. These reduction results are interpreted as follows. If an invocation is allowed to enter the database, the invariant is preserved; but if it is not allowed to enter, it blocks until it can. Clearly, this is only partial correctness. However, techniques for proving *finite termination* (i.e., that an equation set leads to reductions that always terminate) are well known [Gog80].

Figure 21 shows two inductive hypotheses and four reductions proving that the invariant is preserved by an invocation trying to exit the database. This proof is simpler than the database-entry proof due to a general fact that a process has no reason to block when giving up a resource. Both inductive hypotheses import RW1, the inductive hypothesis from Figure 20. Thus, RW2 and RW3 represent additional facts that we assume are true when a reader or writer (resp.) is in the database. Assuming that a reader is in the database, there must be at least one reader and no

```
object RW1 is using RW .
    eq  arvar >s= 0 = true .
    eq  awvar >s= 0 = true .
    eq  awvar <s= 1 = true .
    eq  arvar =s= 0 or awvar =s= 0 = true .
endo

ev (recursion-limit :obj "READERS-WRITERS" :op "•rd" :limit 1)
ev (recursion-limit :obj "READERS-WRITERS" :op "rd" :limit 1)
reduce inv(•rd(kvar,rw(arvar,awvar,dvar))) .
reduce inv(rd(kvar,rw(arvar,awvar,dvar))) .
ev (recursion-limit :obj "READERS-WRITERS" :op "•rd")
ev (recursion-limit :obj "READERS-WRITERS" :op "rd")

ev (recursion-limit :obj "READERS-WRITERS" :op "•wr" :limit 1)
ev (recursion-limit :obj "READERS-WRITERS" :op "wr" :limit 1)
reduce inv(•wr(kvar,ivar,rw(arvar,awvar,dvar))) .
reduce inv(wr(kvar,ivar,rw(arvar,awvar,dvar))) .
ev (recursion-limit :obj "READERS-WRITERS" :op "•wr")
ev (recursion-limit :obj "READERS-WRITERS" :op "wr")
```

(a): Reductions.

```
reduce in RW1 : inv(•rd(kvar,rw(arvar,awvar,dvar)))
rewrites: 17
result Bool: eif awvar =s= 0 then true else inv(•rd(kvar,rw(arvar,awvar,
    dvar))) fi
```

```
reduce in RW1 : inv(rd(kvar,rw(arvar,awvar,dvar)))
rewrites: 17
result Bool: eif awvar =s= 0 then true else inv(rd(kvar,rw(arvar,awvar,
    dvar))) fi
```

```
reduce in RW1 : inv(•wr(kvar,ivar,rw(arvar,awvar,dvar)))
rewrites: 25
result Bool: eif awvar =s= 0 and arvar =s= 0 then true else inv(•wr(
    kvar,ivar,rw(arvar,awvar,dvar))) fi
```

```
reduce in RW1 : inv(wr(kvar,ivar,rw(arvar,awvar,dvar)))
rewrites: 25
result Bool: eif awvar =s= 0 and arvar =s= 0 then true else inv(wr(
    kvar,ivar,rw(arvar,awvar,dvar))) fi
```

(b): Results.

Figure 20: The Inductive Steps for Database Entry.

```
object RW2 is
    using RW1 .
    var NR NW : Int .
    eq  nrvar >s 0 = true .
    eq  nwvar =s= 0 = true .
    eq  •rdout(ivar,rw(NR,NW,dvar)) = rw(NR,NW,dvar) .
endo
reduce inv(rw(nrvar,nwvar,•rdout(ivar,dvar))) .
reduce inv(rw(nrvar,nwvar,rdout(ivar,dvar))) .

object RW3 is
    using RW1 .
    var NR NW : Int .
    eq  nrvar =s= 0 = true .
    eq  nwvar >s 0 = true .
    eq  nwvar <s= 1 = true .
    eq  •wrout(ivar,rw(NR,NW,dvar)) = rw(NR,NW,dvar) .
endo
reduce inv(rw(nrvar,nwvar,•wrout(ivar,dvar))) .
reduce inv(rw(nrvar,nwvar,wrout(ivar,dvar))) .
```

(a): Reductions.

```
reduce in RW2 : inv(rw(nrvar,nwvar,•rdout(ivar,dvar)))
rewrites: 15
result Bool: true
```

```
reduce in RW2 : inv(rw(nrvar,nwvar,rdout(ivar,dvar)))
rewrites: 14
result Bool: true
```

```
reduce in RW3 : inv(rw(nrvar,nwvar,•wrout(ivar,dvar)))
rewrites: 19
result Bool: true
```

```
reduce in RW3 : inv(rw(nrvar,nwvar,wrout(ivar,dvar)))
rewrites: 18
result Bool: true
```

(b): Results.

Figure 21: The Inductive Steps for Database Exit.

writers in the database. Assuming that a writer is in the database, there must be no readers and exactly one writer in the database. The last equation in each object discards an exiting star operation after it has updated the counters and left the database (nonstar operations are already discarded by specification equations). This allows the inv equation to be applied. In the reductions, the equation allowing the invocation to pass by rw is applied first, then the invocation is discarded and inv is expanded. The rest is jittering.

5.7 Explicitly Concurrent Specification

Admittedly, star operations specify concurrency in a subtle way. A more conventional approach, analogous to message passing, is now considered. Unfortunately, the approach relies on nontrivial modifications to OBJ3's interpreter. These changes define a new language, tentatively named OBJC, which is not yet implemented. This section describes the proposed modifications and demonstrates the approach on the Semaphore and Readers/Writers problems. Verification is not discussed.

We begin by recognizing that message passing can be simulated by other language constructs. Common examples that provide nonlocal transfer of data are input/output statements and global variables, but these are unavailable in OBJ3. In some sense, this is because input is a term to reduce, output is a reduced term, and only local variables can be referenced. In fact, OBJ3 enjoys a simple and elegant formal semantics partly because it prohibits global constructs; extending the language could risk these nice semantics. But:

> This world of systems programming, which may be strongly non-Church-Rosser as well as nonterminating, with its many flags and strategies, may seem like Alice's Wonderland to the theorist, and he might well long for a return to the simplicity of initial algebra semantics. However, the problems posed by the need to connect with the real world, including input/output and real-time programming, are not going to go away [GKM87].

5.7.1 Message-Passing Extensions

Nevertheless, the extensions are confined to a single parameterized object, MESSAGE, providing message-passing operations and sorts. Of course, the operations cannot be realized by equations, they must be treated specially by the interpreter. Externally, however, MESSAGE looks like a user-defined object.

Figure 22 shows the parameterization and signature of MESSAGE. Its sorts and operations allow disjoint subterms to exchange message terms. If these disjoint subterms are thought of as computations in distinct processes, MESSAGE operations model interprocess communication. The theme is simplicity. Low-level operations are provided so that high-level mechanisms can be constructed.

The built-in theory TRIV simply requires the parameter X to provide a sort Elt. Messages passed by an instance of MESSAGE are of this sort.

Several new sorts are provided by MESSAGE. Message insulates a user from the nonlocal behavior of message-passing operations. All MESSAGE operations have this coarity. QIDL provides alphanumeric identifiers that are used as symbolic addresses to establish the connection between communicating subterms. MessageId is just Nat. It is used to maintain the connection between communicating subterms for the duration of a transaction. More specifically, when a message is first sent to a symbolic address, it is automatically assigned a unique identification number. A subsequent reply must refer to the original message by its number.

```
object MESSAGE [X :: TRIV] is
    sort Message .
    sort MessageNum .
    protecting QIDL .
    protecting NAT .
    subsort Elt < Message .
    subsort MessageNum < Nat .

    op send : Id Elt -> Message .
    op sending : Id Elt -> Message .
    op sent : MessageNum -> Message .
    op wait : MessageNum -> Message .
    op waiting : MessageNum -> Message .
    op waited : MessageNum Elt -> Message .
    op receive : Id -> Message .
    op receiving : Id -> Message .
    op received : MessageNum Elt -> Message .
    op reply : MessageNum Elt -> Message .
    op replying : MessageNum Elt -> Message .
    op replied : MessageNum -> Message .
    :
    :
endo
```

Figure 22: Signature of MESSAGE.

Notice that parameterization requires the operations — hence messages — to be strongly sorted. This is a tradeoff. Message passing is more convenient with polymorphic operations, but polymorphism is more difficult to reconcile with OBJ3's existing semantics.

When discussing a transaction, we shall call the subterm initiating the exchange the "sender"; the other participating subterm is the "receiver". Although this is superficially obvious, notice that the receiver sends, and the sender receives, the reply.

5.7.2 Message-Passing Protocol

MESSAGE allows subterms to pass messages in a variety of ways. To illustrate how its operations are used, however, a simple synchronous exchange is described here. A sender initiates the exchange by constructing a message, addressing it with an identifier, and sending it to a receiver. When a suitable receiver is located, a message number is generated and the message is delivered. The receiver then processes the message while the sender waits. After a reply is constructed, the receiver completes the exchange by using the message number to reply to the sender. Figure 23 pictorially describes this interaction and suggests alternatives. We now examine this simple exchange in more detail.

A sender sends by applying an equation that introduces a send operation into its subterm. A send takes two arguments: an entry identifier e and a message term m. If, at that time, there exists a subterm receiving(e), message m is sent. Otherwise, send(e,m) reduces to sending(e,m), thus informing the sender that there is currently no receiver.

A receiver receives by applying an equation that introduces a receive operation into its subterm. A receive takes one argument: an entry identifier e. If, at that time, there exists a subterm sending(e,m), message m is sent. Otherwise, receive(e) reduces to receiving(e), thus informing the receiver that there is currently no sender.

The protocol is designed to avoid temporal assumptions. In particular, a send can appear before or after its corresponding receive. The first operation to appear blocks; the second locates the first. In addition, the protocol is nondeterministic. If multiple matching senders or receivers

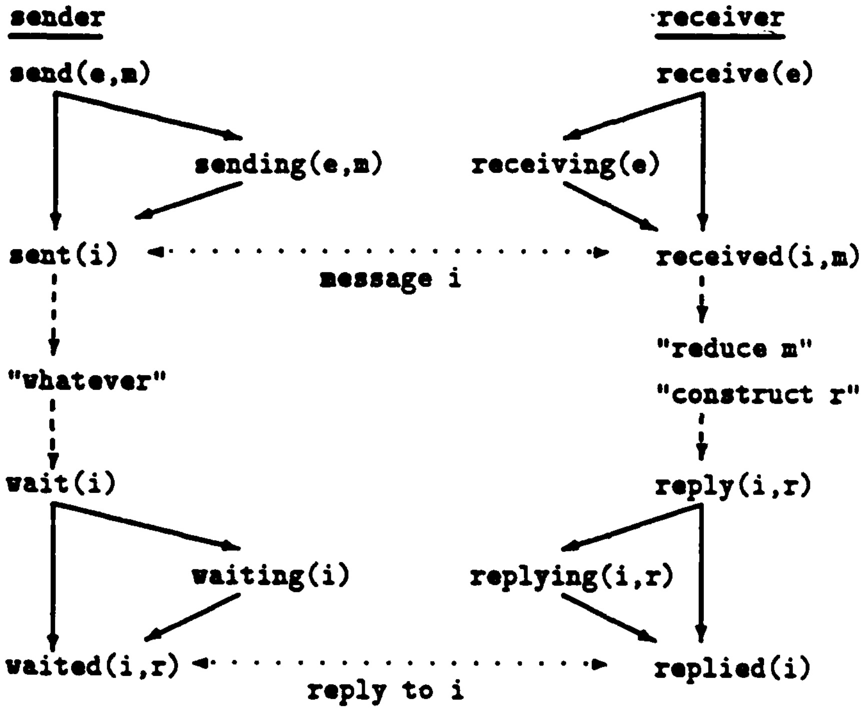

Figure 23: Message-Passing Protocol.

appear, any one might be selected. These considerations apply to the wait and reply operations as well. In fact, the wait/reply subprotocol is virtually identical to the send/receive subprotocol.

Suppose a send(e,m) is matched with some receiving(e). The case where the sender is blocked is analogous. Message m is sent from sender to receiver by assigning a message number i to m, reducing send(e,m) to sent(i), and reducing receiving(e) to received(i,m). These steps are performed without any intervening reductions (i.e., the sequence is atomic).

After the message has been passed, regular reduction resumes. Since the sender needs to receive a reply, it retains the message number i and eventually applies an equation that introduces wait(i) into its subterm. Since the receiver needs to send a reply, it retains i, constructs a reply r, and eventually introduces reply(i,r) into its subterm.

If a subterm replying(i,r) exists when wait(i) is introduced into the sender, the reply r is sent. Otherwise, wait(i) reduces to waiting(i), thus informing the sender that the receiver is not currently replying.

If a subterm waiting(i) exists when reply(i,r) is introduced into the receiver, the reply r is sent. Otherwise, reply(i,r) reduces to replying(i,r), thus informing the receiver that the sender is not currently waiting.

Suppose a wait(i) is matched with some replying(i,r). As before, the case where the sender is blocked is analogous. Reply r is sent from receiver to sender by reducing wait(i) to waited(i,r) and replying(i,r) to replied(i). Again, these steps are performed without any intervening reductions.

Close inspection reveals that the argument i of both waited and replied is unnecessary; the message number can be retained simply by copying it. This brings up an interesting point: as long as both sender and receiver have a copy of the message number, a bidirectional communication channel exists between them. Unfortunately, the channel is half-duplex; the sender and receiver must alternate in their roles as waiter and replier. As with UNIX pipes [Bac86], however, two half-duplex channels can simulate a full-duplex channel. For example, the second message sent from sender to receiver in half-duplex "mode" can be a unique entry identifier constructed from the existing channel's message number. The sender and receiver can then use this new identifier to establish another half-duplex channel in the opposite direction. Of course, they must agree on who sends on which channel.

The protocol in Figure 23 permits both synchronous and asynchronous communication. A synchronous message is one where the sender waits for a reply and the receiver sends a reply. An asynchronous message is one where the sender does not wait for a reply and the receiver does not send a reply. Hybrid methods cause problems. For example, if a sender behaves asynchronously and its receiver behaves synchronously, replying never reduces to replied and the receiver blocks forever. At first, retry counting seems like a solution to this problem. That is, the receiver can repeatedly reduce replying to reply and count the number of times reply is reduced to a replying. Unfortunately, those are probably the only reductions occurring; there is no way to tell what the sender is up to. Fortunately, ignorable replies are not terribly important; a sender can always communicate its desire for a reply as part of the message, thus instructing the receiver to behave accordingly.

Notice that a message could contain MESSAGE operations. Although the idea of a recursive "chain letter" is intriguing, we avoid defining its meaning, at least until a reasonable use for one is discovered. The consequences of such a message are boggling.

MESSAGE provides twelve operations. Earlier, this object was claimed to be simple; arguably, that is not true. But if we try to reduce the number of operations, the object becomes more

```
outside(server(···),clients(
    client(···),
    client(···),
    ⋮
    client(···)))
```

Figure 24: A Client/Server Term with a Static Number of Clients.

```
outside(receive('outside),server(···),clients(
    client(···),
    client(···),
    ⋮
    client(···)))
```

Figure 25: A Client/Server Term with a Message Catcher.

difficult to use. Perhaps the greatest temptation is to eliminate sent and replied. Instead of reducing to these monadic operations, we could simply reduce to their argument. Unfortunately, this complicates the detection of a successful send or receive (resp.). Likewise, and because they bundle together their two arguments, waited and received are necessary. Another temptation is to eliminate sending, waiting, receiving, and replying. Instead of reducing to these blocked operations after a first attempt to communicate, we could wait until both the sender and receiver are ready before reducing in either. But these operations are convenient, both for specification and interpretation. A specification can use them to terminate a blocked communication attempt. An interpreter can use them to record the state of partially completed communications; no internal list of blocked operations is needed. Clearly, neither send nor receive can be eliminated. Likewise, synchronous communication requires wait and reply. Finally, consider the "geometry" of the protocol. Message-passing solutions are often unbalanced, yet these twelve operations exhibit a great deal of symmetry. A simpler object would lack this important attribute.

5.7.3 Centralized Solutions

A concurrent programming problem can generally be solved in two ways. A centralized solution is characterized by a single *server* process that coordinates the execution of multiple *client* processes. In a decentralized solution, the responsibility of global synchronization is shared among all processes. OBJC is general enough to specify both centralized and decentralized solutions, but we shall concentrate on centralized solutions.

A client/server solution can be implemented by a *client object* and a *server object*. A term to reduce is then constructed from a server subterm and multiple client subterms. Equations in the server object cause reductions in the server subterm; client-object equations reduce elsewhere. Figure 24 shows a template for a client/server term with a static number of clients. Operation outside operates as before: it bundles its arguments and provides context. Unfortunately, reducing such a term to a result is also as awkward as before. Figure 25 shows a template that uses message passing to solve the problem. With this term structure, the server or any client can send a message to the 'outside entry. An equation like

```
        outside(received(X,M),S,C) = M
```

can then be used to terminate a reduction and return the message M as its result. This, of course, is reminiscent of LISP's catch/throw mechanism. Alternately, outside can be thought of as a "termination server". If the number of clients is not static, a dyadic version of clients can be

```
outside(receive('outside),server(···),
    clients(client(···),
        clients(client(···),
            ⋮
            clients(client(···),client(···))···)
```

Figure 26: A Client/Server Term for an Arbitrary Number of Clients.

used to maintain a list of clients. Figure 26 shows the even LISP-ier template.

A server subterm comprises several components. At minimum, it must contain the message-passing entries referenced by its clients. Synchronization and scheduling information is probably also stored there, along with application-dependent data.

MESSAGE guarantees that message-passing reductions at the server interface are mutually exclusive. Conditional synchronization and mutually exclusive access to application-dependent structures is enforced by server equations that maintain counters and message queues within the server subterm. Mutually exclusive access to the counters and queues is guaranteed by the atomicity of equation application and the locality of server state. Scheduling is accomplished by reordering these queues. Notice that MESSAGE exhibits FCFS behavior; a FCFS scheduling discipline may not require explicit queues.

A server may service more than one type of request (e.g., allocate requests and deallocate requests). Such a server may receive all requests through a single entry or it may have several entries that each receive requests of a particular type. Requests are segregated by using different entry identifiers. A single-entry server should accept requests promptly, whether or not they can be serviced immediately. These requests should then be queued for scheduling (based on content) and serviced when appropriate. A multi-entry server that implements a FCFS discipline probably does not need scheduling queues.

A client subterm is usually simpler than the server subterm, at least with regard to message passing. It requests service with a send and obtains results with a wait.

5.7.4 Examples

Figure 27 is an OBJC server object that models the behavior of a FCFS semaphore. A companion client object is shown in Figure 28. These two objects are somewhat analogous to Figure 7, where star operations are used.

The server object in Figure 27 begins by instantiating MESSAGE for integer messages. As before, operations must return something, so p and v return the semaphore's value. Operation new takes an integer argument and returns a semaphore whose counter is initialized to the argument's value. A semaphore is bundled by a sem operation, which plays the role of server in Figure 25. The first two arguments of sem are the 'p and 'v message identifiers (resp.) — the server entries. The last argument of sem is the semaphore's counter. Since a semaphore is a FCFS structure, and since the multi-entry approach is used here, explicit queues are unnecessary. Notice that a blocked reply at one entry does not prevent requests at the other entry from being serviced.

The client object in Figure 28 imports the server object, but only uses its operations indirectly, via messages. It also instantiates MESSAGE again, this time for reduction-termination purposes, as in Figure 25. The client's p and v operations are equationally translated into server requests. Since the server provides multiple entries, empty messages are sufficient, but OBJC requires that something be sent. In CSP, an empty messages is called a *signal* [Hoa78]. Operations user and

```
object SEMAPHORE is sort Semaphore .
    protecting INT + MESSAGE[INT] .

    op  new : Int -> Semaphore .
    op  sem : Message Message Int -> Semaphore .
    ops p v : Int -> Int .

    var I : Int .
    var M1 M2 : Message .
    var N : MessageNum .

*** Create an initialized semaphore.
    eq  new(I) = sem(receive('p),receive('v),I) .

*** Service requests.
    cq  sem(received(N,M1),M2,I) = sem(reply(N,p(I)),M2,p(I)) if I > 0 .
    eq  sem(M1,received(N,M2),I) = sem(M1,reply(N,v(I)),v(I)) .

*** Get ready for more requests.
    eq  sem(replied(N),M2,I) = sem(receive('p),M2,I) .
    eq  sem(M1,replied(N),I) = sem(M1,receive('v),I) .

    eq  p(I) = I - 1 .
    eq  v(I) = I + 1 .

endo
```

Figure 27: A Semaphore Server Object.

```
object USER is sort User .
    protecting SEMAPHORE .
    protec:ing MESSAGE[INT] .

    ops p v : -> Message .
    op  user : Int Message Message -> User .
    op  done : Message -> User .
    op  outside : Message Semaphore User User -> Message .

    var M M1 M2 : Message .
    var N N1 N2 : MessageNum .
    var S : Semaphore .
    var U1 U2 : User .
    var I : Int .

*** Request semaphore service.
    eq  p = send('p,0) .
    eq  v = send('v,0) .
    eq  sent(N) = wait(N) .

*** Reduce to the number of the user that finished first.
    eq  user(I,waited(N1,M1),waited(N2,M2))) = done(send('outside,I)) .
    eq  outside(received(N,M),S,U1,U2) = M .

endo
```

Figure 28: A Semaphore Client Object.

done play the role of clients in Figure 25. Operation user bundles a user's two requests together with a "user number". After both of a user's requests have been serviced, its user number is sent to 'outside. The last equation reduces a term to the user number of the first user to finish. As an example,

```
        outside(receive('outside),new(1),user(1,p,v),user(2,p,v))
```

reduces to either 1 or 2.

Notice the awkwardness involved in ensuring that the send to 'outside is issued only after a user's requests have been serviced. When message passing is used, evaluation strategies cannot be relied upon to provide sequencing.

The next example is an OBJC solution to the Readers/Writers problem. Again, a client/server structure is used to provide mutual exclusion and conditional synchronization. Figure 29 is the server object. The client object is shown in Figure 30. These two objects should be compared to Figure 9.

The server object in Figure 29 is parameterized; as before, it can be instantiated with various KEY and ITEM objects. Since messages are database items, MESSAGE is instantiated with ITEM. Operation rw plays the role of server in Figure 25. However, it behaves quite differently than the rw of Figure 9. Specifically, wr still modifies the Database argument of rw in place, but rd initiates a read traversal on a copy of the database, which is put on a list of active read traversals, which is the last argument of rw. This technique simplifies the read traversal algorithm. Operation new creates an initialized database; the two conditional equations succinctly provide conditional synchronization for it. A database contains only one entry, 'rw. Therefore, a request message must specify whether a read or write is desired. This is done by passing a rd or wr subterm, which is translated into a get or put. Operation gets is used like LISP's cons to construct the list of active read traversals. Notice that even though rw has only one entry, the FCFS discipline avoids an explicit queue.

The client object in Figure 30 instantiates a Readers/Writers database that maps identifiers to the natural numbers, as before. The client's read and write operations reduce to rd and wr server requests (resp.). The user, done, and outside operations behave almost as before. A user bundles two requests together. After these are serviced, the user reduces to done. After both users are done, a term reduces to its database argument. For example,

```
        outside(new,
            user(write('fred,100),read('fred)),
            user(write('fred,200),write('fred,300)))
```

reduces to

```
        rw(receive('rw),0,0,rec('fred,x,eed),eer)
```

where x is either 100, 200, or 300. Notice that the initially intuitive

```
        outside(new,
            user(write('fred,100),write('fred,read('fred) + 1)),
            user(write('fred,200),write('fred,read('fred) + 1)))
```

is not only nondeterministic, but it contains a chain letter, as discussed previously. In order to achieve the intended effect, explicit sequencing must be used. The trick is to put the read and write operations on a list and only reduce one to a send when it is at the head of the list. After the send is serviced, it could be removed from the front of the list to allow the next operation to be processed.

5.8 Verification of Implementations

The verification techniques of Section 5.8 are useful for proving properties of a particular specification or solution. In this section, we investigate how OBJ3 can assist in the top-down development

```
object READERS-WRITERS[Key :: KEY, Item :: ITEM] is
    sorts Readers-Writers Database .
    protecting INT + MESSAGE[ITEM] .

    op  new : -> Readers-Writers .
    op  rw : Message Int Int Database Message -> Readers-Writers .
    op  rd : Key -> Item .
    op  wr : Key Item -> Item .
    op  get : MessageNum Key Database -> Message .
    op  put : MessageNum Key Item Database -> Database .
    op  rec : Key Item Database -> Database .
    op  eod : -> Database .
    op  eor : -> Message .
    op  gets : Message Message -> Message .
    op  putreply : Message Database -> Database .
    op  putdone : Database -> Database .

    var M R : Message .
    var N : MessageNum .
    var D : Database .
    var K K1 : Key .
    var I I1 : Item .
    var NR : Int .      *** number of readers
    var NW : Int .      *** number of writers

*** Create an initialized database.
    eq  new = rw(receive('rw),0,0,eod,eor) .

*** Service requests.
    cq  rw(received(N,rd(K)),NR,NW,D,R) =
        rw(receive('rw),NR + 1,NW,D,gets(get(N,K,D),R))
        if NW == 0 .
    cq  rw(received(N,wr(K,I)),NR,NW,D,R) =
        rw(receive('rw),NR,NW + 1,put(N,K,I,D),R)
        if (NR == 0) and (NW == 0) .

*** Purge serviced requests.
    eq  rw(M,NR,NW,D,gets(replied(N),R)) = rw(M,NR - 1,NW,D,R) .
    eq  rw(M,NR,NW,putdone(D),R) = rw(M,NR,NW - 1,D,R) .

*** Read traversal algorithm.
    eq  get(N,K,eod) = reply(N,notfound) .
    eq  get(N,K,rec(K1,I,D)) =
        if K == K1 then
            reply(N,I)
        else
            get(N,K,D)
        fi .

*** Write traversal algorithm.
```

Figure 29: A Readers/Writers Server Object (1 of 2).

```
    eq  put(N,K,I,eod) = putreply(reply(N,I),rec(K,I,eod)) .
    eq  put(N,K,I,rec(K1,I1,D)) =
        if K == K1 then
            putreply(reply(N,I),rec(K,I,D))
        else
            rec(K1,I1,put(N,K,I,D))
        fi .
    eq  putreply(replied(N),D) = putdone(D) .
    eq  rec(K1,I1,putdone(D)) = putdone(rec(K1,I1,D)) .

endo
```

Figure 29: A Readers/Writers Server Object (2 of 2).

```
object USER is sort User .
    protecting QIDL + NAT + READERS-WRITERS[QIDL,NAT] .

    op  read : Key -> Message .
    op  write : Key Item -> Message .
    op  user : Message Message -> User .
    op  done : -> User
    op  outside : Readers-Writers User User -> Readers-Writers .

    var M1 M2 : Message .
    var N N1 N2 : MessageNum .
    var RW : Readers-Writers .
    var K : Key .
    var I : Item .

*** Request database service.
    eq  read(K) = send('rw,rd(K)) .
    eq  write(K,I) = send('rw,wr(K,I)) .
    eq  sent(N) = wait(N) .

*** Reduce to the final database value.
    eq  user(waited(N1,M1),waited(N2,M2)) = done .
    eq  outside(RW,done,done) = RW .

endo
```

Figure 30: A Readers/Writers Client Object.

of a solution.

An early step in developing an object is to construct a *specification object*. A specification object implements its operations autonomously; it does not employ operations from other user-defined objects.[13] This is also known as *direct implementation*. A specification object is more than just a "thought tool". Its executability allows a specification, and the implementations that import it, to be tested. More importantly, properties of the specification, and the correctness of the implementations that import it, can be proven. "Thus a true top-down implementation methodology can be achieved" [GHM78].

A later step in developing an object is to construct an *implementation object*. An implementation object implements its operations via operations from other user-defined objects. Typically, this is done to reduce time or space requirements. A specification object may have multiple implementation objects.

Verifying such a development means proving that the implementation object correctly implements the operations specified by the specification object. This is done by proving that for each equation in the specification object, the equation holds when its operations are replaced by their implementations from the implementation object. This, in turn, can be done by showing that each specification equation reduces to `true`, when reduced in the context of the implementation object.

As before, however, complications arise when OBJ3 is used as a theorem prover. In particular, the proof tools described in Section 5.8 are also needed here.[14] The proof technique is demonstrated by verifying the correctness of an implementation step in the development of a symbol table maintenance program. Both the methodology and sample problem are taken from a seminal paper [GHM78]. Aside from using OBJ3, the remainder of this section tries to follow the original work as closely as possible.

[13] An object can hardly avoid using built-in operations.

[14] Actually, this verification was done before the Readers/Writers verification — as a warmup.

```
theory IDENTIFIER is
    sort Identifier .
endth

theory ATTRIBUTE is
    sort Attribute .
    op undefined : -> Attribute .
endth
```

Figure 31: The Requirements for Symbol Table Parameters.

5.8.1 The Symbol Table Problem

A good example of a problem whose solution (i.e., whose implementation) must be both time and space efficient is the symbol table for a compiler of a block-structured programming language, the example for which was introduced in section 3.11 of the report. This section considers further details of the OBJ3 implmentaion and verfication of the problem.

The problem is to maintain a dynamic mapping from program identifiers to their attributes. This mapping changes, according to the rules of static scoping, when the parser: enters a program block, encounters a variable declaration, and exits a program block. A symbol table provides the following operations.

- init: Allocate and initialize the symbol table for the outermost scope.
- enterblock: Prepare a new local naming scope.
- addid: Add an identifier and its attributes to the symbol table.
- leaveblock: Discard entries from the most current scope and reestablish the next outer scope. If already in the outermost scope, do nothing.
- isinblock: Has a specified identifier already been declared in this scope? (Used to check for duplicate declarations.)
- retrieve: Return the attributes associated with the most local definition of a specified identifier.

5.8.2 Specification of the Symbol Table

For a symbol table to map abstract identifiers to abstract attributes, its object must be parameterised. The parameters are required to satisfy the IDENTIFIER and ATTRIBUTE theories in Figure 31.

Specification of the required operations is straightforward; the specification object is shown in Figure 32. In order to test the specification, it must be instantiated. Figure 33 instantiates a SYMBOLTABLE that maps strings to the natural numbers; some example reductions are also shown.

5.8.3 Implementation of the Symbol Table

A symbol table is implemented as a stack of mappings. As promised, the STACK and MAPPING objects do not need to be implemented, the direct implementations provided by their specification objects are sufficient. Both objects are parameterized. STACK takes one parameter, characterized by the ELEMENT theory in Figure 34. MAPPING takes two parameters, characterized by the DOMAIN and

```
object SYMBOLTABLE[I :: IDENTIFIER, A :: ATTRIBUTE] is
    sort Symboltable .

    op  init : -> Symboltable .
    op  enterblock : Symboltable -> Symboltable .
    op  addid : Symboltable Identifier Attribute -> Symboltable .
    op  leaveblock : Symboltable -> Symboltable .
    op  isinblock : Symboltable Identifier -> Bool .
    op  retrieve : Symboltable Identifier -> Attribute .

    var S : Symboltable .
    var I I1 : Identifier .
    var A : Attribute .

    eq  leaveblock(init) = init .
    eq  leaveblock(enterblock(S)) = S .
    eq  leaveblock(addid(S,I,A)) = leaveblock(S) .

    eq  isinblock(init,I) = false .
    eq  isinblock(enterblock(S),I) = false .
    eq  isinblock(addid(S,I,A),I1) =
        if I == I1 then
            true
        else
            isinblock(S,I1)
        fi .

    eq  retrieve(init,I) = undefined .
    eq  retrieve(enterblock(S),I) = retrieve(S,I) .
    eq  retrieve(addid(S,I,A),I1) =
        if I == I1 then
            A
        else
            retrieve(S,I1)
        fi .

endo
```

Figure 32: The Symbol Table Specification Object.

```
view IDENTIFIER-TO-QID from IDENTIFIER to QID is
  sort Identifier to Id .
endv

view ATTRIBUTE-TO-NAT from ATTRIBUTE to NAT is
    sort Attribute to Nat .
    op undefined to 0 .
endv

make ST is SYMBOLTABLE[IDENTIFIER-TO-QID,ATTRIBUTE-TO-NAT] .
```

(a): Instantiation.

```
==========================================
reduce in ST : retrieve(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bud)
rewrites: 7
result NzNat: 3
==========================================
reduce in ST : retrieve(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bill)
rewrites: 8
result Zero: 0
==========================================
reduce in ST : isinblock(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bud)
rewrites: 1
result Bool: false
==========================================
reduce in ST : isinblock(leaveblock(enterblock(addid(addid(init,'bud,
    3),'ed,6))),'bud)
rewrites: 7
result Bool: true
==========================================
reduce in ST : isinblock(leaveblock(enterblock(addid(addid(init,'bud,
    3),'ed,6))),'bill)
rewrites: 8
result Bool: false
```

(b): Reductions.

Figure 33: Instantiating and Testing a Symbol Table Specification.

```
theory ELEMENT is
    sort Element .
    op undefined : -> Element .
endth
```

Figure 34: The Requirements for Stack Parameters.

```
theory DOMAIN is
    sort Domain .
endth

theory RANGE is
    sort Range .
    op undefined : -> Range .
endth
```

Figure 35: The Requirements for Mapping Parameters.

```
object STACK[E :: ELEMENT] is
   sort Stack .

   op  newstack : -> Stack .
   op  push : Stack Element -> Stack .
   op  pop : Stack -> Stack .
   op  top : Stack -> Element .
   op  isnew : Stack -> Bool .
   op  replace : Stack Element -> Stack .

   var S : Stack .
   var E : Element .

   eq  pop(newstack) = newstack .
   eq  pop(push(S,E)) = S .

   eq  top(newstack) = undefined .
   eq  top(push(S,E)) = E .

   eq  isnew(newstack) = true .
   eq  isnew(push(S,E)) = false .

   eq  replace(S,E) = push(pop(S),E) .

endo
```

Figure 36: The Stack Specification Object.

RANGE theories in Figure 35. The STACK and MAPPING specification objects are shown in Figure 36 and Figure 37 (resp.).

A SYMBOLTABLE implementation object is shown in Figure 38. There are only two "highlights". First, notice the overwhelming ugliness of the importation of a stack of mappings. As the comment indicates, parameterized views would be a big improvement.[15] Second, consider the new operation symt. This is a *representation operation*. Like a type-breaking function in MODULA2, a representation operation allows something of one sort to be treated as if it is of another sort, but no conversion or coercion takes place. For example, in the equation

```
init = symt(push(newstack,newmap))
```

from Figure 38, symt is used to treat a nearly new stack as an empty symbol table. Subsorting could obviate representation operations in many cases, but circular subsort relations might arise in others.

Since the implementation object has not yet been verified, it should be tested. Figure 39 is analogous to Figure 33, it instantiates a SYMBOLTABLE mapping strings to numbers and shows some example reductions. Clearly, the rewrite count is an inappropriate measure of an implementation's efficiency.

5.8.4 Verification of the Symbol-Table Implementation

As before, there are several important steps in the verification. First, a *proof object* is constructed from the implementation object in Figure 38. This proof object is then instantiated and augmented with any necessary lemmas.[16] Equations from the specification object are then reduced in the context of this augmented object. Finally, of course, the lemmas are proven.

Although a correct implementation must implement every operation in a specification, we consider only a single equation from the specification object. This equation is an interesting one;

[15] There are hints that parameterised views might be supported in a later version of the interpreter [GW88].

[16] Automatic theorem proving is an iterative exercise; each unsuccessful iteration suggests new lemmas.

```
object MAPPING[D :: DOMAIN, R :: RANGE] is
    sort Mapping .

    op  newmap : -> Mapping .
    op  defmap : Mapping Domain Range -> Mapping .
    op  evmap : Mapping Domain -> Range .
    op  isdefined : Mapping Domain -> Bool .

    var M : Mapping .
    var D D1 : Domain .
    var R : Range .

    eq  evmap(newmap,D) = undefined .
    eq  evmap(defmap(M,D,R),D1) =
        if D == D1 then
            R
        else
            evmap(M,D1)
        fi .

    eq  isdefined(newmap,D1) = false .
    eq  isdefined(defmap(M,D,R),D1) =
        if D == D1 then
            true
        else
            isdefined(M,D1)
        fi .

endo
```

Figure 37: The Mapping Specification Object.

in particular, it is recursive and its proof requires a lemma.

A proof object is constructed exactly as before. Conditional equations are transformed into regular equations, variable operations replace variables, the symbolic equality operation replaces `_==_`, and the eager conditional operation replaces `if_then_else_fi`. Recall that these proof tools are provided by PROOF-TRUTH. Figure 40 shows the proof object corresponding to the implementation object in Figure 38.

Unfortunately, implementation via importation causes a subtle problem. It is not difficult to solve, just inconvenient. Namely, a specification object imported by an implementation object must also be transformed into a proof object; the proof object is imported instead. Transformation stops at this second level because — by definition — a specification object does not import other user-defined objects. The STACK and MAPPING proof objects are not shown here.

The next step is to construct a verification condition. We want to verify that the equation

```
retrieve(addid(S,I,A),I1) =
if I == I1 then
    A
else
    retrieve(S,I1)
fi
```

from Figure 32 holds in the context of Figure 40. The = is a metasymbol, not a built-in operation. In order to make the equation reducible, it is replaced by =s=. Variable operations, the representation operation, and PROOF-TRUTH operations transform the equation into the verification condition shown in Figure 41.

Object retrieve in Figure 41 instantiates the proof object from Figure 40 and augments it with a lemma that was discovered to be necessary in an earlier proof attempt. Following that, recursion is limited to avoid an infinite reduction and the verification condition is reduced. Examination of the detailed OBJ3 reduction trace demonstrates that the proof proceeds essentially as it does in

```
object SYMBOLTABLE[I :: IDENTIFIER, A :: ATTRIBUTE] is
    sort Symboltable .

    op  init : -> Symboltable .
    op  enterblock : Symboltable -> Symboltable .
    op  addid : Symboltable Identifier Attribute -> Symboltable .
    op  leaveblock : Symboltable -> Symboltable .
    op  isinblock : Symboltable Identifier -> Bool .
    op  retrieve : Symboltable Identifier -> Attribute .

*** protecting STACK[
***     ELEMENT-TO-MAPPING[DOMAIN-TO-IDENTIFIER,RANGE-TO-ATTRIBUTE]] .

    protecting STACK
        [view to MAPPING
            [view to I is
                sort Domain to Identifier .
            endv,
            view to A is
                sort Range to Attribute .
                op undefined to undefined .
            endv]
        is
            sort Element to Mapping .
            op undefined to newmap .
        endv] .

    op  symt : Stack -> Symboltable .

    var S : Stack .
    var I : Identifier .
    var A : Attribute .

    eq  init = symt(push(newstack,newmap)) .

    eq  enterblock(symt(S)) = symt(push(S,newmap)) .

    eq  addid(symt(S),I,A) = symt(replace(S,defmap(top(S),I,A))) .

    eq  leaveblock(symt(S)) =
        if isnew(pop(S)) then
            symt(replace(S,newmap))
        else
            symt(pop(S))
        fi .

    eq  isinblock(symt(S),I) = isdefined(top(S),I) .

    eq  retrieve(symt(S),I) =
        if isnew(S) then
```

Figure 38: The Symbol Table Implementation Object (1 of 2).

```
            undefined
        else
            if isdefined(top(S),I) then
                evmap(top(S),I)
            else
                retrieve(symt(pop(S)),I)
            fi
        fi .

endo
```

Figure 38: The Symbol Table Implementation Object (2 of 2).

```
view IDENTIFIER-TO-QID from IDENTIFIER to QID is
  sort Identifier to Id .
endv

view ATTRIBUTE-TO-NAT from ATTRIBUTE to NAT is
    sort Attribute to Nat .
    op undefined to 0 .
endv

make ST is SYMBOLTABLE[IDENTIFIER-TO-QID,ATTRIBUTE-TO-NAT] endm
```

(a): Instantiation.

```
------------------------------------------------------------
reduce in ST : retrieve(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bud)
rewrites: 35
result NzNat: 3
------------------------------------------------------------
reduce in ST : retrieve(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bill)
rewrites: 33
result Zero: 0
------------------------------------------------------------
reduce in ST : isinblock(enterblock(addid(addid(init,'bud,3),'ed,6)),
    'bud)
rewrites: 13
result Bool: false
------------------------------------------------------------
reduce in ST : isinblock(leaveblock(enterblock(addid(addid(init,'bud,
    3),'ed,6))),'bud)
rewrites: 23
result Bool: true
------------------------------------------------------------
reduce in ST : isinblock(leaveblock(enterblock(addid(addid(init,'bud,
    3),'ed,6))),'bill)
rewrites: 24
result Bool: false
```

(b): Reductions.

Figure 39: Instantiating and Testing a Symbol Table Implementation.

```
object SYMBOLTABLE[I :: IDENTIFIER, A :: ATTRIBUTE] is
    sort Symboltable .

    op  init : -> Symboltable .
    op  enterblock : Symboltable -> Symboltable .
    op  addid : Symboltable Identifier Attribute -> Symboltable .
    op  leaveblock : Symboltable -> Symboltable .
    op  isinblock : Symboltable Identifier -> Bool .
    op  retrieve : Symboltable Identifier -> Attribute .

*** protecting STACK[
***     ELEMENT-TO-MAPPING[DOMAIN-TO-IDENTIFIER,RANGE-TO-ATTRIBUTE]] .

    protecting STACK
        [view to MAPPING
            [view to I is
                sort Domain to Identifier .
            endv,
            view to A is
                sort Range to Attribute .
                op undefined to undefined .
            endv]
        is
            sort Element to Mapping .
            op undefined to newmap .
        endv] .

    op  symt : Stack -> Symboltable .

    protecting NAT .

    op  svar : Nat -> Stack .
    op  ivar : Nat -> Identifier .
    op  avar : Nat -> Attribute .

    protecting PROOF-TRUTH[Symboltable] .
    protecting PROOF-TRUTH[Stack] .
    protecting PROOF-TRUTH[Identifier] .
    protecting PROOF-TRUTH[Attribute] .

    var S : Stack .
    var I : Identifier .
    var A : Attribute .

    eq  init = symt(push(newstack,newmap)) .

    eq  enterblock(symt(S)) = symt(push(S,newmap)) .

    eq  addid(symt(S),I,A) = symt(replace(S,defmap(top(S),I,A))) .
```

Figure 40: The Symbol Table Proof Object (1 of 2).

```
    eq  leaveblock(symt(S)) =
        eif isnew(pop(S)) then
            symt(replace(S,newmap))
        else
            symt(pop(S))
        fi .

    eq  isinblock(symt(S),I) = isdefined(top(S),I) .

    eq  retrieve(symt(S),I) =
        eif isnew(S) then
            undefined
        else
            eif isdefined(top(S),I) then
                evmap(top(S),I)
            else
                retrieve(symt(pop(S)),I)
            fi
        fi .

endo
```

Figure 40: The Symbol Table Proof Object (2 of 2).

```
object RETRIEVE is
  extending SYMBOLTABLE[IDENTIFIER-TO-QID,ATTRIBUTE-TO-NAT] .
  var S : Stack .
  eq  isnew(S) = false .
endo

ev (recursion-limit :obj "SYMBOLTABLE" :op "retrieve" :limit 1)

reduce
    retrieve(addid(symt(svar(0)),ivar(0),avar(0)),ivar(1))
    =s=
    eif ivar(0) =s= ivar(1) then
        avar(0)
    else
        retrieve(symt(svar(0)),ivar(1))
    fi .

ev (recursion-limit :obj "SYMBOLTABLE"
```

(a): Instantiation and Reduction.

```
reduce in RETRIEVE : retrieve(addid(symt(svar(0)),ivar(0),avar(0)),
    ivar(1)) =s= eif ivar(0) =s= ivar(1) then avar(0) else retrieve(
    symt(svar(0)),ivar(1)) fi
rewrites: 18
result Bool: true
```

(b): Result.

Figure 41: Correctness Proof of retrieve Implementation.

```
object RETRIEVE is
      extending SYMBOLTABLE[IDENTIFIER-TO-QID,ATTRIBUTE-TO-NAT] .
      op  p1 : Symboltable -> Bool .
      using PROOF-CASE * (sort Sort to Symboltable, op p to p1) .
      var S : Stack .
      eq  p1(symt(S)) = isnew(S) =s= false .
endo

reduce p1(init) .

object RETRIEVE1 is extending RETRIEVE .
  eq isnew(svar(0)) = false .
endo

reduce p1(enterblock(symt(svar(0)))) .
reduce p1(addid(symt(svar(0)),ivar(0),avar(0))) .
reduce p1(leaveblock(symt(svar(0)))) .
```

(a): Instantiations and Reductions.

```
reduce in RETRIEVE : p1(init)
rewrites: 4
result Bool: true
```

```
reduce in RETRIEVE1 : p1(enterblock(symt(svar(0))))
rewrites: 4
result Bool: true
```

```
reduce in RETRIEVE1 : p1(addid(symt(svar(0)),ivar(0),avar(0)))
rewrites: 5
result Bool: true
```

```
reduce in RETRIEVE1 : p1(leaveblock(symt(svar(0))))
rewrites: 10
result Bool: true
```

(b): Results.

Figure 42: Proof of Lemma for `retrieve` Proof.

the original work.

The final step is to prove the lemma assumed in Figure 41. For proof, the lemma is formulated as part of an equation for a predicate on symbol tables. If the predicate can be proven invariant — that is, if it is true for all symbol tables — then the lemma is true. Invariance is proven as before: by structural induction.

Figure 42 shows the proof. Once again, the proof object from Figure 40 is instantiated. This time, however, it is augmented with the predicate operation p1 and the case-analysis rule from PROOF-CASE. The first reduction is the basis; it shows that the predicate is true for an initialised symbol table. Object RETRIEVE1 is the inductive hypothesis; it augments RETRIEVE with an equation that makes the predicate true for symbol tables constructed from one fewer operation invocation than those considered in the inductive steps. The last three reductions are inductive steps; they show that the predicate holds for symbol tables constructed from one more operation invocation than those assumed in the inductive hypothesis.

In general, an inductive hypothesis is necessary for structural induction. For this proof, however, the equation added in RETRIEVE1 is never used; the equations from RETRIEVE are sufficient to prove the inductive steps. Incidentally, only the `leaveblock` proof requires the case-analysis rule.

5.9 Conclusions and Future Work

Although an equational language is not typically used to record design decisions regarding concurrency, our work demonstrates that concurrent behavior can be specified equationally. Star operations can specify the allowable interaction of processes that are competing for some shared resource and message-passing operations can model interprocess communication. An equational language is also a useful tool for reasoning about an equational specification. In particular, properties of a specification can be verified and development steps can be proven correct. Actually, verifying a development step can be viewed as verifying a set of properties; however, the two activities are pragmatically distinct.

OBJ3 is a good language for this research. It has powerful parameterization mechanisms and a flexible type system. Subsorting and overloading is especially convenient. In addition, it is available, well documented, and appears to be readily modifiable. This last characteristic is only partially confirmed.

The work described here is experimental and preliminary. In fact, equational specification and verification is only one facet of an envisioned development methodology for concurrent software. This is a two-tiered methodology (cf. LARCH [Win87]), where equationally defined operations are used as auxiliary functions in an axiomatic specification of a concurrent program. The implementation tier is supported by a conventional concurrent programming language (e.g., SR [AOC*88]).

There is much more work to be done. The following proposals are in suggested chronological order.

As with any methodology, experience with more and larger examples is necessary to determine if it is general enough to be useful. There are several traditional benchmarks that seem suitable: the Dining Philosophers problem, the Sleeping Barber problem, and some variety of electronic-mail system. Each problem should require about one man-week to specify. Property verification tends to be more time consuming; however, two man-weeks per problem should be sufficient for small sets of fundamental properties.

The task of constructing a proof object should be substantially simplified — or even eliminated — thereby obviating the proof tools in Appendix 5.10. One way to accomplish this is to modify OBJ3 to reduce terms containing variables (cf. AFFIRM [TE81]). An estimate of the time required for this ambitious project should be deferred until the original developers are consulted. A less-ambitious modification is to overload the built-in operations so that they evaluate their arguments normally, symbolically, or eagerly depending upon the presence of variable operations. Such a project might require two or three man-months. Alternatively, a group of objects that implement a unification algorithm is available [GW88]. These should be evaluated for suitability as proof tools.

The semantics of conditional equations should be defined equationally, so that they can be reduced directly rather than requiring a transformation to regular equations. This should be easy.

The MESSAGE protocol should be shown to be general enough to model the message-passing protocols of currently available concurrent programming languages. In particular, the flexibility of SR's mechanisms should be captured.

Finally, the mathematical semantics of the message-passing operations, as well as their impact on the verification methodology, should be investigated. In addition, these operations should be implemented, by modifying the OBJ3 interpreter, thus constructing an interpreter for OBJC. One approach to mathematically specifying the semantics is that used in the definition of FOOPS [GM86b], an equational language supporting states (i.e., program variables). Namely, an OBJ3 interpreter could be implemented in OBJ3, then modified to support message passing. The semantics of OBJC would, therefore, be defined in terms of the semantics of OBJ3. A more straightforward definition — if attainable — would be desirable; the topic should be studied for one or two man-months. An OBJC implementation should require about three or four man-months.

5.10 Proof Tools

```
object PROOF-BOOL is
    protecting TRUTH-VALUE .
    protecting BOOL .

    op  eif_then_else_fi : Bool Bool Bool -> Bool
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Bool Bool -> Bool [strategy (1 2 0) prec 51] .

    var B T E : Bool .

    eq  eif true then T else E fi = T .
    eq  eif false then T else E fi = E .
    eq  eif B then true else true fi = true .
    eq  eif B then false else false fi = false .

    var B1 B2 : Bool .

    eq  B =s= B = true .

*** logical substitution

    eq  eif B then B =s= B1 else B2 fi =
        eif B then true =s= B1 else B2 fi .

    eq  eif B then B1 =s= B else B2 fi =
        eif B then B1 =s= true else B2 fi .

    eq  eif B then B1 else B =s= B2 fi =
        eif B then B1 else false =s= B2 fi .

    eq  eif B then B1 else B2 =s= B fi =
        eif B then B1 else B2 =s= false fi .

*** more logical substitution

    var A B C D E F G H I J K L : Bool .

    eq  eif A then (B or A) =s= true else C fi =
        eif A then (B or true) =s= true else C fi .

    eq  eif A then (B and A) =s= true else C fi =
        eif A then (B and true) =s= true else C fi .

    eq  eif A and B then (C and (A or D)) =s= true else E fi =
        eif A and B then (C and (true or D)) =s= true else E fi .

    eq  eif A and B then (C or B) =s= true else D fi =
        eif A and B then (C or true) =s= true else D fi .

endo

object PROOF-TRUTH[X :: TRIV] is
    protecting TRUTH-VALUE .
    protecting PROOF-BOOL .

    op  eif_then_else_fi : Bool Elt Elt -> Elt
        [strategy (1 2 3 0) gather (& & &) prec 0] .
    op  _=s=_ : Elt Elt -> Bool [strategy (1 2 0) prec 51] .

    var T E : Elt .

    eq  eif true then T else E fi = T .
    eq  eif false then T else E fi = E .

    eq  E =s= E = true .

    var P Q R : Bool .
    var A B C D : Elt .

*** if distribution

    eq  eif eif P then Q else R fi then A else B fi =
        eif P then eif Q then A else B fi else eif R then A else B fi fi .

*** logical substitution

    eq  eif P then eif P then A else B fi else C fi =
        eif P then A else C fi .

    eq  eif P then C else eif P then A else B fi fi =
        eif P then C else B fi .

    eq  eif P then eif Q then eif P then A else B fi else C fi else D fi =
        eif P then eif Q then A else C fi else D fi .

    eq  eif P then eif Q then C else eif P then A else B fi fi else D fi =
        eif P then eif Q then C else A fi else D fi .

    eq  eif P then D else eif Q then eif P then A else B fi else C fi fi =
        eif P then D else eif Q then B else C fi fi .

    eq  eif P then D else eif Q then C else eif P then A else B fi fi fi =
        eif P then D else eif Q then C else B fi fi .

endo

object PROOF-INT is
    protecting TRUTH-VALUE .
    protecting INT .
    protecting PROOF-TRUTH[INT] .

    op  _<s_ : Int Int -> Bool [prec 51] .
    op  _<s=_ : Int Int -> Bool [prec 51] .
    op  _>s_ : Int Int -> Bool [prec 51] .
    op  _>s=_ : Int Int -> Bool [prec 51] .

    var A B : Int .

    eq  A <s A = false .
    eq  A <s= A = true .
    eq  A >s A = false .
    eq  A >s= A = true .

    eq  0 <s 1 = true .
    eq  0 <s= 1 = true .
    eq  0 >s 1 = false .
    eq  0 >s= 1 = false .

    eq  A + 1 >s= B = ((A >s= B) or (A =s= B - 1)) .
    eq  A + 1 <s= B = ((A <s B - 1) or (A =s= B - 1)) .
    eq  A + -1 >s= B = A >s B .
    eq  A + -1 <s= B = ((A <s= B) or (A - 1 =s= B)) .

endo

object PROOF-CASE is
    sort Sort .

    op  p : Sort -> Bool .
    op  eif_then_else_fi : Bool Sort Sort -> Sort .
    op  eif_then_else_fi : Bool Bool Bool -> Bool .

    var B : Bool .
    var S1 S2 : Sort .

    eq  p(eif B then S1 else S2 fi) =
        eif B then
            p(S1)
        else
            p(S2)
        fi .

endo
```

6 Verification of Security in OBJ

6.1 Introduction

This section discusses the OBJ3 system [GW88] and its usefulness in the verification of security properties in operating systems. In addition, an example of an OBJ3 specification of a simple operating system with security properties is presented.

6.1.1 Using OBJ3 to Verify Security

This section will discuss ways in which OBJ3 may be used to verify security. The general format is meant to correspond to [CM81], which compares the specification systems/languages HDM, Special, Ina Jo, FDM, Gypsy, and AFFIRM. The types of security properties which are of interest include the following (description taken from [Lan83]):

- *Simple security condition.* A subject can read an object only if the security level of the subject is at least that of the object.
- **-property.* A subject can modify an object O_1 in a manner dependent on an object O_2 only if the security level of O_1 is at least that of O_2.

Suggestions for the implementation of information flow models and take-grant models will be discussed in the following sections.

Information Flow Models OBJ3 does not have any tools specifically designed for verifying security, unlike HDM (which has an associated multilevel security formula generator). Information flow analysis, which focuses on the transfer of information between objects, would be difficult to perform directly in OBJ3. This is partially caused by OBJ3's lack of the concept of state: OBJ3 may only reduce terms in isolation, not sequences of terms. Thus, there is no way to directly implement side effects. However, one can get around this problem by implementing state as an OBJ3 object. For example, one might define a module ARCHITECTURE which contains sorts Memory, Register, and Mar, then define specific operations which modify items having these sorts. Undesirable information flow would occur when one of the operations causes information to be moved improperly within the items. For example, if sort Memory could be divided into private and public parts, then undesirable information flow would occur if items located in the private area could be moved into the public area. One could attempt to determine whether this could occur by formulating a boolean expression which stated that the information *had* been moved improperly. If the system reduced this expression to boolean *true*, then the system clearly permits undesirable flow.

One disadvantage of the above technique is that the system must be "tricked" into doing the analysis of information flow. It is possible that the rewrite engine will not "find" the appropriate sequence of events which will cause undesirable information flow in a reasonable amount of time; thus, one cannot rely upon a *false* reply to an expression stating undesirable flow. In addition, covert channels cannot always be detected by this method (particularly those which rely upon timing or system characteristics, such as page fault rate).

Take-Grant Models Take-grant models use digraphs with labeled arcs. Each label defines whether the vertex at the arc's origin has a particular "right" over the vertex at the end of the arc.

In this context, rights refer to the ability to *take* or *grant* access rights to another node. Take-grant models are used to answer the question "Can a subject (node) A gain access to an object (node) B?"

OBJ3 provides a natural method of analysis for this model. An object LABELED-DIGRAPH can be defined; this object would export functions which perform graph traversals and the sort Labeled-digraph. Traversal from one node to the other would depend upon the labeling of the arc between them; for example, if node A has *take* rights over node B, then any node C which has a path to B would also have a path to A. Similarly, if node B has *grant* rights over node A, then any node C which has a path to B would have a path to A. The user could then instantiate (view) a Labeled-digraph item with the desired configuration. Finally, the user would request the system to reduce an expression declaring that a path exists between a particular subject and object. An affirmative result would indicate that the subject could gain rights to the object; a negative result would indicate the opposite.

Unlike the information flow method, all answers returned by OBJ3 are conclusive. Of course, the user is still at the mercy of the rewrite engine and may find that large graphs require a great deal of computation. Additionally, covert channels are still not addressed.

6.1.2 General Observations

OBJ3 has a fairly easy syntax to learn, although it is somewhat unforgiving regarding punctuation. Since OBJ3 specifications are executable — via term rewriting — they are somewhat easier to work with than purely specificational systems. Not all properties may be proven using term rewriting (and others cannot be proven efficiently in this fashion); this implies that some human intervention is necessary.

OBJ3 does not specifically provide tools which manipulate security properties or permit specification of concurrency. However, it is possible to compensate for this by building objects which provide such concepts as state and input/output histories.

6.2 An OBJ3 Specification oí A Simple Operating System

In this section, an OBJ3 specification of a secure operating system will be presented. The operating system is based on the one described in [Lev80]. In brief, the system may be broken down into three components: the architecture, the supervisor, and the user processes. Each of these components will be described by one or more OBJ3 objects. The following sections consist of a description of the component being defined by the object, the OBJ3 code, and asummary of the code.

6.2.1 Architecture

The architecture used in this system is very simple. It consists of a memory, a collection of user registers, and a memory management unit (mmu). These components are all built out of registers, each of which contains a single word of information.

The following operations may be performed:

- Fetch.
- Store.
- Trap.

The fetch instruction causes information transfer from a memory register into a user register; the store instruction transfers information from a user register to a memory register, and the trap instruction transfers control to the supervisor (generally so that a supervisor — privileged — instruction may be executed).

The following pages contain the object definition of object ARCHITECTURE.

```
obj ARCHITECTURE is
*** The sorts provided by ARCHITECTURE
sorts Regblock Register Block-num Displacement .
sorts Memory Block Mmu .
*** Imported modules
protecting INT .
protecting BOOL .
*** A partial ordering of sorts
subsort Int < Displacement Block-num  .
subsort Block < Memory .
subsort Mmu < Regblock .
***
*** register operations
***
op (__) : Int Int -> Register .
op (__) : Register Regblock -> Regblock .
op _[_] : Regblock Int -> Register .
op Reg-val : Register -> Int .
op Reg-id : Register -> Int .
op Set-reg : Regblock Int Int -> Regblock .
op nil : ->  Regblock .
op nilregister : -> Register .
op Clear-regblock : Regblock -> Regblock .
***
*** memory operations
***
op (__) : Int Regblock -> Block .
op Block-id : Block -> Int .
op Block-contents : Block -> Regblock .
op Clear-block : Block -> Block .
op  __ : Block Memory -> Memory .
op eom : -> Memory .
op Mem : Memory Block-num Displacement -> Register .
op Set-mem : Memory Block-num Displacement Int -> Memory .
op Set-mem : Memory Block-num Regblock -> Memory .
op _[_] : Memory Block-num -> Regblock .
op Clear-mem : Memory -> Memory .
***
*** MMU operations
***
op __ : Register Register -> Mmu .
op Mar : Mmu Int -> Register .
op Set-mar : Mmu Int Int -> Mmu .
op Regblock-to-mmu : Regblock -> Mmu .
***                          Equational Part               ***
var I I' : Int .
var J K  : Int .
var R R' : Register .
var Rb Rb' : Regblock .
var B B' : Block-num .
var Bl Bl' : Block .
var D D' : Displacement .
var M M' : Memory .
***
*** register
***
eq nil[I] = nilregister .
eq ((I I') Rb)[J] =  if I == J then (I I') else Rb[J] fi .
eq Reg-val( I I' ) = I' .
eq Reg-val( nilregister ) = -1 .
eq Reg-id( I I' ) = I .
eq Reg-id( nilregister ) = -1 .
eq Set-reg(nil, I, I') = nil .
eq Set-reg(((I I') Rb), J, K) =
if I == J then (I K) Rb else (I I') Set-reg(Rb, J, K) fi .
eq Clear-regblock(nil) = nil .
eq Clear-regblock((I I') Rb) = (I 0) Clear-regblock(Rb)  .
```

```
***
*** memory
***
eq Block-id(I Rb) = I .
eq Block-contents(I Rb) = Rb .
eq Clear-block( I Rb) = I Clear-regblock(Rb) .
eq Mem(Bl M, B, D) =
if B == Block-id(Bl) then (Block-contents(Bl))[D] else Mem(M,B,D) fi .
eq Mem(eom, B, D) = nilregister .
eq Set-mem(Bl M, B, D, J) =
if B == Block-id(Bl) then (B Set-reg(Block-contents(Bl), D, J)) M
        else Bl Set-mem(M,B,D,J) fi .
eq Set-mem(Bl M, B, Rb) =
if B == Block-id(Bl) then (B Rb) M
        else Bl Set-mem(M,B,Rb) fi .
eq Set-mem(eom, B, D, J) = eom .
eq (Bl M)[B] =
if B == Block-id(Bl) then Block-contents(Bl)
        else M[B] fi .
eq Clear-mem(Bl M) = Clear-block(Bl) Clear-mem(M) .
eq Clear-mem(eom) = eom .
***
*** mmu
***
eq Mar(R R', I) =
if I == 0 then R else R' fi .
eq Set-mar(R R', I, I') =
if I == 0 then (I I') R' else R (I I') fi .
eq Regblock-to-mmu( Rb ) = (Rb[0] Rb[1]) .
endo .
```

The most important sorts provided by ARCHITECTURE are Memory, Mmu, Register, and Block. Sort Register is defined to be a pair of integers; the first in the pair gives the register id number and the second is the value contained in the register. Sort Block contains an id number and sequence of registers Sort Memory is defined to be a collection of blocks. Sort Mmu is defined as a pair of registers. Note that this specification does not require a particular size memory; for brevity's sake it was decided that the user would be responsible for instantiating the proper size memory.

ARCHITECTURE provides many operations for manipulating its sorts; among them are operations for concatenation of registers (to form blocks), an array-style notation for getting the value of a particular register in a register block or in memory, operations to clear memory/blocks/registers (*clear* means *set to zero*), and operations for changing the value of a particular register in memory.

It is worth noting that the sorts Displacement and Block-num have been defined as supersorts of sort Int (integer). This indicates that all integer values are also Displacements (or Block-num) and also implies that there may be Displacements (or Block-num) which are not integers.

6.2.2 Supervisor

The supervisor is fairly complicated, so four objects have been used to define it: OPSYS, SUPERVISOR, SUP-OPS, RUN-SUPERVISOR. The object OPSYS contains objects which are "borderline" between the architecture of the system and the system's supervisor. For example, OPSYS defines the state of the system (State) as consisting of the mode of operation, the contents of the memory, the contents of the mmu, and the contents of the user registers. OPSYS also provides functions which allow the manipulation of State's components. Additionally, OPSYS provides the unprivileged mode instructions described earlier.

```
obj OPSYS is extending ARCHITECTURE .
sort State .
sort Mode .
op Initial-state : Mode Memory Regblock Regblock -> State .
op Set-state-reg : State Int Int -> State .
op Set-state-reg : State Regblock -> State .
op Set-state-mmu : State Int Int -> State .
op Set-state-mmu : State Regblock -> State .
op Set-state-mem : State Block-num Displacement Int -> State .
op Set-state-mem : State Memory -> State .
op Get-state-reg : State  -> Regblock .
op Get-state-mmu : State  -> Mmu .
op Get-state-mem : State  -> Memory .
op Get-state-mod : State  -> Mode .
op privileged  : -> Mode .
op unprivileged  : -> Mode .
*** unprivileged mode instructions
op Fetch  : State Int Int Int -> State .
op Store  : State Int Int Int -> State .
op Compute  : State  Int Int -> State .
op Trap : State -> State .
vars M M' : Memory .
vars Rb Rb' : Regblock .
vars  Mu Mu' : Mmu .
vars S S' : State .
vars D : Displacement .
vars B : Block-num .
vars I J K N : Int .
vars Mo Mo' : Mode .
eq Get-state-reg( Initial-state(Mo, M, Rb, Mu) ) = Rb .
eq Get-state-mmu( Initial-state(Mo, M, Rb, Mu) ) = Mu .
eq Get-state-mem( Initial-state(Mo, M, Rb, Mu) ) = M .
eq Get-state-mod( Initial-state(Mo, M, Rb, Mu) ) = Mo .
eq Get-state-reg( Set-state-mem(Initial-state(Mo, M, Rb, Mu), B, D, N )) = Rb .
eq Get-state-mmu( Set-state-mem(Initial-state(Mo, M, Rb, Mu), B, D, N )) = Mu .
eq Get-state-reg( Set-state-mmu(Initial-state(Mo, M, Rb, Mu), I, J )) = Rb .
eq Get-state-mem( Set-state-mmu(Initial-state(Mo, M, Rb, Mu), I, J )) = M .
eq Get-state-mmu( Set-state-reg(Initial-state(Mo, M, Rb, Mu), I, J )) = Mu .
eq Get-state-mem( Set-state-reg(Initial-state(Mo, M, Rb, Mu), I, J )) = M .
eq Set-state-mem(Initial-state(Mo, M, Rb, Mu), B, D, N ) =
Initial-state(Mo, Set-mem(M, B, D, N), Rb, Mu) .
eq Set-state-mem(Initial-state(Mo, M, Rb, Mu), M' ) =
Initial-state(Mo, M', Rb, Mu) .
eq Set-state-reg(Initial-state(Mo, M, Rb, Mu), I, J) =
Initial-state(Mo,  M, Set-reg(Rb, I, J), Mu) .
eq Set-state-reg(Initial-state(Mo, M, Rb, Mu), Rb') =
Initial-state(Mo,  M, Rb', Mu) .
eq Set-state-mmu(Initial-state(Mo, M, Rb, Mu), I, J) =
Initial-state(Mo,  M, Rb, Set-mar(Mu, I, J) ) .
eq Set-state-mmu(Initial-state(Mo, M, Rb, Mu), Mu') =
Initial-state(Mo,  M, Rb, Mu' ) .
eq Set-state-mmu(Initial-state(Mo, M, Rb, Mu), Rb') =
Initial-state(Mo,  M, Rb, Rb' ) .
eq Fetch(S, I, J, K) = Set-state-reg(S, I,
Reg-val(Mem( Get-state-mem(S), Reg-val(Mar(Get-state-mmu(S),J)), K))) .
eq Store( S, I, J, K) = Set-state-mem( S,
Reg-val(Mar(Get-state-mmu(S),J)),
K,
Reg-val((Get-state-reg(S))[I])) .
eq Trap( Initial-state(unprivileged, M, Rb, Mu) ) =
Initial-state(privileged, M, Rb, Mu) .
endo .
```

The second object which is used to define the system supervisor is named SUPERVISOR. This object defines the supervisor state (Supstate) as a collection of supervisor variables and the system state (State) defined previously. The supervisor variables defined are: the current process id (CP), a list containing each process' authorization level (PAL), a save area for each process' user registers and mapping registers (SR, SMAR), and information regarding the security level of each block in the system: authorization level (BAL), status (BAP), access count (BAC), attached to device (BDF).

In addition to providing operations which read/set each of these supervisor variables, SU-

PERVISOR defines authorization levels (Alevel). This is done by taking advantage of the *subsort* construct to declare integers as a subsort of Alevels, providing two operations Syshi and Syslo which are defined to return sort Alevel, and then defining the value of the < operator when applied to integers and Alevels. It is interesting to note that only two equations are needed in addition to the subsort and the op definitions. The first of these equations declares that $I < Syshi$ is true for all integers I, and the second declares that $I < Syslo$ is false for all integers I. Since Int is a subsort of Alevel, all Ints are automatically of sort Alevel; however, the standard integer < operation will be used to compare them. This combination of declarations ensures that all "integer" authorization levels will automatically fall between Syshi and Syslo.

Another item worth noting is the use of overloading to simplify setting values in SMAR and SR. This permits the user to use the same function call (Set-Smar/Set-Sr) to modify either one of the SMAR registers or the entire SMAR block. It is also interesting to observe that object SUPERVISOR imports OPSYS via an *extending* clause; this permits SUPERVISOR to add new data items having sorts defined originally in OPSYS.

```
obj SUPERVISOR is
protecting INT .
protecting BOOL .
protecting ARCHITECTURE .
extending OPSYS .
sorts Supstate Alevel  Pid  .
sorts Block-info-list Block-info  .
subsort Int < Alevel .
subsort Pid < Int .
op Syshi    : -> Alevel .
op Syslo    : -> Alevel .
op nillevel : -> Alevel .
op (__) _   : Int Alevel Regblock  -> Regblock  .
op _[_]     : Regblock  Int -> Alevel .
op _<_     : Int Alevel -> Bool .
op Next-Cp : Pid -> Pid .
op Cp : Supstate -> Pid .
op Pal : Supstate Pid -> Alevel .
op Sr : Supstate Pid  -> Regblock .
op Smar : Supstate Pid -> Regblock .
op Bal : Supstate Int -> Alevel .
op Bap : Supstate Int -> Bool .
op Bac : Supstate Int -> Int .
op Bdf : Supstate Int -> Bool .
op Set-Cp : Supstate Pid -> Supstate .
op Set-Pal : Supstate Pid Int -> Supstate .
op Set-Sr : Supstate Pid Regblock -> Supstate .
op Set-Smar : Supstate Pid Regblock -> Supstate .
op Set-Bal : Supstate Int Alevel -> Supstate .
op Set-Bap : Supstate Int Bool -> Supstate .
op Set-Bac : Supstate Int Int -> Supstate .
op Set-Bdf : Supstate Int Bool -> Supstate .
op Get-State : Supstate -> State .
op Set-State : Supstate State -> Supstate .
op Make-block : Int Alevel  Bool Int Bool -> Block-info .
op bal : Block-info -> Alevel  .
op bap : Block-info -> Bool .
op bac : Block-info -> Int .
op bdf : Block-info -> Bool .
op set-bal : Block-info Alevel  -> Block-info .
op set-bap : Block-info Bool -> Block-info .
op set-bac : Block-info Int -> Block-info .
op set-bdf : Block-info Bool -> Block-info .
op set-bal-list : Block-info-list Int Alevel -> Block-info-list .
op set-bap-list : Block-info-list Int Bool -> Block-info-list .
op set-bac-list : Block-info-list Int Int -> Block-info-list .
op set-bdf-list : Block-info-list Int Bool -> Block-info-list .
op Block-id   : Block-info -> Int .
op __ : Block-info Block-info-list -> Block-info-list .
op _[_] : Block-info-list Int -> Block-info .
op nil-b : -> Block-info-list .
op nilblock : -> Block-info .
op Init-supstate : Pid Regblock  Memory Memory Block-info-list
State -> Supstate .
vars P P'  : Pid .
vars A A'  : Regblock  .
vars Ai Ai'  : Alevel .
vars Srl Srl' : Memory .
vars Sml Sml' : Memory .
vars Bil Bil' : Block-info-list .
var Bi : Block-info .
vars R1 R2 R3 R : Register .
vars St St' : State .
vars Su Su' : Supstate .
vars I J K L M N : Int .
vars T1 T2 T3 : Bool .
vars Me Me' : Memory .
vars Re Re' : Regblock .
*** level comparisons
eq I < Syshi = true .
eq I < Syslo = false .
*** getting elements from a block list
eq Block-id( Make-block(I, Ai, T1, N, T2) ) = I .
eq (Bi Bil)[I] = if Block-id(Bi) == I then Bi else Bil[I] fi .
eq nil-b[I] = nilblock .
*** setting list elements in block list
eq set-bap-list( Bi Bil, I, T1) =
if Block-id(Bi) == I then set-bap(Bi, T1) Bil
else Bi set-bap-list(Bil, I, T1) fi .
eq set-bal-list( Bi Bil, I, Ai) =
if Block-id(Bi) == I then set-bal(Bi, Ai) Bil
else Bi set-bal-list(Bil, I, Ai) fi .
eq set-bac-list( Bi Bil, I, J) =
if Block-id(Bi) == I then set-bac(Bi, J) Bil
else Bi set-bac-list(Bil, I, J) fi .
eq set-bdf-list( Bi Bil, I, T1) =
if Block-id(Bi) == I then set-bdf(Bi, T1) Bil
else Bi set-bdf-list(Bil, I, T1) fi .
*** internal block access routines
eq bal(Make-block(I, Ai, T1, N, T2)) = Ai .
eq bap(Make-block(I, Ai, T1, N, T2)) = T1 .
eq bac(Make-block(I, Ai, T1, N, T2)) = N .
eq bdf(Make-block(I, Ai, T1, N, T2)) = T2 .
eq set-bal(Make-block(I, Ai, T1, N, T2), Ai') = Make-block(I, Ai', T1, N, T2) .
eq set-bap(Make-block(I, Ai, T1, N, T2), T3) = Make-block(I, Ai, T3, N, T2) .
eq set-bac(Make-block(I, Ai, T1, N, T2), M) = Make-block(I, Ai, T1, M, T2) .
eq set-bdf(Make-block(I, Ai, T1, N, T2), T3) = Make-block(I, Ai, T1, N, T3) .
*** Read values from supervisor state
eq Cp( Init-supstate( P, A, Srl, Sml, Bil, St )) = P .
eq Pal( Init-supstate( P, A, Srl, Sml, Bil, St ), P') = A[P'] .
eq Sr( Init-supstate( P, A, Srl, Sml, Bil, St ), P') = Srl[P'] .
eq Smar( Init-supstate( P, A, Srl, Sml, Bil, St ), P') = Sml[P'] .
eq Bal( Init-supstate( P, A, Srl, Sml, Bil, St ), I) = bal(Bil[I]) .
eq Bap( Init-supstate( P, A, Srl, Sml, Bil, St ), I) = bap(Bil[I]) .
eq Bac( Init-supstate( P, A, Srl, Sml, Bil, St ), I) = bac(Bil[I]) .
eq Bdf( Init-supstate( P, A, Srl, Sml, Bil, St ), I) = bdf(Bil[I]) .
eq Get-State( Init-supstate(P,A,Srl,Sml,Bil,St)) = St .
*** Write values of supervisor state
eq Set-Cp( Init-supstate( P, A, Srl, Sml, Bil, St ), P') =
Init-supstate(P',A,Srl,Sml,Bil,St) .
eq Set-Pal( Init-supstate( P, A, Srl, Sml, Bil, St ), P', N) =
Init-supstate(P, Set-reg(A,P',N), Srl, Sml, Bil, St) .
eq Set-Sr( Init-supstate( P, A, Srl, Sml, Bil, St ), P', Re) =
Init-supstate(P, A, Set-mem(Srl, P', Re), Sml, Bil, St) .
eq Set-Smar( Init-supstate( P, A, Srl, Sml, Bil, St ), P', Re) =
Init-supstate(P, A, Srl, Set-mem(Sml, P', Re), Bil, St ) .
eq Set-Bal( Init-supstate( P, A, Srl, Sml, Bil, St ), I, N) =
Init-supstate(P, A, Srl, Sml, set-bal-list(Bil,I,N), St) .
eq Set-Bap( Init-supstate( P, A, Srl, Sml, Bil, St ), I, T1) =
Init-supstate(P, A, Srl, Sml, set-bap-list(Bil,I,T1), St) .
eq Set-Bac( Init-supstate( P, A, Srl, Sml, Bil, St ), I, N) =
Init-supstate(P, A, Srl, Sml, set-bac-list(Bil,I,N), St) .
eq Set-Bdf( Init-supstate( P, A, Srl, Sml, Bil, St ), I, T1) =
Init-supstate(P, A, Srl, Sml, set-bdf-list(Bil,I,T1), St) .
eq Set-State( Init-supstate( P, A, Srl, Sml, Bil, St ), St') =
Init-supstate(P, A, Srl, Sml, Bil, St') .
eq Next-Cp( P ) = if P == 1 then 2 else 1 fi .
endo .
```

The third object making up the supervisor component of the system is SUP-OPS. This object contains the definitions of the four privileged instructions provided by the system: Purge, Raise, Get, Swap. These instructions are defined in detail in [Lev80]; a brief description follows here.

Purge is used to set the contents of a memory block to zero and raise its authorization level to Syshi (effectively making it unusable by a user process). Purging may only take place in privileged mode, and only when no processes are accessing the block and the block is not attached permanently to an input/output device.

Raise is used to change the status of a block to 'active' and change its authorization level to match that of the current process. Raise is only performed on a purged block; in effect, Raise makes a purged memory block available to a user process.

Get is used to give a process access to a particular block. This is done by setting the block's access count to one, placing a pointer to the block in the process' mmu, and setting to zero the access count of the block that the process' mmu previously indicated. Get may only be performed upon blocks that are active, not currently being accessed, and having an authorization level equal to that of the current process. This prevents a process from using Get to gain unauthorized access to another process' data.

Swap is a simple operation; it is used to switch current processes. Swap is responsible for transferring all of the current process' user registers (and mmu) into temporary storage so that the succeeding process cannot manipulate it. Any process may call Swap.

It is worth noting that object SUP-OPS contains all the explicit security information. By that it is meant that the security policy of the system is embedded in the four supervisor operations (i.e., processes may access only blocks at the same authorization level). It is in this block that the user could modify security policy. For example, one might wish to permit processes to access blocks with authorization levels less than or equal to their own; this would require modifying the precondition for operation Get. One might also like to specify both a read and a write authorization policy, such as the one given by the conjunction of *-property and simple security. These could also be accomplished by modifying SUP-OPS (and some modifications to Supstate in SUPERVISOR). OBJ3's hierarchical structure is very useful for localizing security properties.

```
obj SUP-OPS is
protecting BOOL .
protecting INT .
protecting ARCHITECTURE .
protecting OPSYS .
extending SUPERVISOR .
op Purge : Supstate Int -> Supstate .
op Raise : Supstate Int -> Supstate .
op Get : Supstate Int Int -> Supstate .
op Swap : Supstate  -> Supstate .
vars P P' : Pid
vars A A' : Regblock .
vars Ai Ai' : Alevel
vars Srl Srl'  : Regblock .
vars Sml Sml'  : Regblock .
vars Bil Bil : Block-info-list .
vars Bi Bi' : Block-info .
vars  R1 R2 R3 : Register .
vars St St' : State .
vars Su Su' : Supstate .
vars I J K L M N : Int .
vars Q R S T : Bool .
```

```
eq Purge( Su, N) =
if (Bac(Su,N) == 0) and (not Bdf(Su, N)) then
Set-State(Set-Bap(Set-Bal( Su, N, Syshi), N, not Bap(Su,N)),
  Set-state-mem(Get-State(Su),
Clear-mem(Get-state-mem(Get-State(Su)))))
else Su fi .
eq Raise( Su, N) =
if (not Bap(Su,N)) then
Set-Bap(Set-Bal(Su, N, Pal(Su, Cp(Su))),
N, true )
else Su fi .
eq Get( Su, N, K) =
if Bap(Su,N) and (Bac(Su,N)== 0) and
(Bal(Su,N) == Pal(Su,Cp(Su)))
then
Set-Bac(Set-Bac(Su,N,1),
Reg-val(Mar(Get-state-mmu(Get-State(Su)),K)),
0)
else Su fi .
eq Swap(Su) =
      Set-State(
Set-Smar(
    Set-Sr(Set-Cp(Su, Next-Cp(Cp(Su))), Cp(Su),
Get-state-reg(Get-State(Su))),
    Cp(Su),
    Get-state-mmu(Get-State(Su))),
Set-state-mmu(
Set-state-reg(Get-State(Su),Sr(Su,Next-Cp(Cp(Su)))),
Regblock-to-mmu(Smar(Su, Next-Cp(Cp(Su)) )))).
endo .
```

6.2.3 User Processes

Both user processes and the final portion of the supervisor will be discussed in this section, as they are intertwined.

The user process object USER-PROCESS provides the sorts Instruction, Instruction-sequence, and Process. Instructions are composed of instruction names and their arguments (zero to three arguments). Once again, overloading is used to permit a 'variable number' of arguments in a strongly sorted system. Instruction-sequences are simply ordered sequences of instructions. Processes are made up of an instruction sequence. No other information was included in Process at this time, in the interest of brevity.

In addition to these sorts, USER-PROCESS provides operations to remove an instruction from an instruction sequence or a process and to read (without removing) an instruction.

The object RUN-SUPERVISOR embodies the supervisor chores which correspond to process manipulation. This object provides the sort System-state, which consists of the supervisor state combined with the state of the system processes (only two processes provided in this implementation). The operations provided by RUN-SUPERVISOR are: Execute (which causes a process to 'execute' one of its instructions), and Step (which returns the System-state after the current process has executed an instruction). At present, processes are themselves responsible for calling Swap; however, if the concept of fair scheduling were to be introduced it would be in this module that the changes would be placed.

```
::::::::::::::
user-process.obj
::::::::::::::
obj USER-PROCESS is
protecting INT .
protecting SUPERVISOR .
sort Instruction .
sort Instruction-sequence .
sort Instruction-name .
sort Process .
```

```
subsort Instruction < Instruction-sequence .
op Do-Fetch : -> Instruction-name .
op Do-Store : -> Instruction-name .
op  Do-Trap  : -> Instruction-name .
op Do-Purge : -> Instruction-name .
op Do-Raise : -> Instruction-name .
op Do-Get   : -> Instruction-name .
op Do-Swap  : -> Instruction-name .
op Instr : Instruction-name Int Int Int -> Instruction .
op Instr : Instruction-name Int Int  -> Instruction .
op Instr : Instruction-name Int  -> Instruction .
op Instr : Instruction-name -> Instruction .
op eoi   : -> Instruction .
op nil-instruction : -> Instruction .
op __ : Instruction Instruction-sequence -> Instruction-sequence .
op Get-instruction : Instruction-sequence -> Instruction .
op Remove-instruction : Instruction-sequence -> Instruction-sequence .
op Init-process : Instruction-sequence -> Process .
op Advance : Process -> Process .
op Read-instruction : Process -> Instruction .
var In In' : Instruction .
var Ins Ins' : Instruction-sequence .
eq Get-instruction(In Ins) = In .
eq Remove-instruction(In Ins) = Ins .
eq Remove-instruction(eoi) = eoi .
eq Advance( Init-process(In Ins)) = Init-process(Ins) .
eq Advance( Init-process(eoi)) = Init-process(eoi) .
eq Read-instruction( Init-process( In Ins)) = In .
eq Read-instruction( Init-process(eoi)) = nil-instruction .
endo
::::::::::::::
run-supervisor.obj
::::::::::::::
obj RUN-SUPERVISOR is
protecting INT .
protecting USER-PROCESS .
protecting SUPERVISOR .
protecting SUP-OPS .
protecting ARCHITECTURE .
protecting OPSYS .
sort System-state .
op Select-process : System-state Int -> Process .
op Init-system-state : Supstate Process Process -> System-state .
op Run : System-state -> System-state .
op Step : System-state -> System-state .
op Execute : Supstate Instruction -> Supstate .
vars Su Su' : Supstate .
vars Sst Sst' : System-state .
vars Proc Proc' : Process .
vars I J K : Int .
eq Select-process(Init-system-state(Su, Proc, Proc'), I) =
if I == 0 then Proc else Proc' fi .
eq Step(Init-system-state(Su, Proc, Proc')) =
if Cp(Su) == 0 then
    Init-system-state(Execute( Su, Read-instruction(Proc)),
 Advance(Proc),
 Proc')
else Init-system-state(Execute(Su, Read-instruction(Proc')),
 Proc,
 Advance(Proc') ) fi .
eq Execute( Su, Instr(Do-Fetch, I, J, K)) =
Set-State(Su, Fetch(Get-State(Su), I, J, K)) .
eq Execute( Su, Instr(Do-Store, I, J, K)) =
Set-State(Su, Store(Get-State(Su), I, J, K)) .
eq Execute( Su, Instr(Do-Trap, I, J, K)) =
Set-State(Su, Trap(Get-State(Su))) .
eq Execute( Su, Instr(Do-Purge, I)) = Purge( Su, I ) .
eq Execute( Su, Instr(Do-Raise, I)) = Raise( Su, I ) .
eq Execute( Su, Instr(Do-Get, I, J)) = Get( Su, I, J ) .
eq Execute( Su, Instr(Do-Swap)) = Swap( Su ) .
endo .
```

6.2.4 Conclusion

As shown by the example, it is possible to use OBJ3 to specify a simple operating system. An advantage of using OBJ3 is that one may write a sample term defining the state of the system (including process information, etc.) and let OBJ3 apply the reduction rules. This permits specifications to be exercised, which allows the user to decide whether the system the specifications describe corresponds to the system the user desires. Additionally, the modular form of OBJ3 specifications permits the user to experiment with, for example, different security models within the system. The hierarchical design of an OBJ3 specification, combined with restricted forms of importation should also lead to easier maintenance of the specification by reducing the scope of modifications.

6.3 An alternate security model

```
***
***  This security object provides the operations
***  may-read and may-write, which have been
***  defined to comply with the *-property and
***  simple security (DoD model)
***

obj SECURITY is sorts Security .
 protecting INT .
 protecting BOOL .
 subsorts Security < Int .

 *** security levels
 op top-secret : -> Security .
 op secret : -> Security .
 op classified : -> Security .
 op confidential : -> Security .

 *** security operations
 op level(_) : Security -> Int .
 op may-read(_ _) : Security Security -> Bool .
 op may-write(_ _) : Security Security -> Bool .
 op may-declassify(_ _ _) : Security Security Security -> Bool .
 op may-classify(_ _ _) : Security Security Security -> Bool .
 op _>_ : Security Security -> Bool [assoc] .

 *** equational specification
 vars S T U : Security .

 *** assign levels
 eq level(top-secret) = 4 .
 eq level(secret) = 3 .
 eq level(classified) = 2 .
 eq level(confidential) = 1 .
 eq S > T =  level(S) > level(T) .

 *** can read down
 eq may-read(S T) =  S == T or level(S) > level(T) .
 *** can write up
 eq may-write(S T) = S == T or level(T) > level(S) .

 *** reduction/increase of classification level can only be done by t-s level
 eq may-declassify(S T U) = level(S) == level(top-secret) and level(T) > level(U) .
 eq may-classify(S T U) = level(S) == level(top-secret) and level(U) > level(T) .
endo
```

7 Specification and Testing of Security in FASE

7.1 Introduction

In response to the National Computer Security Center's demand for A1 certified systems, there is increasing interest in verification systems that can reason about specifications. An operating system is A1 certified if its specifications are verified to be correct with respect to a policy that limits (1) the objects a process can access based on the security level of the process and object, and (2) the flow of information between processes based on their security levels; as discussed by Goguen and Meseguer [GM82b][GM84b] Feiertag, et al. [FLR77], and Rushby [Rus84], (2) subsumes (1). A number of verification systems have been developed with this need in mind, several specialized to carry out security flow analysis.

Clearly, program verification is difficult. However, verification of a specification with respect to the security flow policy need not be as difficult as verifying a large program such as an operating system with respect to a specification that fully describes what the operating system does. The security policy is expressed as follows: Given a sequence of operations invoked by processes at different security levels, an operation at a given level L1 can interfere with the results of an operation at level L2 only if L1 $\leq$ L2. It is difficult to verify a system with respect to this policy, as it entails a proof that considers every sequence of operations. It can be shown that the verification effort can be significantly simplified to just showing that the individual specifications of operations satisfy a policy. However, this method generally requires that the objects have constant security levels [FLR77]. This assumption is clearly unacceptable if the specifications are at a level of abstraction where objects are shared among users of different security levels – clearly the case for most operating systems. There are various tricks for converting a specification that allows for mutable security levels to one in which the security levels are constant, but at the cost of significantly increasing the cost of certification of the implementation – by verification, testing, etc.

Furthermore, it does not make sense for verification to be the only supported certification method. Most specifications will contain errors that are better detected by conventional testing. Furthermore, although not mandated for A1 certification, verification of a system's specifications does not guarantee system correctness; errors can certainly occur in an implementation. Although verification of the implementation of complex operating systems is beyond the stamina of human and mechanical verifiers, verification of abstractions below the top-level operating system interface is feasible. However, at the more detailed levels the assumptions underlying the mechanical flow analyzers are no longer valid.

Accordingly, this section presents two implemented tools that can assist in the testing of operating systems with respect to security policies; the tools can be used, in principle, at any level of abstraction.

The first tool is based on the Final Algebra Specification and Execution system (FASE) [KJA83], and would be used to test specifications with real input values. FASE uses an executable specification language which is operational in style, elements being represented in terms of their observable behavior. The style allows for specifications that are highly abstract but to a great extent executable. FASE is implemented in Lisp and allows the intermixing of Lisp code and FASE specifications.

To facilitate the testing of an operating system (and its specification), using FASE we have specified a Secure Resource Manager (SRM), a generic template of an operating system. The SRM specification can be specialized to a specification of a particular operating system; the SRM is quite general and handles most features of modern operating systems.

The second tool, called the PLANNER, is used to derive a sequence of operations that exhibits a security flaw, most often a covert channel for information flow. The PLANNER is based on classical methods of AI planning, specialized to achieve goals concerned with information flow. The PLANNER attempts to achieve a goal wherein one of a process' visible registers acquires a value different from its value in an initial state, the change being caused by a process at a level not less than or equal to that of the original process. The PLANNER uses a backwards-based search strategy. To make the search informed, the planner has heuristics that identify the operations that are most likely to be useful in achieving a goal. The planner can also be used to determine a measure of the channel capacity associated with a particular sequence of insecure operations through determining the number of possible values assignable to unbound variables in the plan – in essence, the uncertainty of the information flow. The PLANNER is implemented in Prolog.

Although these two tools can be used independently, they can be used together in two ways. For example, the FASE tool can be used to determine the initial and final states associated with a sequence of operations suspected to be a subsequence of an insecure sequence; then the PLANNER can determine a prefix and suffix to this sequence that achieves the unwanted flow. In addition, the FASE tool can be used to test partial plans produced by the PLANNER, the plans being extended by the user if the PLANNER becomes involved in long and fruitless searches.

The tools are demonstrated with respect to a simple operating system specification developed by Millen [Mil79]. This operating system provides user operations to read and write memory, and system operations to swap processes, and purge and claim memory blocks, the memory blocks accessible only indirectly through segment registers. Not surprisingly, the system contains a covert channel, the heart of which is the flow through the purging of a bloc' at one security level and claiming of the same block at a different level.

7.2 A generic specification for a secure resource manager

To provide a general tool for the rapid prototyping of secure systems, we have developed a general specification for a secure resource manager. To illustrate its usefulness, we will show in detail hov it specializes to a specification of the Millen operating system, and describe how the same process could be applied to obtain specifications of arbitrary complexity.

The specification method we use is that of final algebra specifications, as in [Kam83], [KJA83]. It is less familiar than those usually encountered, which include, for example, algebraic specifications, and specifications given by defining pre- and post-conditions on operations. Therefore, before giving details of the general specification and its specializations, we will review the method itself.

7.2.1 Writing final algebra specifications: the methodology

Despite its name, the final algebra specification method is not what is generally considered to be an algebraic method; rather, it is operational. It represents elements, essentially, by their observable behavior. Thus, a set S of objects of sort Elt is represented by a map from the carrier of Elt to Bool; this map determines for each e of sort Elt whether it is a member of S. If the operation for observing membership is named isin, then the map corresponding to S is the same as the map on elements e of Elt obtained by holding S constant in the expression isin(S, e).

In general, an element of an abstractly specified sort s will be represented by a tuple of maps (some of which may be 0-ary, and hence "maps" only by courtesy). Each element of the tuple will correspond to an operation on elements of sort s in analogy to the way a map from Elt to Bool corresponds to isin: it is obtained by holding the element of sort s constant while letting the other arguments of the operation vary. The operations corresponding to tuple components of elements of sort s form the *distinguishing set* for the sort s. Thus, the distinguishing set of the sort SetofElt consists of the single operation isin.

To give a final algebra specification of an abstract data type, one must first determine the operations in its distinguishing set. Any choice of distinguishing set operators leads to a representation of elements of the specified sort as elements of a product space; this representation then permits one to define the value returned by an operation as a tuple whenever that value is of the specified sort.

There is no fixed recipe for finding the distinguishing set, but there are some hueristics. In some cases, what certain components of this product space must be is clear. For example, it was clear to us in writing our generic specification that any SRM (secure resource manager) must have, among other things, a State component, a Scheduler component, and a SecPol (security policy) component. When this happens, it is clear that there will be corresponding distinguishing set operations that yield these components. In the case of an SRM, we have chosen to call them **stateofSRM, schedofSRM,** and **polofSRM.** In other cases, one thinks in the other direction, and starts from the operations to get the product space representation. A typical example of this is an abstract sort StackofElt, in which one realizes that two stacks behave the same if one gets the same things by reading their top and by popping them. This leads to a (recursive) representation of StackofElt as a product Elt × StackofElt.

As we have indicated, once one has determined the tuple representation of elements of the sort being specified, one can then define certain of the operations that return values of this sort — the constructor operations — by giving their results as tuples. All other operations must be defined in terms of these constructors, the distinguishing set operations, and the visible operations from other data type specifications, without the use of tuples. A corollary of this requirement is that tuples can be used to express elements of any sort only inside the specification of that sort.

We finish this subsection with a brief description of the notation and conventions used in our specifications; these are as in the FASE system as described in [KJA83]

Each specification begins with the name of the sort being specified, followed by a declaration of operations and arities; this is called the signature section. Any operations in the AUXILIARY OPERATIONS section are available only inside the given specification; all other operations are visible operations, and may be used by programs (including other specifications). Following the signature section is the list of those declared operations that constitute the distinguishing set.

Distinguishing set operations do not require explicit definitions. To compute the application of such an operation to a list of arguments, one extracts the (unique) argument of the specified sort, and applies the corresponding function in its tuple to the remaining arguments (a trivial application if the tuple function is 0-ary). All other operations, however, require definitions. Tuples in these definitions are indicated by square brackets, their elements separated by commas. 0-ary functions inside tuples are simply given as expressions. For functions with arguments, the notation "<a, b> |-> ···" can be read as "λ a,b. ···".

With regard to error values, there is an error value of each sort, together with a hidden element of the distinguishing set that detects it: thus, there is an **errSRM** and an **iserrSRM**, etc.. Function definitions are strict with respect to errors. Occasionally, the specifications will include definitions of the form **iserrType(opType(args))** => ···; these are translated as a prefix to the definition of **opType** which, when satisfied, causes **opType** to return an error.

The following notation is used for Boolean operations: "&" is used for and, "|" for or, "~" for not, and "=" for eqType for an appropriate choice of Type; these have the usual precedences.

Note that in the case of =, except for a primitive sort Type, it is not required that eqType be a true equality relation, but only that it be explicitly defined and have the appropriate arity (except that the system forces eqType to correctly detect equality to errType).

7.2.2 Structure of the generic SRM specification

For us, the term "secure resource manager" (abbreviated SRM) includes practically any operating system. In our attempt to construct a generic specification for an SRM, we wished to create a template which could be elaborated into specifications of many particular systems. We at first envisioned that such a template would consist of certain fixed, unchanging specifications, with subsorts being allowed variable specifications whose details would depend on a given application. Instead, it became clear to us that the specified sorts in the template would fall into three categories.

We will call specifications in the first category "fixed." These are essentially what we originally envisioned as fixed, but to make specialization to an application more convenient, we have loosened the requirement to permit addition of operations derived from other operations in the (combined) specification. For instance, our Millen example has an added derived operation updateobjState in the specification of State. In such specifications, the *reachable carrier* of the specified sort is fixed relative to those of the sorts in its representation. Hence, any properties proved from the general specification about all reachable elements of the specified sort, say by structural induction, will hold in any application.

The second category contains what we will call "fixed representation" specifications. These will specify a sort of fixed name, and will have a fixed list of distinguishing set operations of fixed arity. As a result, the *representation* of the specified sort will be fixed relative to the sorts involved in its representation. In general, such abstract types will have a fixed set of constants whose definitions are independent of the application, and sometimes, other operations with fixed definitions. Any properties that can be proved without structural induction about the elements of such a sort based on provable properties of the sorts in its representation will still hold in any application. Two sorts in our template which have fixed representation specifications are SRMop and SecPol. SRMop has a fixed null operation NOop, and a fixed "equality" operation eqSRMop (which is not a true equality, but behaves like one in this and any application in which no two operations can have the same name). SecPol has default policies noPol and recalcitrantPol.

The third category of specifications contains those having a fixed subsignature. In fixed subsignature specifications, at the very least, there must be a sort of the same name as the specified sort. In general, certain operations of a certain arity also must be present. Other than arity, there is no restriction on how the operations are defined. Such specifications correspond to the parameter part of parameterized specifications which occur in other specification methods (e.g., that of OBJ [GM82a], [GM84b]). Two of the sorts with such specifications in our template are Object and Request. Object has a fixed operation eqObject, which has fixed arity. Request has the fixed operations opofRequest and argsofRequest, also with fixed arities.

In any application, there will be subsidiary data type specifications, which we simply refer to as application specific. The sorts specified need not appear in all applications. Frequently, they are required in order to define sorts needed in the representation of sorts of fixed subsignature, or, possibly, of other application specific sorts. Among the application specific sorts present in our Millen example is Content; it exists only to define part of the representation of sort Object.

Instead of giving the SRM template specification in isolation, we will show its completion for the Millen example, and indicate which parts belong the the template. Table 1 gives a complete list of the sort specifications in our Millen specification, together with an indication of the kind of the specification (1 = fixed, 2 = fixed representation, 3 = fixed subsignature, 4 = application specific) and the abstract representation of each sort. Note that the alternate representations mentioned in the table as being the "real" representations of certain sorts show the structure of the reachable part of the sorts.

As we have indicated, the general template consists of parts of those specifications of kinds 1, 2, and 3. In table 2, we indicate for each of these specifications those operations in the Millen example that are present in the template. The *degree* to which they are "present" is determined as described earlier by the specification kind. For kind 2 specifications, we don't mention the distinguishing set operations, since they must always be present in the template.

As a further aid to understanding the template specification, figure 6.2.2 gives a graphic representation of the structures of sorts in the template having kind 1 and kind 2 specifications. The sort at each node in the tree is represented abstractly by the sorts of its children.

As indicated in table 1, the representations of ObjectSet and ProcessSet are different in the Millen specification from those in the generic specification. In general, for the purposes of rapid prototyping, it is sometimes convenient to replace a specification by an implementation, for a number of reasons. In our Millen example, we have actually given *implementations* of ObjectSet and ProcessSet, rather than their ideal specifications (in which sets would be represented as maps to Bool). These ideal specifications can, in fact, be given in the FASE language, but would require the use of quantifiers (the operations findinObjectSet and findinProcessSet would use the quantifier "some" — the existential operator that returns an instance). While many quantified expressions can, in fact, be executed in FASE [JK86], this execution tends to be slow. Moreover, for printing all members of an element of a "set" data type, as one frequently wants to do in an interactive driver for a prototype, it is much more convenient to represent the set as a list. Thus, we have chosen to do so in the Millen specification. Of course, there is, strictly speaking, a proof obligation attached — one must verify that the implementation is correct. In this case, the implementation is actually *partial* — one cannot create an arbitrary set of Objects, for example, but only one in which no two Objects have the same ObjectId. However, one can show that no SRMop causes one to attempt to create a disallowed ObjectSet, provided one starts with a State having a permissible one. Thus, in the context in which the Millen ObjectSet specification will be used, it can be considered an implementation of the generic ObjectSet ([KA84], [Arc88]).

As a matter of fact, the notion of partial implementation crops up frequently in the context of specializing the generic template to a specific example. One place, as we have indicated earlier, is the specification of "equality" operators that are not true equality operators, but are sufficient in context. Another place is in the use of a History component in the generic SRM. It is there to facilitate detection of information flow in cases where this cannot be observed simply by looking at the rest of the State. Provided that the History is never used to affect operations (i.e., in the definition of any SRMop), but only used to observe the system, there is no reason to implement it in implementing a system whose specification one has proved secure.

7.2.3 Specializing the template SRM specification

We believe that the usefulness of the template SRM specification lies in how it organizes one's thinking about the workings of an (almost) arbitrary secure system by factoring the system into its components that play conceptually different roles, and also in relieving one of having to specify certain high-level operations. Some of these components (e.g., the Scheduler) will play more im-

SORT	KIND	REPRESENTATION
SRM	1	SRMopSet × State × Scheduler × Interp × SecPol
State	1	ObjectSet × ProcessSet × RequestList × History
SRMopSet	1	SRMop → Bool
SRMop	2	Symbol × (State × ArgList → State)
Scheduler	2	(State → Request) × (State → RequestList)
Interp	2	Request → Request
SecPol	2	(State × Request → Bool) × (State × Request → Request)
ObjectSet	1	Object × ObjectSet [in generic template, Object → Bool]
ObjectPred	2	Object → Bool
ObjectId	3	Symbol × Symbol × Int × ProcessId [really {'X} + Int + Int × ProcessId]
Object	3	ObjectId × Content × Level × Int × Int
ProcessSet	1	Process × ProcessSet [in generic template, Process → Bool]
ProcessPred	2	Process → Bool
ProcessId	3	Symbol × Int [really {'System} + Int)]
Process	3	ProcessId × Bool × Level × [Int → Object] × [Int → Object]
RequestList	1	Request × RequestList
Request	3	SRMop × ArgList × ProcessId
History	3	State × RequestList
ArgList	1	Arg × ArgList
Arg	3	Object × Process × Int [really Object + Process + Int]
Content	4	Symbol × Int × (Int → Int) [really Int + (Int → Int)]
Level	4	Bool × Symbol [really {'syshi} + Symbol]
RegAssn	4	Int → Int
ComputeFn	4	Symbol × (RegAssn → Int)

Table 1: Millen specification overview

SORT	KIND	GENERIC OPERATIONS
SRM	1	(all)
SRMopSet	1	(all but mitreSRMopSet)
SRMop	2	eqSRMop , NOop
State	1	(all but updateobjState)
Scheduler	2	null- and trivScheduler
Interp	2	trivInterp
SecPol	2	no- and recalcitrantPol
ObjectSet	1	(all)
ObjectPred	2	trivObjectPred
ObjectId	3	eqObjectId and precedesObjectId
Object	3	idofObject
ProcessSet	1	(all)
ProcessPred	2	trivProcessPred
ProcessId	3	eqProcessId and precedesProcessId
Process	3	idofProcess
RequestList	1	(all)
Request	3	opof-, argsof-, procidof-, and nullRequest
History	3	init- and appendHistory
ArgList	1	(all)
Arg	3	objidof-, procidof- , objidto- , and procidtoArg

All distinguishing set operations of kind 2 and kind 1 specifications are also generic. For operations in kind 3 specifications, only the arity is generic.

Table 2: Generic template

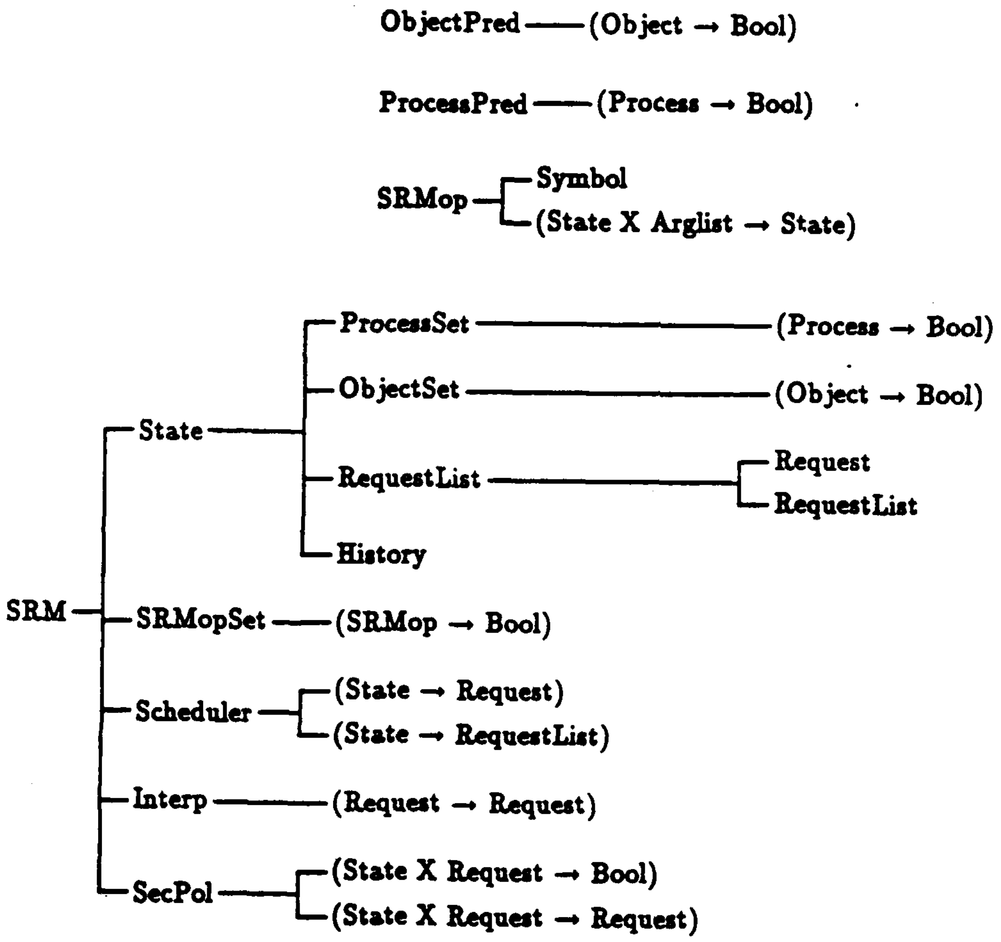

Figure 1: Overview of the SRM template specification

portant roles in some systems than others, but for any relatively complex system, most of them will be non-trivial. The template has one further advantage: when it is completed (and in many cases, only partially completed), it can be used as a rapid prototype of a system (or part of it).

Obtaining the Millen specification: As a concrete illustration of how the template SRM specification can be useful, we will describe here the major decisions we needed to make in specializing it to a specification of the Millen system. In the process of specialization, a number of omissions in the description of the Millen system in [Mil79] became clear, and we had to make some decisions about how to handle them. We tried to make these decisions so as to allow as much flexibility as possible in the system objects one might wish to create for testing purposes.

In specializing the template, one first has to make decisions about how to complete the kind 2 specifications. We will take these in order.

The first such specification we encounter is SRMop, which determines what can be in an SRMopSet. Here, we soon discovered that the list of operations in [Mil79] needed to be expanded. For example, a user, not being privileged, cannot directly execute certain commands such as **Swap** and **Purge** (SWAP and PURGE in [Mil79]). The user is also unable to do a Get (GET, in [Mil79]), not having access to information about free memory blocks. Thus we have added user operations **uSleep**, **uRelease**, and **uGet** which, in effect, ask the system to request the corresponding privileged operations. We similarly decided that the privileged operation **Raise** will normally be invoked as the indirect result of certain user calls to **uGet**, and that the privileged operations **ReadX** and **WriteX** (READ and WRITE the X register in [Mil79] — created there to illustrate a possible way to introduce a security flaw, which we wanted to be able to exhibit) similarly required corresponding user calls to **uReadX** and **uWriteX**, respectively. There are also problems with the nominally purely user operations **uFetch**, **uStore**, and **uCompute** (FETCH, STORE, and $COMPUTE_k$ in [Mil79]) since a process would have to pass itself (or its name) as an argument so that the correct memory block and registers to reference can be determined. Thus we also added correspc nding system (although not privileged) operations **Fetch**, **Store**, and **Compute**.

Our template specification is structured to permit the definitions of operations to be factored into what the operations do, and the pre-conditions under which the operations will be allowed. In our Millen example, we isolate the pre-conditions into the SecPol factor of the SRM, particularly since their enforcement is the system's attempt to enforce the "true" security policy. Thus, our operation definitions in the SRMop specification are free of the clutter of the pre-conditions.

The actual definitions of the operations force a number of implementation decisions concerning what will be the elements of Arg and what some of the operations on Objects and Processes must be. E.g there must be Int and ObjectId arguments, and there must be an operation by which to observe the Content of an Object.

The next kind 2 specifications we encounter are ObjectPred and ProcessPred. We see that we cannot add constructors to these data types without knowing more about the structure of Objects and Processes. Since the Millen specification refers to Processes and memory blocks (which will be one kind of Object) by number, and registers (another kind of Object) by name or number, we decide that both Processes and Objects must have Ids, and that there must be predicates to decide when a given Object or Process has a given Id. Thus we are also led to create additional data types called ObjectId and ProcessId, having equality operators. Having done this, we can now defined the desired predicates that test equality of an Id to a given one.

We next encounter the components of the State. The only component not of kind 1 is the History (kind 3). Though we are mainly discussing only kind 2 specifications, it is worth looking more closely at History in order to understand the role it plays. In general, a History object could be less detailed than we make it; since it is there for flow-detection purposes, it is enough to record whatever information is needed for this purpose. We have chosen to keep essentially all the information needed to reconstruct all changes of system State. Thus, a History is a State and a RequestList; **initHistory** saves the initial State, and **appendHistory** is used to successively save requests.

There being no explicit mention of scheduling in [Mil79], we have taken the Scheduler to be **trivScheduler**.

In general, we see the Interp component of an SRM — the Request interpreter — to be a mechanism by which the system can trap user requests and interpret them in terms of commands only known to the system. This seems to us to be the natural way to allow for operations such as TRAP [Mil79]. The specific element of sort Interp that needs to be included in any Millen system is **mitreInterp**, which changes user Requests into lists of Requests involving "internal" operations; thus, a user **uGet** request becomes one of the sequences **<Get>** or **<Raise, Get>** of system requests.

In any SRM, the SecPol component will be the "effective security policy" — actually, the filter on Requests that attempts to enforce the true security policy — of the system. In operation, it tests a Request for whether the pre-conditions associated with its SRMop are satisfied. In the Millen example, these pre-conditions actually simply implement an access policy; in general, they could make much more complex tests that might even examine the History component of the State. However, the way we have structured **mitreSecPol** is typical of the way we would expect the SecPol of any SRM to be structured: separate SecPols for the operations are combined into the overall SecPol. The tests made by the SecPol on operations force some further decisions. For example, the fact that the SecPol wants to know the Level of the Process requesting certain operations means that such information must be obtainable from a Request. We have chosen to give each Request a ProcessId component, although it would be sufficient to just use the Level of the corresponding Process. Other decisions forced by the definition of SecPol are that each Object must have a Level, and Int representing its access count, and so on.

Obtaining a specification of your pet system: The procedure we used above for the Millen example can be followed to yield a specification of almost any system. Basically, one completes the kind 2 specifications; in the process, one finds out the structure of the sorts having kind 3 specifications, and the operations needed in these specifications. In the process of elaborating the kind 3 specifications, one creates the needed kind 4 specifications.

More particularly, one starts by determining the SRMops — the operations provided in the system. Any restrictions on the permissiblity of these operations can be factored out and made part of the SecPol. Which constructors to add to ObjectPred and ProcessPred depend mainly on the needs of the SRMops, since these operations are the ones that ultimately select Objects or Processes in order to change (or, in our applicative system, replace) them. There may be cases, however, in which SecPol, for example, needs particular ObjectPreds or ProcessPreds. The basic properties of an Object or Process can usually be determined from the high-level description of the system. Further details that distinguish individual Objects or Processes are determined by what various SRMops do, what the SecPol needs to test for, and so on. The needs of the Scheduler and the SecPol, primarily, determine what, besides an SRMop and an ArgList, distinguishes one Request from another. In the Millen example, a Request had an associated ProcessId; in systems with a nontrivial Scheduler, Requests may need to have an associated priority instead, or as well.

Whether there will be a nontrivial History component in the State depends on what information flows one wants to test for, and how much information about the previous states of the system one needs in order to do so.

7.3 Running the specification: the Millen example

As mentioned in the previous section, we attempted to make our Millen specification as flexible as possible; the reason for this is that one can then initialize the system to any permissible configuration desired: any number of user processes (numbered starting from 0), any number of registers for any given user process, any number of memory blocks, etc.. Having done this, we *assume* that the system will be initialized in a permissible configuration; we also assume that the driver accepts requests only if they could occur in the current state of the system. These conditions could be enforced inside the specification, but it would be cumbersome to do so, and we have chosen not to.

In the case of the Millen example, the SRMopSet of the initial configuration must be **mitreSRMopSet**. Also, the State of an initial configuration will have only one swapped-in Process, and no memory block Objects that are referenced by more than one Process; moreover, its ProcessSet must consist of the **systemProcess** and zero or more user Processes with ProcessIds numbered consecutively from 0. Finally, the Scheduler of any initial configuration must be **trivScheduler**, its Interp must be **mitreInterp**, and its SecPol must be **mitreSecPol**.

Once an initial configuration is assigned to a Lisp variable, updates to this configuration must be done by **stepSRM**, which will never violate the conventions of the system, and **newreqSRM**. The driver should accept a new Request only if it comes from a Process of which **inofProcess** is true (meaning the Process is swapped in).

Thus, to complete the rapid prototype of the Millen system one should write a driver that 1) creates a permissible initial configuration of the system upon reading input that controls the variables in such a configuration (such as how many Processes there are, what memory block Objects of what sizes there are, etc.), 2) takes as input instructions either to call **stepSRM** or to call **newreqSRM** with a new Request, and 3) checks that incoming Requests could in fact arise at the point they are given as input. As we will explain in the next section, the FASE system provides a means to make the input instructions convenient to give interactively without the use of the long function names appearing in the specification, and makes it convenient to write a driver for the system in Lisp.

Although we have not yet written a driver for the Millen system, we have written drivers for a number of other FASE specifications, and anticipate no difficulties in doing so.

7.4 Advantages of the FASE system

The FASE system [KJA83] is a system for executing final algebra specifications [Kam83] of abstract data types. We find its use convenient for several reasons. Perhaps the most important reason is that the specifications are executable whenever they do not involve quantifiers. Even some specifications with quantifiers are executable, and the system is able to identify a class of such specifications which it guarantees it can execute (although, for efficiency reasons, it is best to avoid quantification in a rapid prototype).

There are, of course, other systems for executing specifications, such as OBJ and Affirm [GHM78]. These mostly involve algebraic (equational) specifications. While such specifications are sometimes straightforward to write, knowing when one has enough equations (or so many as to lead to inconsistency) can be a problem. By contrast, we find final algebra specifications to be among the easiest specifications to write. One does not have the problem of proving sufficient completeness and consistency of equations. Once one has determined the abstract representations as tuples of elements of various sorts, the definitions of their associated operations are usually straightforward to derive from verbal descriptions.

In addition, the FASE system is integrated with Lisp, so that one can easily write Lisp drivers to exercise, and hence test, one's specifications interactively. One can easily incorporate into a driver print routines that call operations from the specification in order to display, for example, a system state represented in some convenient fashion by its observable behavior.

Specification and testing are further much facilitated by a provision for almost arbitrary user-defined syntax. With an appropriate grammar, one can easily handle the formation of sets and sequences, as well as coercions, and avoid typing in lengthy (though possibly, as in our example, descriptive) function names. User expressions are delimited in Lisp by exclamation points, and in specifications by underscores. Thus, in our Millen example, we can provide a grammar that will allow us to say at the lisp level:

    !user 0 calls uReadX on 2, 5 in M!

to abbreviate

```
  (newreqSRM (mkRequest (uReadX)
                        (appendArgList (inttoArg 5)
                                       (appendArgList (inttoArg 2) nilArgList))
                        (mkuserProcessId 0))
             M)
```

(note that M in both the above can be a Lisp variable whose value is of sort SRM). While overloading of operators in the specifications per se is not allowed, it is feasible to a great extent in the grammars, since the parser is able to do type-checking of all but variables.

7.5 The PLANNER

This subsection discusses an approach to formal testing of an operating system with respect to a security policy. The goal of the work is as follows: Given specifications for the operations of an operating system, determine a sequence of operations that effect a disallowed information flow between a pair of processes, assuming that such a flow exists.

This concept is implemented in a prototype system called the PLANNER; it is implemented in Prolog, but uses its own searching code to make the search more informed that the basic Prolog resolution method. It is based on an extension of classical AI planning methods [Nil80].

7.5.1 Detecting flows

Planning (of actions) is concerned with the automatic derivation of a sequence of operations (called the plan) that achieve a goal. In classical planning, the goal is expressed in terms of two states: an initial state and a final state, and the sequence of operations is to achieve the final state given the initial state. Planning has been used to derive plans for moving robot vehicles so as to achieve a goal such as "refueling a tank."

The basic approach to planning involves searching combined with unification. The searching can be forward driven, backward driven, or mixed. The backward approach is as follows. The goal and initial states are described in terms of predicates on constants and variables. The operations are specified in terms of preconditions and postconditions, each expressed in terms of variables and constants. Variables can be bound during the planning process.

If the goal state unifies with the initial state, the plan is complete. The collection of operations is scanned to locate an operation with a postcondition that unifies with the goal state. This operation might have preconditions; for simplicity, assume there is only a single precondition. The variables of the precondition might be bound to values as a result of the unification of the goal state with the postconditions. Then the postcondition, with the variables appropriately bound, becomes the new goal.

Complications arise when the goal is a conjunction of terms; a conjunction of terms can arise at any point in the planning process; for example, when an operation is used that has more than one precondition. Significant research has been devoted to this problem, none of the solutions avoiding possible long and fruitless searches. One technique is as follows. When encountering a conjunction of goals, pick one to work on; it might be possible to identify the most difficult of the goals assuming a measure of difficulty is available. Identify an operation with a postcondition that unifies with this selected goal. Then the other goals are regressed backwards through this operation. If the regressed goals conflict with the preconditions, than the choice of operation or the goal selected to pursue was flawed, and the procedure backtracks to select a different operation or a different goal. The planning process continues until the collection of goals unifies with the initial state.

Our security-based PLANNER is similar, but introduces some new concepts to cope with information flow.

The goal state expresses an insecure flow, and is based on the concept of "interference" of processes in an operating system. The concept of flow and of a a model "process" is wired-into the PLANNER to reduce the search space. A more general planner would take an arbitrary security policy and model of processes, but at the expense of search time.

Assume a goal state in which process P_i is running and has read access to a collection of registers; it is through these registers that a process acquires information. Assume an initial state in which P_i is also running. The contents of a register in the initial state and final state are to differ. In order to exhibit information flow, the difference is to be caused by an operation by a process P_j whose security level is not less than or equal to that of P_i; for simplicity the PLANNER assumes a category model of security levels: processes at different levels are forbidden to communicate with each other. Still working backwards, the PLANNER will have two types of goals to achieve:

1. Goals that involve information flow between processes: These goals are achieved by two operations, one that produces a change in state (e.g., to a register) and the other that causes a swap of processes. Assuming no single operation satisfies both of these needs, the final plan must contain these two operations but not necessarily contiguously.

2. "Ordinary" goals as faced by the classical planner. These goals will arise from preconditions of operations that the planner conjectures will become part of the plan.

These concepts are discussed in the next section with reference to the PLANNER deriving a plan for insecure flow in the Millen operating system.

7.5.2 Detailed description

The current implementation of the PLANNER consists of a collection of rules written in Prolog. Prolog was selected in order to take advantage of backtracking and resolution. Use of backtracking provides a method whereby all possible plans may be considered, and use of resolution permits planning variables to be left unbound as long as possible. The PLANNER contains four types of rules: *planning rules*, *difference computation rules*, *operation rules*, and *architecture rules*.

Planning rules: Planning rules are used to manipulate plans. There are two types: those that eliminate "unnecessary" goals, and those that select the subgoal which will be achieved next. Unnecessary goals are goals that have already been achieved (as a side effect of solving other goals), or goals that do not cause any real change in state. For example, the system may have two goals: causing user A's register i (denoted $R(A,i)$) to contain value X (Goal 1), and causing user A to become the currently active user (Goal 2). Suppose that in the initial state, user B is active and user A is blocked, and that only active processes may modify their registers. Further suppose that the system chooses to work on Goal 1 first, and achieves it via the following plan:

1. Make user A active so that X may be written into $R(A,i)$

2. Write X into $R(A,i)$

Clearly, Step 1 of the plan also achieves Goal 2, so that goal may be eliminated from the system. Step 1 and Step 2 may be considered *subgoals* of Goal 1.

The second type of planning rules consists of those rules that determine which goal the PLANNER will try to achieve first. In theory, the goals are achievable in any order: if the PLANNER determines that it is impossible to achieve all goals at some point in the computation, then it backtracks and tries them in a different order. However, in practice this does not always work with the current version of the PLANNER, since it does not recognize infinite loops in planning sequences. The PLANNER may attempt to achieve a sequence of goals where the solution to the first goal "undoes" the solution to the last goal. This may be seen clearly in the following example.

Consider a system containing Goal 1 (described earlier) and Goal 2′: make user B active. This time, user A is active initially. The following sequence will loop infinitely:

1. Make user B active (to achieve Goal 2′).

2. Make user A active (to achieve Subgoal 1 of Goal 1). This, however, undoes Goal 2′.

3. Make user B active (to achieve Goal 2′)

⋮

Both subgoals could have been achieved if the PLANNER had chosen to complete both of Goal 1's subgoals before attempting to achieve Goal 2'. In order to avoid this type of looping, the current version of the PLANNER uses two heuristics: complete all the subgoals of a goal at one time, and complete the most complicated goals first (these are the goals most likely to undo other goals). The complexity of a rule is measured by counting the number of subgoals it contains. These two heuristics are not sufficient to prevent all infinite loops, so future implementations of the PLANNER are expected to contain some form of loop detection and escape. Possible methods of achieving this are discussed in the conclusion.

Difference rules: The PLANNER operates by determining the differences between an initial state and a final state, and then attempting to eliminate these differences. The PLANNER computes differences by comparing state components in the initial and final states. For example, if process A's register $R(A,i)$ contained the value X in the initial state and the value Y in the final state, then the PLANNER would add a *register difference* of [[A, i, X], [A, i, Y]] to the current goal list. Every state component has its own collection of difference rules, since these depend upon the representation of that component.

Operation rules: The operation rules embody the semantics of system operations. Each operation the system provides is described in terms of preconditions and postconditions. For example, it was mentioned earlier that a precondition of writing into a given process' register is the requirement that the process be currently active. A postcondition of writing the value X into register R is that register R now contains the value X.

Figure 2 is a high-level description of the *Fetch* operation defined in the Millen system. Operationally, *Fetch* reads a value from a specified memory location and places it into a local register. *Fetch* is called with four arguments: the user doing the fetching, the register into which the new value will be written, and the virtual address of the memory block and physical offset within the memory block of the new value. The with clause in the definition describes the system state that will result from applying *Fetch*. The original system state, which is described in the full Millen system definition given in subsection 6.8, is *Memory, Register, Mmu, CurrentUid, WaitingUid* ; thus, *Fetch* is defined to modify only the *Register* portion of the system state.

The **produces** clause describes the actual difference of the operation: the value of register Rn for User will be changed from its previous value (DontCare) to the new value (New). The value DontCare indicates that the actual value that was originally stored in register Rn does not affect the validity of the *Fetch* operation. The by clause states that this modification is produced by the architecture rule *doStoreReg*; this rule is the one that the PLANNER uses to modify its internal version of state. Architecture rules are described in more detail in a later section.

The **with** clause introduces the preconditions that must hold before *Fetch* may actually be applied. In this example, the three preconditions state that (1) there is a physical memory location which contains the value New, (2) User has access to that location (i.e., User has a virtual address corresponding the the physical block address), and (3) User is currently active (i.e., the user id of the currently active user—CurrentUid—is identical to User). If any of these preconditions does not hold at a stage in the plan where the PLANNER wishes to apply *Fetch*, then the PLANNER will make them subgoals and attempt to achieve them first. If these subgoals cannot be achieved, then the PLANNER cannot add *Fetch* to the plan.

Associated with each of the preconditions there is a **trigger** and a **difference** clause. These clauses are used to indicate ways in which the preconditions can be satisfied if they are not currently true. The **trigger** clause specifies the particular component of state that must be modified. For example, if there is a memory location containing the desired information but the current user cannot access it (i.e., *memMap* fails), then the PLANNER triggers a change in current user *Uid*. The **difference** clauses describes the exact difference that must be achieved in order for the preconditior to be satisfied.

Operation rules are used by the system to create steps in the plan. As mentioned earlier, the PLANNER operates by trying to achieve goals which are stated as differences between states. The PLANNER starts from a given final state and works backward to satisfy the given initial state. (Recall that information flow to a user A is defined here as information that was not available to A in the initial state but is available to A in the final state. For the Millen system, information is considered to be available to a user when it is in one of that user's registers.) When the PLANNER is working on a particular goal, it searches through the system operations until it finds one which has a postcondition containing the desired difference. In our example, the PLANNER would select *Fetch* to achieve the goal of modifying a register's contents. In order to use an operation, the PLANNER must ensure that the operation preconditions hold at the point in the plan where it is to be used. These preconditions are then subgoals. Note that preconditions do not themselves contain operations, but only architecture rules (described later).

```
operation  Fetch(User, Rn, Mvirtual, Mn)  with
state   Memory, RegisterOut, Mmu, CurrentUid, WaitingUid
produces  Register:
   [[ User, Rn, DontCare] . [ User, Rn, New]]
        by doStoreReg( Register, User, Rn, Newval, RegisterOut)
with
    precond  fetchMemory(Memory, Mphys, Mn, New)
                  trigger Memory
                  difference [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
               end ;
    precond  memMap(Mmu, User, Mvirtual, Mphys)
                  trigger Uid
                  difference [[Uid], [WaitingUid]]
               end ;
    precond  equal(CurrentUid, User)
                  trigger Uid
                  difference [[Uid], [WaitingUid]]
               end
```

Figure 2: High-level description of Fetch: $R(i) \leftarrow Mem(j, k)$

Some complications can arise when an operation has more than one postcondition, since all of the postconditions of an operation must hold at the stage in the plan following the operation's application. If one of the operation's postconditions does not hold, it is necessary to insert a *subplan* between these stages. The initial state of the subplan corresponds to the state that holds after the current operation is applied (i.e., operation postconditions hold), and the goal state is the one to which the operation is to be prepended (see Figure 3).

Architecture rules: The PLANNER's architecture rules are used internally by the PLANNER to modify its view of the system state. They may also be considered predicates on the state that are instantiated or revoked depending upon the operation being applied. For example, in a system containing a collection of registers, there must be architectural rules that will allow the PLANNER to observe and modify register values within the current state. Alternately, one may consider the system to contain predicates such as "Register I of user A has value X" and "Register I of user A is modified to contain value X." Architectural rules are used within the plan to describe preconditions and postconditions.

High-level System Description: As was alluded to in an earlier section, the PLANNER is written in Prolog. However, a high-level translator has been developed so that it is not necessary for the user to describe entire systems directly using Prolog code. This translator is capable of generating most of the planning rules and operation rules, and some of the difference computation rules. It is still necessary for the user to write Prolog code describing most of the difference computation rules and the architecture rules. Fortunately, there are not many of these and they tend to be easy to formulate. The full high-level description of the Millen system is given in the Extended Examples Section 6.8.

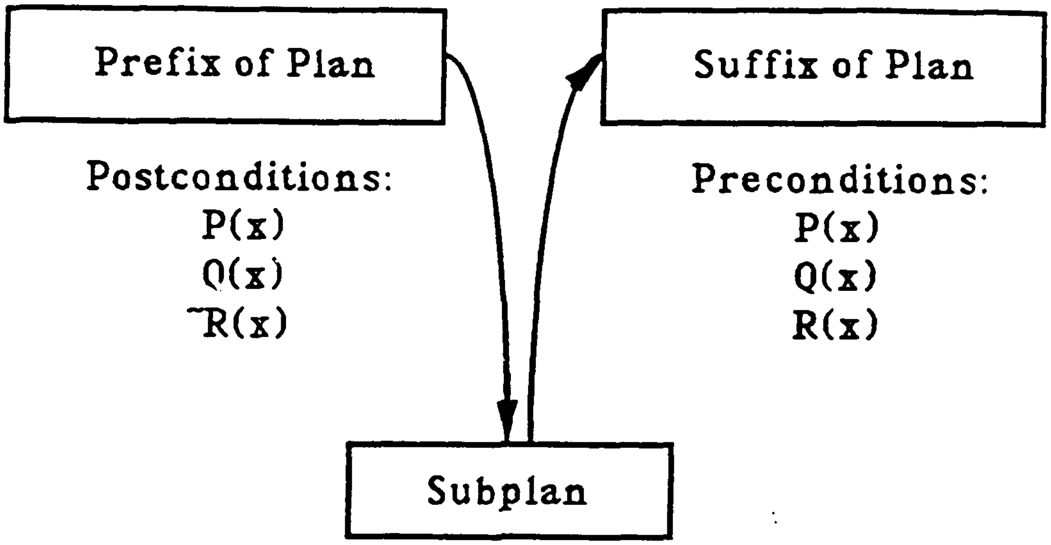

Figure 3: Insertion of a subplan

7.5.3 Planner input vs. specification input

There are many similarities between the high-level description of the planner and the generic specification of the system. Both systems contain an internal state and operation descriptions. The preconditions of the PLANNER's operation rules correspond to the security policy described in the generic specification. The postconditions of the PLANNER's operation rules correspond to the functional description of the operations in the generic specification. Further, plans found by the PLANNER are executable by the generic specification (see Figure 4). At present, the PLANNER does not contain all of the information in the full specification: for example, there is no explicit history, interpreter or scheduling policy. The PLANNER does, however, contain history information implicitly in the plan that it produces. An interpreter is not necessary for the PLANNER, since it uses the expanded version of user operations to generate plans. However, it is easy to find examples where the scheduling policy of a system can itself be either a source of flow violations, or may eliminate certain types of flow. The only notion of scheduling available to the PLANNER lies in its ordering of goals, and the preconditions associated with operations.

7.5.4 Examples

An obvious example of insecure flow: This section describes the path which the PLANNER follows to come up with a very simple example of information flow. The operations available to the PLANNER include all of those used by the general Millen operating system, plus two that have been added to induce flow: *ReadX*, *WriteX*. The Millen operating system has been expanded to include a special system register $\mathcal{X}$, which any user may read or modify using the new operations.

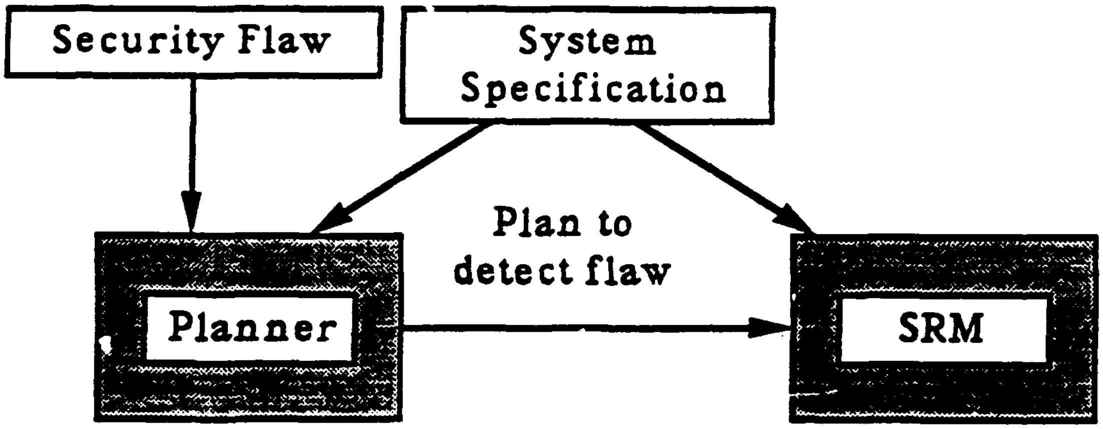

Figure 4: Planner and SRM specification

Clearly, flow between users may occur via this special register. To further simplify the example, the system is assumed to contain only two users, each with exclusive access to two blocks of memory. At each step in the plan, the system state and the plan state will be given.

The PLANNER is given the goal of finding a plan whereby one user obtains information originally contained in the second user's memory. This is stated by defining an initial state where User 1's block does not contain User 2's information, and a final state where User 1's block does contain User 2's information. Recall that we are working backwards from the final state toward the initial state.

Initial state:

$$\left(\begin{array}{lll} R(User1, i) = R_1 & R(User2, i) = R_2 & Mem(0, k) = M_0 \\ Mmu(User1, m) = 0 & Mmu(User2, j) = 1 & Mem(1, k) = Secret \\ \mathcal{X} = DontCare & & \\ \text{Current user} = 2 & & \end{array} \right)$$

Final state:

$$\left(\begin{array}{lll} R(User1, i) = Secret & R(User2, i) = R_2 & Mem(0, k) = M_0 \\ Mmu(User1, m) = 0 & Mmu(User2, j) = 1 & Mem(1, k) = Secret \\ \mathcal{X} = DontCare & & \\ \text{Current user} = 1 & & \end{array} \right)$$

1. $Mem(Mmu(User1, m), k)_0 \neq Mem(Mmu(User2, j), k)_0 \neq 0$

   $Mem(Mmu(User1, m), k)_t = Mem(Mmu(User2, j), k)_t \neq 0$

   To shorten the plan, use User2 as the active process initially and User1 as the active process in the final state. (The PLANNER will produce a correct plan even if this is not the case.)

Possible Plans for Goal 1:

- $Store(i,m,k)$ by User1, with $R(User1,i) = Mem(Mmu(User2,j),k)_0$

- $Purge(Mmu(User1,m))$

There are two possible plans, since two operations contain a postcondition with a modified memory. However, *Purge* can only write 0 into memory and we have excluded 0 as a possible secret message.

Choose plan *Store(i,m,k)*.

$$\overbrace{\ldots\ Store(i,m,k)}^{User1}$$

This sets up a new goal:

2. $R(User1,i) = Mem(Mmu(User2,j),k)_0$ Possible Plans for Goal 2:

   - $Fetch(i,j,k)$ with t=0
   - $ReadX(i)$ with $\mathcal{X} = Mem(Mmu(User2,j),k)_0$

   (Either possibility will produce a valid plan here. If plan *Fetch* is chosen, the PLANNER will attempt to satisfy the precondition that there must be a register within User1's register set containing the value in User2's block. This produces a longer plan, so for illustration, *ReadX* will be chosen.)

   Choose plan *ReadX(i)*.

   $$\overbrace{\ldots\ ReadX(i)\ Store(i,m,k)}^{User1}$$

   This sets up a new goal:

3. $\mathcal{X} = Mem(Mmu(User2,j),k)_0$ Only a *WriteX* can cause $\mathcal{X}$ to contain the desired value. The problem is that the current process cannot do the write, since it does not have User2's information. Thus, User2 must have done the write into $\mathcal{X}$ earlier. This sets up a subgoal that must be achieved before goal 3 can be achieved.

4. Current process = $User2$

   Possible Plans for Goal 4:

   - $Swap$

   Only one possible plan, so choose plan *Swap* and propagate the goal 3.

   $$\overbrace{\ldots\ Swap}^{User2}\overbrace{ReadX(i)\ Store(i,m,k)}^{User1}$$

   There is now a plan for goal 3 which may be applied: Possible Plans for Goal 3:

- *WriteX*

Choose plan *WriteX(i)*. The plan now becomes:

$$\overbrace{\ldots\ WriteX(i)\ Swap}^{User2}\ \overbrace{ReadX(i)\ Store(i,m,k)}^{User1}$$

This sets up a final goal:

5. $R'(User2, i) = Mem(Mmu(j), k)_0$

   This may be achieved directly by plan *Fetch(i, j, k)*.

Thus, the resultant plan is:

$$\overbrace{Fetch(i,j,k)\ WriteX(i)\ Swap}^{User2}\ \overbrace{ReadX(i)\ Store(i,m,k)}^{User1}$$

A more subtle example of insecure flow: The previous example described an obvious example of flow that was easily discovered by the PLANNER. In that example, information was directly transferred from one user to another. This section describes a more complex example of information flow where information is transferred indirectly. Here, one user will observe one of two possible results, depending upon the actions of a second user. The operations used in creating this plan are those described in subsection 6.6, excluding both the *Compute* operations and the two operations that modify register X.

Some new notation will be introduced at this point. Objects and operation names may be subscripted with plan stages: $R_t(U, i), Store_t(i, j, k)$. Plan goals such as $=, \neq$ may also be subscripted with a user level to indicate that they may only be satisfied by a user operating at that particular level. For example, $R_t(U, i) \neq_y R_0(U, i)$ indicates that user U's register i at plan stage t must not have the same value that it did at plan stage *0*; further, the change in value must have been caused by a user operating at level y.

The following describes the reasoning the PLANNER could use to locate such a security flaw. The relevant part of the system state is shown in this format:

$$\begin{pmatrix} Mmu(X,j) = b_1 & Mmu(Y,j) = b_2 & Mem(b_1,k) = M \\ R(X,i) = R_1 & R(Y,i) = R_2 & \end{pmatrix}$$

1. $R_t(X, i) \neq_y R_0(X, i)$

   The initial goal of the system is to find a plan where user X has a different value in one of its registers if user Y takes a particular action (note that this violates the restrictiveness property). Only user X can modify $R_t(X, i)$. Possible Plans for Goal 1:

   - $Fetch(i, j, k)$ by user X

   Thus, goal 1 will be achieved when the following goal is achieved:

2. $Mem_{t-1}(Mmu(X, j), i) \neq_y R_0(X, i)$

   To fulfill this goal, we must ensure that something other than $R_0(X, i)$ is written into the memory block b_1, where $b_1 = Mmu_{t-1}(X, j)$. Since this inequality is restricted to be caused by user Y, user Y must have had access to the block at some stage in the plan. The current plan looks like this:

$$\overbrace{\ldots Swap}^{User X}\overbrace{\ldots Swap}^{User Y}\overbrace{\ldots Fetch(i,j,k)}^{User X}$$

User Y cannot modify memory unless it has a pointer to it in its Mmu. Both users cannot point to memory simultaneously, so user X must have regained access to the memory block after Y. This sets up a new goal:

3. $Mmu_{t-2}(X,j) = b_1$

   Possible Plans for Goal 3:

   - $Get(b_1,j)$

   The preconditions to this operation require block b_1 to be active, have no other user accessing it, and have the proper security level, leading to three new goals:

4. b_1 is active

5. b_1 is unused

6. b_1 has level X

   Possible Plans for Goal 6:

   - $Raise(b_1)$

   This plan also achieves goal 4. The goals which still need to be achieved are: 2, 5. Possible Plans for Goal 2:

   - $Purge(b_1)$

   This portion of the plan should take place before user Y performs *Swap*, since user X no longer has access to b_1 here. The current plan and system state at $t-4$ are:

$$\overbrace{\ldots Swap}^{User X}\overbrace{\ldots Purge_{t-4}(b_1)\ Swap}^{User Y}\overbrace{\ldots Raise(j)\ Get(b_1,j)\ Fetch(i,j,k)}^{User X}$$

$$\begin{pmatrix} Mmu(X,j) = b_n & Mmu(Y,j) = b_1 & Mem(b_1,k) = 0 \\ R(X,i) = R_1 & R(Y,i) = R_2 & \end{pmatrix}$$

Note that goal 2 has been achieved, since b_1 now contains 0 after the *Purge*. It is necessary to achieve the preconditions of *Purge* (same as goal 5). This goal is achieved as a side effect of *Get*, since that is the only operation which makes blocks available to the system. Additionally, since we assumed that user X originally had b_1, it makes sense to place this plan before user X's first *Swap*. Assuming that User X initialized $R(i)$ to something other than 0, the plan and initial system state become:

$$\overbrace{\text{Initialize}\ Get(b_3,j)\ Swap}^{User X}\ \overbrace{Purge_{t-4}(b_1)\ Swap}^{User Y}\ \overbrace{Raise(j)\ Get(b_1,j)\ Fetch(i,j,k)}^{User X}$$

$$\begin{pmatrix} Mmu(X,j) = b_1 & Mmu(Y,j) = b_n & Mem(b_1,k) = X \neq 0 \\ R(X,i) = R_1 & R(Y,i) = R_2 & \end{pmatrix}$$

A necessary component of this plan is that the system is in collusion with the users, since a user cannot directly name blocks or force the system to obtain a particular block.

7.6 Conclusions and future directions

Although verification of secure operating systems specifications is important and, hopefully, feasible, testing is also important and less expensive than verification at discovering flaws. Furthermore, testing can be used to discover properties of a system aside from those related to security. Accordingly, we have developed two testing tools as a first step towards an automated testing methodology for secure systems.

The first tool is based on the FASE executable specification language. FASE has the advantages of an algebraic language (abstraction, parameterization, and error handling) but has the intuitive feel of an operational language. The FASE tool is used to execute specifications with real values. To facilitate the testing of specifications for a secure operating system, we have written a specification for a generic secure resource manager (SRM) in FASE, the specifications being a template for a large class of secure operating systems. Although it remains to be determined that the SRM is a suitable template for many of the operating systems of interest, our initial experience has convinced us that generic specifications are possible for an operating system and for its security policies; at the abstract level, operating systems have much in common. Although the generic feature is lost at lower levels of abstraction, FASE can be used to mix specifications with decisions expressed imperatively.

The second tool, the PLANNER, is used to derive a sequence of operations that exhibits a security flaw, usually a covert channel. It is based on classical planning techniques, but extended to handle goals that express a flow of information between processes. Different from the classical planners, PLANNER is specialized to the domain of security flow. This specialization has led to a system whose performance is better than that achievable with a general planner.

Both the SRM and the PLANNER have been evaluated on small examples. The main example is an operating system described by Millen, whose purpose is to demonstrate security verification. Its main operating system features are process switching including swapping, and reclaiming of memory blocks. Although clearly a toy, it has the basic features of a multiprogrammed operating system with dynamic allocation of main memory.

Clearly, it is essential to consider larger systems. A suitable next step would be a system based on MINIX, an operating system with the system calls of UNIX, but a vastly simpler kernel.

We have several improvements in mind for FASE. Currently, many uses of quantifiers lead to specifications that are not executable. When FASE is unable to execute a quantified expression, the user has no choice but to try a different specification. We are considering a feature whereby a user can define an implementation for a quantified expression (such as through an iterator over a type). Furthermore, FASE can execute only specifications with respect to real values of input. Extensions to symbolic evaluation would improve its utility in testing, but at the cost of producing complicated expressions to the user.

The SRM is an initial step towards a generic operating system specification. It should be extended to include additional features in support of distributed systems. A good first step would be generic message passing and remote procedure call.

The PLANNER's primitive search strategies work for operating systems that contain relatively few operations. As the number of operations increases, so does the the probability of the PLANNER making poor choices in attempting to achieve goals. Heuristics are needed to help the planner in making choices.

7.7 Extended Examples

This subsection contains details of the MITRE system specification used in the planner and specification details for the Millen example.

7.7.1 Description of Millen system operations

This section describes the modified Millen operating system operations used in the generic specification and in the planner. Table 3 contains the commands that only the operating system supervisor may issue (i.e., requires privileged mode). All block addresses used in these commands are the physical block addresses. Table 4 contains the commands that only the users may issue (requires unprivileged mode). All block addresses used in these commands are virtual addresses, which transformed to physical addresses via the memory management unit (MMU).

The Millen system has been modified so that each user has its own MMU and register set R, thus eliminating the need for the private storage areas described in the original paper.

In the operations described below, the following notation is used:

b	Physical block number
j	Virtual block number
k	Offset within block
R'	New value of R (after operation)
i,m	Register index

There are several system variables appearing in the operation tables that have not yet been discussed. These variables are used in the original Millen specification to enforce security constraints. The meaning of each of these is given below:

Cp	Current process id
Pal(c)	Security (access) level of user process *c*
Bal(b)	Security level of block *b*
Bap(b)	Activity status (block in use) of block *b*
Bac(b)	Number of processes with access to block *b*
Bdf(b)	Indicates whether block *b* is attached to an IO device

The security system used in the Millen operating system relies upon distinct security levels. These levels are not ordered in the low water mark sense: a user may only gain access to a block at its own level. Blocks can change level via the system command *Purge*, which sets the block security level equal to syshi (a level which no user process can reach). Block levels are also modified by system command *Get(b,n)*: this command is issued on behalf of a user process, and causes the block level to match the user security level. In addition, the user's MMU is set to point to the block, which has the side effect of releasing one of the user's current blocks.

Plan	Preconditions	Postconditions
$Purge(b)$	$Bac(b) = 0$ $\neg Bdf(b)$ mode = privileged	$Bal'(b) =$ syshi $\neg Bap'(b)$ $\forall k Mem'(b,k) = 0$ mode = unpriv
$Raise(b)$	$\neg Bap(b)$ mode = privileged	$Bal'(b) = Pal(Cp)$ $Bap'(b)$ mode = unpriv
$Get(b,n)$	$Bap(b)$ $Bac(b) = 0$ $Bal(b) = Pal(Cp)$ mode = privileged	$Bac'(b) = 1$ $Bac'(MMU(n)) = 0$ $MMU'(n) = b$ mode = unpriv
$Swap$	mode = privileged	$Cp' = (Cp + 1) \bmod P$

Table 3: Commands issued by the supervisor

Plan	Preconditions	Postconditions
$Fetch(i,j,k)$	mode = unpriv	$R'(i) = Mem(MMU(j),k)$
$Store(i,j,k)$	mode = unpriv	$Mem'(MMU(j),k) = R(i)$
$Compute_k(m)$	mode = unpriv	$R'(m) = f_k(\cup R(m))$

Table 4: Commands issued by users

Plan	Preconditions	Postconditions
$ReadX(i)$	mode = privileged	$R'(i) = \mathcal{X}$ mode = unpriv
$WriteX(i)$	mode = privileged	$\mathcal{X}' = R'(i)$ mode = unpriv

Table 5: Commands added to induce a flow violation

7.7.2 Specification of the Millen example

```
SRM  /* kind 1 */

/* Secure Resource Manager */

stateofSRM : SRM -> State
opsofSRM : SRM -> SRMopSet
schedofSRM : SRM -> Scheduler
interpofSRM : SRM -> Interp
polofSRM : SRM -> SecPol
initSRM : ObjectSet ProcessSet RequestList SRMopSet Scheduler Interp SecPol
           -> SRM
newreqSRM : Request SRM -> SRM
stepSRM : SRM -> SRM

DISTINGUISHING SET stateofSRM opsofSRM schedofSRM interpofSRM polofSRM;

initSRM (OB, PR, RL, OP, SCH, INT, POL) =>
   [initState(OB,PR,RL), OP, SCH, INT, POL];

newreqSRM (r, M) =>
   [addreqState(r,stateofSRM(M)),
    opsofSRM(M), schedofSRM(M), interpofSRM(M), polofSRM(M)];

stepSRM (M) =>
   [stepState(stateofSRM(M),schedofSRM(M),interpofSRM(M),polofSRM(M)),
    opsofSRM(M), schedofSRM(M), interpofSRM(M), polofSRM(M)];
```

```
State  /* kind 1 */

/* The operation updateobjState is a derive operation included for
   convenience in this application. */

/* This specification contains the definition of the most central operation
   of the entire specification, stepState. */

trivState : -> State
initState : ObjectSet ProcessSet RequestList -> State
updhistState : Request State -> State
addobjState : Object State -> State
addprocState : Process State -> State
addreqState : Request State -> State
remobjState : Object State -> State
remprocState : Process State -> State
remreqState : Int State -> State
objsofState : State -> ObjectSet
procsofState : State -> ProcessSet
reqsofState : State -> RequestList
histofState : State -> History
stepState : State Scheduler Interp SecPol -> State
updateobjState: Object Object State -> State

DISTINGUISHING SET objsofState procsofState reqsofState histofState;

trivState =>
   [emptyObjectSet, emptyProcessSet, nilRequestList, initHistory(trivState)];

initState (O, P, R) => [O, P, R, initHistory(initState(O,P,R))];

updhistState (r, S) =>
   [objsofState(S), procsofState(S), reqsofState(S),
    appendHistory(r,histofState(S))];

addobjState (o, S) =>
   [addObjectSet(o,objsofState(S)),
    procsofState(S), reqsofState(S), histofState(S)];

addprocState (p, S) =>
   [objsofState(S), addProcessSet(p,procsofState(S)),
    reqsofState(S), histofState(S)];

addreqState (r, S) =>
   [objsofState(S), procsofState(S), appendRequestList(r,reqsofState(S)),
    histofState(S)];

remobjState (o, S) =>
   [remObjectSet(o,objsofState(S)),
    procsofState(S), reqsofState(S), histofState(S)];

remprocState (p, S) =>
   [objsofState(S), remProcessSet(p,procsofState(S)),
    reqsofState(S), histofState(S)];

remreqState (n, S) =>
   [objsofState(S), procsofState(S), remRequestList(n,reqsofState(S)),
    histofState(S)];

stepState (S, SCH, I, P) =>
   let R be getreqScheduler(SCH,S) in
   let SS be
      updhistState(R,
                   [objsofState(S), procsofState(S), getreqlisScheduler(SCH,S),
                    histofState(S)]) in
   let RR be getreqSecPol(SS,R,P) in
   let RRI be getreqInterp(SS,RR,I) in
   let f be opofRequest(RRI) and A be argsofRequest(RRI) in apSRMop(f,SS,A);

updateobjState (o1, o2, S) => addobjState(o2,remobjState(o1,S));
```

```
ProcessSet  /* kind 1 */

/* This is an implementation of the "ideal" ProcessSet specification that is better
   for interactive use in that it speeds the computation necessary to display a
   ProcessSet and to perform the operation findinProcessSet. These things could, in
   principle, be done by allowing quantifiers (specifically, the "some" quantifier)
   in the definitions of findinProcessSet, firstinProcessSet, and restofProcessSet;
   such definitions would remain executable in the FASE system. */

emptyProcessSet : -> ProcessSet
addProcessSet : Process ProcessSet -> ProcessSet
remProcessSet : Process ProcessSet -> ProcessSet
isinProcessSet : Process ProcessSet -> Bool
findinProcessSet : ProcessPred ProcessSet -> Process
firstinProcessSet : ProcessSet -> Process
restofProcessSet : ProcessSet -> ProcessSet
isemptyProcessSet : ProcessSet -> Bool

DISTINGUISHING SET firstinProcessSet restofProcessSet;

emptyProcessSet => [ errProcess, emptyProcessSet ];

isinProcessSet (p, S) =>
   if isemptyProcessSet (S)
   ther false
   else (p = firstinProcessSet(S) | isinProcessSet(p,restofProcessSet(S)));

addProcessSet (p, S) =>
   if isemptyProcessSet(S) | precedesProcess(p, firstinProcessSet(S))
   then [p, S]
   else if p = firstinProcessSet(S)
        then S
        else [ firstinProcessSet(S),
               addProcessSet(p,restofProcessSet(S)) ];

remProcessSet (p, S) =>
   if isemptyProcessSet(S) then S
   else if p = firstinProcessSet(S) then restofProcessSet(S)
   else [ firstinProcessSet(S),
          remProcessSet(p,restofProcessSet(S)) ];

isemptyProcessSet (S) => iserrProcess(firstinProcessSet(S));

findinProcessSet (P, S) =>
   if isemptyProcessSet(S)
   then errProcess
   else let o be firstinProcessSet(S) in
        if apProcessPred(P,o)
        then o
        else findinProcessSet(P,restofProcessSet(S));

isemptyProcessSet (S) => iserrProcess(firstinProcessSet(S));
```

```
ProcessPred

/* The sort ProcessPred exists in order to make the definition of
   findinProcessSet, useful in the definitions of many SRMops, independent
   of the application. */

/* Only the constructor isidProcessPred is specific to this application. */

apProcessPred : ProcessPred Process -> Bool
trivProcessPred : Bool -> ProcessPred
isidProcessPred : ProcessId -> ProcessPred

DISTINGUISHING SET apProcessPred;

trivProcessPred (b) => [<p> |-> b];

isidProcessPred (i) => [<p> |-> idofProcess(p) = i];
```

```
Process  /* kind 3 */

/* Only the name and arity of the operator idofProcess is part of the template
   specification. */

idofProcess : Process -> ProcessId
eqProcess : Process Process -> Bool
inofProcess : Process -> Bool
levofProcess : Process -> Level
memblkofProcess : Int Process -> Object
regofProcess : Int Process -> Object
systemProcess : -> Process
mkuserProcess : Int Level Int -> Process
swapinProcess : Process -> Process
swapoutProcess : Process -> Process
precedesProcess : Process Process -> Bool
isprivileged : Process -> Bool
isregofProcess : Int Process -> Bool
updregProcess : Process Int Content -> Process
updmemblkProcess : Process Int Object -> Process

DISTINGUISHING SET idofProcess inofProcess levofProcess
                   regofProcess memblkofProcess;

eqProcess (p1, p2) => idofProcess(p1) = idofProcess(p2);

systemProcess =>
   [ sysProcessId, true, syshi, <n> |-> errObject, <n> |-> errObject ];

mkuserProcess (i, l, k) =>
   [ mkuserProcessId(i), false, l,
     <n> |-> if k > n then errObject
             else mkregObject(mkregObjectId(i,mkuserProcessId(i)),
                              errContent,
                              errLevel),
     <n> |-> errObject ];

swapinProcess (p) =>
   [idofProcess(p), true, levofProcess(p),
    <n> |-> regofProcess(n,p), <n> |-> memblkofProcess(n,p)];

swapoutProcess (p) =>
   [idofProcess(p), false, levofProcess(p),
    <n> |-> regofProcess(n,p), <n> |-> memblkofProcess(n,p)];
```

```
precedesProcess (p1, p2) => precedesProcessId(idofProcess(p1), idofProcess(p2));

isprivileged (p) => idofProcess(p) = sysProcessId;

isregofProcess (n, p) => ~iserrObject(regofProcess (n, p));

updregProcess (p, n, c) =>
   [idofProcess(p), inofProcess(p), levofProcess(p),
    <m> |-> if m=n then updatecontObject(regofProcess(m,p),c)
            else regofProcess(m,p),
    <m> |-> memblkofProcess(m,p)];

updmemblkProcess (p, n, o) =>
   [idofProcess(p), inofProcess(p), levofProcess(p), <m> |-> regofProcess(m,p),
    <m> |-> if m=n then o else memblkofProcess(m,p)];
```

```
RequestList  /* kind 1 */

nilRequestList : -> RequestList
nullRequestList : RequestList -> Bool
prependRequestList : Request RequestList -> RequestList
appendRequestList : Request RequestList -> RequestList
concatRequestList : RequestList RequestList -> RequestList
remRequestList: Int RequestList -> RequestList
hdRequestList: RequestList -> Request
tlRequestList: RequestList -> RequestList

DISTINGUISHING SET hdRequestList tlRequestList;

nilRequestList => [errRequest, nilRequestList];

nullRequestList (R) => iserrRequest(hdRequestList(R));

prependRequestList (r, R) => [r, R];

appendRequestList (r, R) =>
   if nullRequestList(R) then [r, R]
   else [hdRequestList(R), appendRequestList(r,tlRequestList(R))];

concatRequestList (R1, R2) =>
   if nullRequestList(R1) then R2
   else [hdRequestList(R1), concatRequestList(tlRequestList(R1),R2)];

remRequestList (n, R) =>
   if nullRequestList(R) then R
   else if n = 1 then tlRequestList(R)
   else [hdRequestList(R), remRequestList(n-1,tlRequestList(R))];
```

```
Request

nullRequest : -> Request
opofRequest : Request -> SRMop
argsofRequest : Request -> ArgList
oktoRequest : Request State -> Bool
procidofRequest : Request -> ProcessId
mkRequest : SRMop ArgList ProcessId -> Request

DISTINGUISHING SET opofRequest argsofRequest procidofRequest;

nullRequest => [NOop, errArgList, sysProcessId];

mkRequest (f, A, i) => [f, A, i];

oktoRequest (r, S) =>
   isprivileged(findinProcessSet(isidProcessPred(procidofRequest(r)),
                                 procsofState(S)));
```

```
History  /* kind 3 */

/* Only the operations initHistory and appendHistory, with their declared
   arities, are part of the template.  Their definitions and the remaining
   operations are peculiar to this application. */

startHistory : History -> State
listofHistory : History -> RequestList
initHistory : State -> History
appendHistory : Request History -> History

DISTINGUISHING SET startHistory listofHistory;

initHistory (S) => [S, nilRequestList];

appendHistory (r, H) => [startHistory(H),appendRequestList(r,listofHistory(H))];
```

```
ArgList  /* kind 1 */

nilArgList : -> ArgList
nullArgList : ArgList -> Bool
appendArgList : Arg ArgList -> ArgList
hdArgList: ArgList -> Arg
tlArgList: ArgList -> ArgList
nthArgList : Int ArgList -> Arg
lengthArgList : ArgList -> Int

DISTINGUISHING SET hdArgList tlArgList;

nilArgList => [errArg, nilArgList];

nullArgList (R) => iserrArg(hdArgList(R));

appendArgList (r, R) =>
   if nullArgList(R) then [r, R]
   else [hdArgList(R), appendArgList(r,tlArgList(R))];

nthArgList (n, R) =>
   if n<1 then errArg
   else if n=1 then hdArgList(R)
   else nthArgList(n-1,tlArgList(R));

lengthArgList (R) =>
   if nullArgList(R) then 0 else 1 + lengthArgList(tlArgList(R));
```

```
Arg  /* kind 3 */

/* Arg is an application dependent union type allowing a variety of arguments.
   ObjectIds and ProcessIds are guaranteed to be valid arguments in any
   application, and hence objidtoArg, objidofArg, procidtoArg, and procidofArg
   are part of the template.  Int and ComputeFn Args are special to this
   application.  The get...Arg operators are also special to this application,
   but we anticipate that their analogues (which, essentially, select
   arguments and do the appropriate coercions) will be useful in defining the
   SRMops of most other applications as well. */

objidtoArg : ObjectId -> Arg
procidtoArg : ProcessId -> Arg
inttoArg : Int -> Arg
fntoArg : ComputeFn -> Arg
objidofArg: Arg -> ObjectId
procidofArg: Arg -> ProcessId
intofArg : Arg -> Int
fnofArg : Arg -> ComputeFn
getblkidArg : Int ArgList -> ObjectId
getublkindArg : Int ArgList -> Int
getoffsetArg : Int ArgList -> Int
getregindArg : Int ArgList -> Int
getuidArg : Int ArgList -> ProcessId
getfnArg : Int ArgList -> ComputeFn

DISTINGUISHING SET objidofArg procidofArg intofArg fnofArg;

objidtoArg (i) => [i, errProcessId, errInt, errComputeFn];

procidtoArg (i) => [errObjectId, i, errInt, errComputeFn];

inttoArg (n) => [errObjectId, errProcessId, n, errComputeFn];

fntoArg (f) => [errObjectId, errProcessId, errInt, f];

getblkidArg (n, A) => objidofArg(nthArgList(n, A));

getublkindArg (n, A) => intofArg(nthArgList(n, A));

getoffsetArg (n, A) => intofArg(nthArgList(n, A));

getregindArg (n, A) => intofArg(nthArgList(n, A));

getuidArg (n, A) => procidofArg(nthArgList(n, A));
```

```
getfnArg (n, A) => fnofArg(nthArgList(n, A));
```

```
Content  /* kind 4 */

/* The sort Content is specific to this application.  A Content "is" either an
   Int (this will be the content of a register) or a map from Int to Int (which
   will be the content of a memory block). */

typeofContent : Content -> Symbol
regContent : Content -> Int
blkContent : Content Int -> Int
assignregContent : Int -> Content
assignblkContent : Content Int Int -> Content
isregContent : Content -> Bool
isblkContent : Content -> Bool

DISTINGUISHING SET typeofContent regContent blkContent;

assignregContent (n) => ['Register, n, <m> |-> errInt];

iserrContent (assignblkContent (c, n, N)) => typeofContent(c) = 'Register;

assignblkContent (c, n, N) =>
   ['Memblock, errInt, <m> |-> if m=n then N else blkContent(c,m)];

isregContent (c) => typeofContent(c) = 'Register;

isblkContent (c) => typeofContent(c) = 'Memblock;
```

```
Level  /* kind 4 */

/* The sort Level is particular to this application.  It is essentially the
   union of the sort Symbol with a one-element set (containing syshi).  The
   intent is to have a distinguished highest Level and a selection of other
   otherwise incomparable Levels. */

eqLevel : Level Level -> Bool
syshi : -> Level
mkLevel : Symbol -> Level
symofLevel : Level -> Symbol
issyshi : Level -> Bool

DISTINGUISHING SET issyshi symofLevel;

syshi => [true, errSymbol];

mkLevel (s) => [false, s];

eqLevel (l1, l2) => issyshi(l1)=issyshi(l2) & symofLevel(l1)=symofLevel(l2);
```

```
RegAssn  /* kind 4 */

/* The sort RegAssn is specific to this application.  Its function is to
   facilitate the definition of elements of sort ComputeFn, which permit
   user processes to do computations on the contents of the registers available
   to them. */

retrRegAssn : Int RegAssn -> Int
procRegAssn : Process -> RegAssn

DISTINGUISHING SET retrRegAssn;

procRegAssn (p) => [<n> |-> regContent(contofObject(regofProcess(n,p)))];
```

```
ComputeFn  /* kind 4 */

/* We have defined only the most obvious of the possible elements of sort
   ComputeFn here.  Clearly, one could also define any desired arithmetic
   operations. */

nameofComputeFn : ComputeFn -> Symbol
evalComputeFn : ComputeFn RegAssn -> Int
selectComputeFn : Int -> ComputeFn
constantComputeFn : Int -> ComputeFn

DISTINGUISHING SET nameofComputeFn evalComputeFn;

selectComputeFn (n) => [ 'select, <R> |-> retrRegAssn(n,R) ];

constantComputeFn (n) => [ 'constant, <R> |-> n ];
```

```
SRM /* secure resource manager */

opsofSRM : SRM -> SRMopSet
stateofSRM : SRM -> State
schedofSRM : SRM -> Scheduler
interpofSRM : SRM -> Interp
polofSRM : SRM -> SecPol
initSRM : SRMopSet ObjectSet ProcessSet RequestList Scheduler Interp SecPol
           -> SRM
newreqSRM : Request SRM -> SRM
stepSRM : SRM -> SRM

-

SRM ::= Request in SRM |-> newreqSRM(Request, SRM)
      | next SRM |-> stepSRM(SRM)

/* Note that a variable can parse as an SRM, so that this simple grammar is
   not useless. */
```

```
Request

nullRequest : -> Request
opofRequest : Request -> SRMop
argsofRequest : Request -> ArgList
oktoRequest : Request State -> Bool
procidofRequest : Request -> ProcessId
mkRequest : SRMop ArgList ProcessId -> Request

-

Request ::= ProcessId calls SRMop on ArgList
            |-> mkRequest(SRMop,ArgList,ProcessId)
```

```
ProcessId

sysProcessId : -> ProcessId
mkuserProcessId : Int -> ProcessId
typeofProcessId : ProcessId -> Symbol
intofProcessId : ProcessId -> Int
eqProcessId : ProcessId ProcessId -> Bool
precedesProcessId : ProcessId ProcessId -> Bool

-

ProcessId ::= user Int |-> mkuserProcessId(Int)
```

```
SRMop

nameSRMop : SRMop -> Symbol
apSRMop : SRMop State ArgList -> State
eqSRMop : SRMop SRMop -> Bool
NOop : -> SRMop
okargsSRMop : SRMop ArgList -> Bool
uStore : -> SRMop
Store : -> SRMop
uCompute : -> SRMop
Compute : -> SRMop
uFetch : -> SRMop
Fetch : -> SRMop
uReadX : -> SRMop
ReadX : -> SRMop
uWriteX : -> SRMop
WriteX : -> SRMop
uRelease : -> SRMop
Purge : -> SRMop
uGet : -> SRMop
Raise : -> SRMop
Get : -> SRMop
uSleep : -> SRMop
Swap : -> SRMop


SRMop ::= uStore |-> uStore()
        | Store |-> Store()
        | uCompute |-> uCompute()
        | Compute |-> Compute()
        | uFetch |-> uFetch()
        | Fetch |-> Fetch()
        | uReadX |-> uReadX()
        | ReadX |-> ReadX()
        | uWriteX |-> uWriteX()
        | WriteX |-> WriteX()
        | uRelease |-> uRelease()
        | Purge |-> Purge()
        | uGet |-> uGet()
        | Raise |-> Raise()
        | Get |-> Get()
        | uSleep |-> uSleep()
        | Swap |-> Swap()
```

```
ArgList

nilArgList : -> ArgList
nullArgList : ArgList -> Bool
appendArgList : Arg ArgList -> ArgList
hdArgList: ArgList -> Arg
tlArgList: ArgList -> ArgList
nthArgList : Int ArgList -> Arg
lengthArgList : ArgList -> Int

-

AUXILIARY NONTERMINALS X

ArgList ::= nil |-> nilArgList()
          | X |-> X

X ::= Arg |-> appendArgList(Arg,nilArgList())
    | X , Arg |-> appendArgList(Arg,X)
```

```
Arg

objidtoArg : ObjectId -> Arg
procidtoArg : ProcessId -> Arg
inttoArg : Int -> Arg
fntoArg : ComputeFn -> Arg
objidofArg: Arg -> ObjectId
procidofArg: Arg -> ProcessId
intofArg : Arg -> Int
fnofArg : Arg -> ComputeFn
getblkidArg : Int ArgList -> ObjectId
getublkindArg : Int ArgList -> Int
getoffsetArg : Int ArgList -> Int
getregindArg : Int ArgList -> Int
getuidArg : Int ArgList -> ProcessId
getfnArg : Int ArgList -> ComputeFn


Arg ::= Int |-> inttoArg(Int)
      | ObjectId |-> objidtoArg(ObjectId)
      | ProcessId |-> procidtoArg(ProcessId)
      | ComputeFn |-> fntoArg(ComputeFn)
```

7.8 High-level description of simple PLANNER

```
program
  state  Memory, Registers, Mmu, RegX, Uid, WaitingUid
  begin Memory
         doStoreMem(Memoryin, Mb, Ml, Rn, Memoryout) ;
         doPurge(Memoryin, Memoryout) ;
         doRaise(Memoryin, Mb, Memoryout) ;
         doGet(Memoryin, Mmuin, Mb, MMUid, Memoryout, Mmuout)
  end;

  begin Registers
         doFetchMem(Memoryin, Mb, Ml, Rn, Memoryout) ;
         doReadX(Memoryin, Mb, Ml, Memoryout)
  end;

  begin RegX
         doWriteX(Xin, Newvalue, Xout)
  end;

  begin Mmu
         doGet(Memoryin, Mmuin, Mb, MMUid, Memoryout, Mmuout)
  end;


  begin Uid
         doSwap(UidIn, WaitingUidIn, UidOut, WaitingUidOut)
  end;

  begin WaitingUid
         doSwap(UidIn, WaitingUidIn, UidOut, WaitingUidOut)
  end

  operation Store(Mb, Mn, Rn) with
         state  MemoryOut, Registers, Mmu, RegX, Uid, WaitingUid
         produces  Memory: [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
         by doStoreMem( Memory, Mphys, Mn, Newval, MemoryOut)
         with
                  precond fetchReg(Registers, Uid, Rn, Newval)
                           trigger Register
                           difference [[Uid, Rn, undefined], [Uid, Rn, Newval]]
                  end ;
                  precond memMap(Mmu, Uid, Mphys, Mb)
                           trigger User
                           difference [[Uid], [WaitingUid]]
                  end
  end;

  operation Fetch(Mb, Mn, Rn) with
          state  Memory, RegistersOut, Mmu, RegX, Uid, WaitingUid
          produces  Registers: [[User, Rn, undefined], [User, Rn, Newval]]
          by doStoreReg( Registers, User, Rn, Newval, RegistersOut)
          with

                    precond equal(Uid, User)
                              trigger Uid
                              difference [[Uid], [WaitingUid]]
                    end ;

                    precond doFetchMem(Memory, Mphys, Mn, Newval)
                              trigger Memory
                              difference [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
                    end ;

                    precond memMap(Mmu, User, Mphys, Mb)
                              trigger Uid
                              difference [[Uid], [WaitingUid]]
                    end

  end;

  operation ReadX(Mb, Mn) with
          state  MemoryOut, Registers, Mmu, Xout, Uid, WaitingUid
          produces Memory: [[Mphys, Mn, undefined], [Mphys, Mn, Xout]]
          by doStoreMem( Memory, Mphys, Mn, Xout, MemoryOut)
          with
                    precond equal(RegX, Xout)
                              trigger RegX
                              difference [[RegX], [Xout]]
                    end

  end
end
```

8 Axiomatic Verification of Concurrency in SR

In the two-tier methodology that we are developing, the axiomatic tier provides the means to perform concrete verification of the correctness of actual running implementations. In this regard, our methodology is more complete that the methodologies presented in the current literature, in that the majority of current researchers are focusing solely on the level of specification of concurrency, with no solid links to operational code.

The foundation necessary to verify operational code is the formal semantics of an operational programming language. The next two subsections present our current work on building this foundation for the SR programming language. The first subsection describes the syntactic enhancements that are currently being added to the SR language to support verification. The second subsection then outlines our most current version of the axiomatic semantics of SR. These semantics provide the basis for completing our two-tiered methodology.

8.1 The Syntax of Annotated SR

SYSTEM

```
system        ::=  ( systemcomp // ";" ) *
systemcomp    ::=  globalcomp | rescomp | blockcomp | SYSINVARIANTS
globalcom     ::=  "global" id ( globaldecl // ";" )* "end" [ id ]
globaldecl    ::=  constdecl | typedecl | optypedecl
rescomp       ::=  abstractspec sepspec | specandbody | sepbody
sepspec       ::=  concretespec "separate"
specandbody   ::=  concretespec resbody "end" [ "id" ]
sepbody       ::=  "body" id resbody "end" [ "id" ]
abstractspec  ::=  "resource" id speccomplist "end" [ id ]
concretespec  ::=  "resource" id [ speccomplist "body" id ] "(" ( parmspec // ";" )* ")"
speccomplist  ::=  ( speccomp // ";" )*
speccomp      ::=  importclause | extendclause | operdecl | constdecl | typedecl |
                   optypedecl | EQNDECL | SPECRESTR | RESINVARIANT | SPECVARDECL
importclause  ::=  "import" ( id // "," )*
extendclause  ::=  "extend" ( id // "," )*
resbody       ::=  (bodycomp // ";")* "end" [ id ]
bodycomp      ::=  initial | final | procdecl | processdecl | importclause | decl
initial       ::=  "initial" block "end" [ "initial" ]
final         ::=  "final" block "end" [ "final" ]
block         ::=  ( blockcomp ";" ) *
blockcomp     ::=  decl | stmt | importclause
decl          ::=  typedecl | constdecl | vardecl | optypedecl | operdecl
```

TYPES

```
typedesig     ::=  typedefn | var
typedecl      ::=  "type" id "=" typedefn [ typerestr ]
typedefn      ::=  enumdefn | stringdefn | recorddefn | ptrdefn | capdefn
typerestr      :=  "{ public }" | "{ private }"
enumdefn      ::=  "enum" "(" ( id // "," ) * ")"
stringdefn    ::=  "string" "(" expr ")" | "string ( * ) "
recorddefn    ::=  "rec" "(" [ UninitVar // ";" ] ")"
ptrdefn       ::=  "ptr" typedesgi
capdefn       ::=  "cap" var | "cap" operspec
```

DECLARATIONS

```
vardecl              ::=  “var” ( vardefn // “,”)
constdecl            ::=  “const” ( initvar // “,”)
vardefn              ::=  initvar | uninitvar
initvar              ::=  varname “:=” expr | varname “:” typedesig “:=” expr
uninitvar            ::=  ( varname // “,” )+ “:” typedesig
varname              ::=  id | id subscripts
subscripts           ::=  “[” range “]” | “[” range “,” range “]”
range                ::=  expr | expr “:” expr | “*” | “*” “:” expr | expr “: *”
SYSINVARIANTS        ::=  FORMULA *
RESINVARIANT         ::=  FORMULA *
EQNDECL              ::=  “eqn” [ “let” [ vardecl — constdecl ]] (EQUATION;)* “nqe”
SPECRESTR            ::=  GENERATORS | PARTITIONS | CONSEQUENCES | EXEMPTS
SPECVARDECL          ::=  vardecl
```

OPS, PROCS, etc

```
optypedecl           ::=  “optype” id [ “=” ] operspec
operdecl             ::=  operclass opername operspec | operclass opername “:” id
operclass            ::=  “op” | “external”
opername             ::=  id | id subscripts
operspec             ::=  “(” [ Parmspec // “;” ] “)” [ returnspec ] [IOSPEC ] oprestr
parmspec             ::=  [ parmkind ] uninitvar
returnspec           ::=  “returns” varname “:” typedesig
oprestr              ::=  “[ call ]” | “[ send ]” | “[ call , send ]” | “[ send , call ]”
parmkind             ::=  “var” | “val” | “res” | “ref”
procdecl             ::=  “proc” id “(” [ id // “,” ] “)” [ “returns” id ] block “end” [ id ]
processdecl          ::=  “process” id block “end” [ id ]
IOSPEC               ::=  PRE | POST | LIVE | FAIR | RELIES | GUARANTEES
```

STATEMENTS

```
stmt                 ::=  segstmt | interstmt
seqstmt              ::=  skipstmt | stopstmt | asgnstmt | swapstmt | incstmt | decstmt |
                          ifstmt | dostmt | forallstmt | exitstmt | nextstmt | assert
skipstmt             ::=  “skip”
stopstmt             ::=  “stop” [ exitcode ]
exitcode             ::=  “(” expr “)”
asgnstmt             ::=  var “:=” expr
incstmt              ::=  var “++”
decstmt              ::=  var “--”
swapstmt             ::=  var “:=:” var
ifstmt               ::=  “if” guardedcmd “fi” | “if” guardedcmd ifstmt1 “fi”
ifstmt1              ::=  “[]” guardedcmd | “[]” guardedcmd ifstmt1 | elsecmd
dostmt               ::=  “do” guardedcmd “od” | “do” guardedcmd dostmt1 “od”
dostmt1              ::=  “[]” guardedcmd | “[]” guardedcmd dostmt1 | “[]” elsecmd
```

```
guardedcmd        ::=   expr "- >" block
elsecmd           ::=   "else - >" block
forallstmt        ::=   "fa" ( quant // "," ) "- >" block "fa"
exitcmd           ::=   "exit"
nextcmd           ::=   "next"
quant             ::=   id ":=" expr direction expr [ "st" expr ]
direction         ::=   "to"  | "downto"
interstmt         :: =  invocstmt  | implstmt  | rescontrolstmt  | concstmt
invocstmt         ::=   callstmt  | sendstmt
callstmt          ::=   var  | "call" var
sendstmt          ::=   "send" var
implstmt          ::=   inputstmt  | receivestmt  | returnstmt  | replystmt
inputstmt         ::=   "in" ( opercmd // "[]" ) "ni"
receivestmt       ::=   "receive" operindication "(" [ var // "," ] ")"
returnstmt        ::=   "return"
replystmt         ::=   "reply"
GENERATORS        ::=   operspec "generated by" ???
PARTITIONS        ::=   operspec "partitioned by" ???
opercmd           ::=   [ interquant ] operguard [ schedexpr ] "- >" block
operguard         ::=   operindication "(" [ id // "," ] ")" [ "returns" id ] [ syncexpr ]
interquant        ::=   "(" ( quant // "," ) ")"
operindication    ::=   id  | id opersubscripts
opersubscripts    ::=   "[" expr "]"  | "[" expr "," expr "]"
syncexpr          ::=   "and" expr
schedexpr         ::=   "by" expr
rescontrolstmt    ::=   createstmt  | destrystmt
createstmt        ::=   var ":= create" id "(" [ expr // "," ] ")" [ location ]
location          ::=   "on" var
destroystmt       ::=   "destroy" var
concstmt          ::=   "co" ( conccmd // "//" ) "oc"
conccmd           ::=   [ interquant ] concinvoc [ postproc ]
concinvoc         ::=   callstmt  | asgnstmt
postproc          ::=   "- >" block
```

EXPRESSIONS

```
FORMULA           ::=   expr /* Evaluate with set operators on specvars */
expr              ::=   opexp [ orop expr ]
orexpr            ::=   andexpr [ andop orexpr ]
andexpr           ::=   relexpr [ relop and expr ]
relexpr           ::=   shfexpr [ shfop relexpr ]
shfexpr           ::=   addexpr [ addop shfexpr ]
addexpr           ::=   mulexpr [ mulop addexpr ]
mulexpr           ::=   [ unop ] simpleexpr
simpleexpr        ::=   literal  | var  | arrayaggr
```

HYPER LITERALS

```
arrayaggr           ::=  “(” ( vectorelem // “,” ) “)”
vectorelem          ::=  [ “[” expr “]” ] expr
```

LITERALS

```
literal             ::=  number | stringliteral | “true” | “false” | “null” | “noop”
```

VARIABLES

```
var                 ::=  objnamelist [ args ]
objnamelist         ::=  ( objname // “.” )
objname             ::=  id indices “^” * | id “^” *
indicies            ::=  “[” expr “]” | “[” slice “]” | “[” expr “,” expr “]” | “[” expr “,” slice
                         “[” slice “,” expr “]”
slice               ::=  expr “:” expr | expr “: *” | “* :” expr
args                ::=  “(” [ expr // “,” ] “)”
```

OPERATORS

```
unop                ::=  “-” | “ ” | “not” | “@” | “?”
mulop               ::=  “*” | “/” | “%”
addop               ::=  “+” | “-” | “||”
shfop               ::=  “<<” | “>>”
relop               ::=  “=” | “! =” | “ =” | “>=” | “>” | “<=” | “<” | “and” | “&”
orop                ::=  “or” | “|” | “xor”
```

8.2 The Axiomatic Semantics of SR

operation invocation

call_statement

< b,p> $\{P \mid P'^{in\_actual}_{actual}\}$ call op_denotation (actual) $\{R\}$

send_statement

< b,p> $\{P \mid P'^{in\_actual}_{actual}\}$ **send** op_denotation (actual) $\{P\}$

where actual = = {val_args, res_args, var_args}
P is the precondition of a call/ send
P' is used in satisfaction proof

- evaluating the argument expressions

- serviced by a proc → create new process

- serviced by an input statement → queue the invocation

- invocations of the same operation by the same invoking process are queued in the order they are invoked.

- A call statement terminates when the operation has been serviced and results have been returned

- A send statement terminates when a service process has been created or when the arguments have been queued.

proc

proc oper (in_formal) **returns** result S **end**

input_statement

in op_i (in_formal) **returns** $result_i$ **and** B_i **by** $E_i \rightarrow S_i$ $[]^{+}$ **ni**

- class ≡ operations implemented by the same input statement

- For a given class, at most one process at a time can be selecting an invocation to service or appending a new invocation. The access is in FCFS order.

- input statement delays the executing process until some invocation is selectable

- an invocation is selectable if the boolean-valued synchronization expression in the corresponding operation guards is true

- no scheduling expression → the oldest selectable invocation is serviced

- scheduling expression → the oldest selectable that also minimizes the scheduling expression is serviced

- both synchronization and scheduling expression can reference invocation arguments

- termination: the selected block terminates, return is executed, or exit/next is executed (if input statement is within an iterative statement)

- results are returned to the caller when the input statement terminates or when a reply statement is executed

- reply/return is associated with a smallest enclosing input statement or proc

- a process that executes reply continues executing with the statement following reply.

```
in request(time) and free by time → free := false
[] release() → free := true
ni
```

{Q} **proc** oper (in_formal) **returns** result {T} S {X} **end** {W}

{Q} **in** op_i (in_formal) **returns** $result_i$ **and** B_i **by** E_i → {T_i} S_i {X_i} $[]^{+}$ **ni** {W}

{U} or {U_i} is assertion about invocation's termination

proc

(< b,p> {$T^{r\_aux}_{false}$} S {X} /\ ¯r_aux) → (X → U),
∀ j,k : (< j,k> {V} **reply** {V} /\ k= p /\ ¯r_aux) → (V → $U^{r\_aux}_{true}$),
∀ j,k : (< j,k> {V} **return** {false} /\ k= p /\ ¯r_aux) → (V → ($U^{r\_aux}_{true}$ /\ X))

< b,p> {Q} **proc** oper (in_formal) **returns** result S **end** {W}

input_statement

∀ i : X_i → W,
(< b_i,p_i> {T'$_i$ | $T_i{}^{r\_aux_i}_{false}$ } S_i {X_i} /\ ¯r_aux_i) → (X_i → U_i),
∀ j,k : (< j,k> {V} **reply** {V} /\ k= p_i /\ ¯r_aux_i) → (V → $U_i{}^{r\_aux_i}_{true}$),
∀ j,k : (< j,k> {V} **return** {false} /\ k= p_i /\ ¯r_aux_i → (V → (U_i /\ X_i)),
∀ j,k : (< j,k> {V} **next** {false} /\ j= b /\ ¯r_aux_i) → (V → U_i),
∀ j,k : (< j,k> {V} **exit** {false} /\ j= b /\ ¯r_aux_i) → (V → U_i)

< b,p> {Q} **in** $oper_i$ (in_formal) **returns** $result_i$ **and** B_i **by** E_i → S_i $[]^{+}$ **ni** {W}

RPC rule

- **Satisfaction proof**

For every call c and matching proc (including process), prove Sat_{rpc} (c,proc) valid.

Sat_{rpc} (c,proc):

$$
\begin{aligned}
& P \rightarrow P' \\
& \wedge (Q \wedge P) \rightarrow T^{in\_formal}_{in\_actual} \\
& \wedge (U \wedge P') \rightarrow R^{out\_actual\ result\_id}_{out\_formal\ result} \\
& \wedge (P' \wedge X) \rightarrow W
\end{aligned}
$$

- **Noninterference Proof**

For every assignment statement S and assertion I parallel to S, prove NI_{rpc} (S,I) valid.

NI_{rpc} (S, I):

$$\{I \wedge pre(S)\}\ \ S\ \ \{I\}$$

For every call c and matching proc and every assertion I parallel to both c and proc, prove NI_Sat_{rpc} (c,proc,I) valid.

NI_Sat_{rpc} (c, proc, I):

$$(I \wedge Q \wedge P) \rightarrow I^{in\_formal}_{in\_actual} \wedge (I \wedge P' \wedge U) \rightarrow I^{out\_actual\ result\_id}_{out\_formal\ result}$$

For every assignment statement S of proc, and assertion I in proc that references global variables, prove NI_{self} (S,I):

NI_{self} (S, I):

$$\{I \wedge pre(S)\}\ \ S\ \ \{I\}$$

Rendezvous rule

- **Satisfaction proof**

For every call c and matching op_i of input statement, and every op_j of the same input statement, prove Sat_{rndvs} (c,op_i) valid.

Sat_{rndvs} (c, op_i):

$$
\begin{aligned}
&(P \wedge \neg lock(Pending_{class(op_i)})) \rightarrow P'^{\,pending_{class(op_i)}}_{\,pending_{class(op_i)}+inv(op_i)} \\
&\wedge (Q \wedge P \wedge \neg lock(Pending_{class(op_i)})) \rightarrow T_i'{}^{\,in\_formal_{op_i}, lock(Pending_{class(op_i)})}_{\,in\_actual_{inv(op_i)}, true} \\
&\wedge (Q \wedge P \wedge lock(Pending_{class(op_i)})) \rightarrow \text{false} \\
&\wedge (T_i' \wedge C) \rightarrow T_i{}^{\,in\_formal_{op_i}, lock(Pending_{class(op_i)}), Pending_{class(op_i)}}_{\,in\_actual_{inv(op_i)}, false, Pending_{class(op_i)}-inv(op_i)} \\
&\wedge (T_i' \wedge \neg C) \rightarrow Q^{\,lock(Pending_{class(op_i)})}_{\,false} \\
&\wedge (P' \wedge U_i) \rightarrow R^{\,out\_actual_{inv(op_i)}, result\_id_{inv(op_i)}}_{\,out\_formal_{op_i}, result_{op_i}} \\
&\wedge (P' \wedge X_i) \rightarrow W
\end{aligned}
$$

where $C = B_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv(op_i)}}$

$$
\begin{aligned}
&\wedge (\forall\, inv'(op_i) : B_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv'(op_i)}} \rightarrow pos(inv'(op_i)) > pos(inv(op_i)) \\
&\qquad \vee (B_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv'(op_i)}} \rightarrow (E_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv'(op_i)}} > E_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv(op_i)}} \\
&\qquad\quad \vee (E_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv'(op_i)}} = E_i{}^{\,in\_formal_{op_i}}_{\,in\_actual_{inv(op_i)}} \wedge pos(inv'(op_i)) > pos(inv(op_i)))) \\
&\wedge (\forall\, inv(op_j) : pos(inv(op_j)) < pos(inv(op_i)) \rightarrow \neg B_j{}^{\,in\_formal_{op_j}}_{\,in\_actual_{inv(op_j)}})
\end{aligned}
$$

- **Noninterference Proof**

For every assignment statement S and assertion I parallel to S, prove NI_{rndvs} (S,I) valid.

NI_{rndvs} (S,I): {I $\wedge$ pre(S)} S {I}

For every call c and matching oper of input statement and every assertion I parallel to both c and oper, prove NI_Sat_{rndvs} (c, oper,I) valid.

NI_Sat_{rndvs} (c,oper,I):

$$
(I \wedge Q \wedge P) \rightarrow I^{\,in\_formal_{op_i}}_{\,in\_actual_{inv(op_i)}} \wedge (I \wedge P' \wedge U_i) \rightarrow I^{\,out\_actual_{inv(op_i)}, result\_id_{inv(op_i)}}_{\,out\_formal_{op_i}, result_{op_i}}
$$

Dynamic process creation rule

- **Satisfaction proof**

For every send s and matching proc (including process), prove Sat_{proc} (s,proc) valid.

Sat_{proc} (s,proc):

$$P \rightarrow P' \wedge (P \wedge Q) \rightarrow T^{in\_formal}_{in\_actual} \wedge X \rightarrow W$$

- **Noninterference Proof**

For every assignment and send statement S and every assertion I parallel to S, prove NI_{proc} (S,I) valid.

NI_{proc} (S,I): $\{I \wedge pre(S)\}\ S\ \{I\}$

For every proc/process heading S and every assertion I parallel to S, prove NI_Sat_{proc} (S,I) valid.

NI_Sat_{proc} (S,I): $(I \wedge pre(S)) \rightarrow I^{in\_formal}_{in\_actual}$

For every assignment statement S of proc, and assertion I in proc that references global variables, prove NI_{self} (S,I):

NI_{self} (S, I):

$$\{I \wedge pre(S)\}\ S\ \{I\}$$

Message passing rule

- **Satisfaction proof**

For every send s and matching op_i of input statement, and every op_j of the same input statement, prove Sat_{msg} (s,op_i) valid.

Sat_{msg} (s, op_i):

$$\begin{aligned}
&(P \wedge \neg lock(Pending_{class(op_i)})) \rightarrow P'^{\,pending_{class(op_i)}}_{\,pending_{class(op_i)}+inv(op_i)} \\
&\wedge (Q \wedge \neg lock(Pending_{class(op_i)})) \rightarrow T_i'^{\,in\_formal_{op_i}, lock(Pending_{class(op_i)})}_{\,in\_actual_{inv(op_i)}, true} \\
&\wedge (Q \wedge lock(Pending_{class(op_i)})) \rightarrow \text{false} \\
&\wedge (T_i' \wedge C) \rightarrow T_i{}^{in\_formal_{op_i}, lock(Pending_{class(op_i)}), Pending_{class(op_i)}}_{in\_actual_{inv(op_i)}, false, Pending_{class(op_i)} - inv(op_i)} \\
&\wedge (T_i' \wedge \neg C) \rightarrow Q^{lock(Pending_{class(op_i)})}_{false} \\
&\wedge (P' \wedge X_i) \rightarrow W
\end{aligned}$$

where

$$\begin{aligned}
C = {} & B_i{}^{in\_formal_{op_i}}_{in\_actual_{inv(op_i)}} \\
& \wedge (\forall\, inv'(op_i) : B_i{}^{in\_formal_{op_i}}_{in\_actual_{inv'(op_i)}} \rightarrow pos(inv'(op_i)) > pos(inv(op_i)) \\
& \qquad \vee B_i{}^{in\_formal_{op_i}}_{in\_actual_{inv'(op_i)}} \rightarrow (E_i{}^{in\_formal_{op_i}}_{in\_actual_{inv'(op_i)}} > E_i{}^{in\_formal_{op_i}}_{in\_actual_{inv(op_i)}} \\
& \qquad\quad \vee (E_i{}^{in\_formal_{op_i}}_{in\_actual_{inv'(op_i)}} = E_i{}^{in\_formal_{op_i}}_{in\_actual_{inv(op_i)}} \wedge pos(inv'(op_i)) > pos(inv(op_i)))) \\
& \wedge (\forall\, inv(op_j) : pos(inv(op_j)) < pos(inv(op_i)) \rightarrow \neg B_j{}^{in\_formal_{op_j}}_{in\_actual_{inv(op_j)}})
\end{aligned}$$

- **Noninterference Proof**

For every assignment and send statement S and every assertion I parallel to S, prove NI_{msg} (S,I) valid.

NI_{msg} (S,I): {I /\ pre(S)} S {I}

For every operation indication or receive S and every assertion I parallel to S, prove NI_Sat_{msg}(S,I) valid.

NI_Sat_{msg} (S,I): $(I \wedge pre(S)) \rightarrow I^{in\_formal_{op_i}}_{in\_actual_{inv(op_i)}}$

number assignment rule

Each statement is attached with two sequence numbers delimited by < > . b and p are rvalues, b_aux and p_aux are auxiliary variables of type int, r_aux is auxiliary variable of type bool.

$$\frac{\text{< b,p> } S}{\text{< b,p> } S_1 \,;\, \text{< b\_aux++,p> } S_2 \,;\, \text{< b,p> } S_3}$$

where S $\equiv$ $S_1; S_2; S_3$
S_1 and S_3 are not one of the following: do, fa, and co
S_2 is one of the following: do, fa, or co.

$$\frac{\text{< b,p> if } B_i \rightarrow S_i \ \Box^{+} \text{ fi}}{\text{< b,p> if } B_i \rightarrow \text{< b,p> } S_i \ \Box^{+} \text{ fi}}$$

$$\frac{\text{< b,p> do } B_i \rightarrow S_i \ \Box^{+} \text{ od}}{\text{< b,p> do } B_i \rightarrow \text{< b,p> } S_i \ \Box^{+} \text{ od}}$$

where y = $x_i \vee \ldots \vee x_n$

$$\frac{\text{proc oper (formal) returns arg S end}}{\text{proc oper (formal) returns arg < b\_aux++, p\_aux++> S end}}$$

$$\frac{\text{process oper S end}}{\text{process oper < b\_aux++, p\_aux++> S end}}$$

$$\frac{\text{< b,p> in } oper_i \ (formal_i) \text{ returns } r_i \text{ and } B_i \text{ by } expr_i \rightarrow S_i \ \Box^{+} \text{ ni}}{\text{< b,p> in } oper_i \ (formal_i) \text{ returns } r_i \text{ and } B_i \text{ by } expr_i \rightarrow \text{< b,p\_aux++> } S_i \ \Box^{+} \text{ ni}}$$

if_statement

$$\frac{\begin{array}{l} < b,p > \{P \wedge B_i\}\ S_i\ \{Q\}, \\ (P \wedge \neg BB) \rightarrow Q \end{array}}{< b,p > \{P\}\ \text{if}\ B_i \rightarrow S_i\ \Box^{+}\ \text{fi}\ \{Q\}}$$

$$\frac{\begin{array}{l} < b,p > \{P \wedge B_i\}\ S_i\ \{Q\}, \\ < b,p > \{P \wedge \neg BB\}\ S_n\ \{Q\} \end{array}}{< b,p > \{P\}\ \text{if}\ B_i \rightarrow S_i\ \Box^{+}\ \text{else} \rightarrow S_n\ \text{fi}\ \{Q\}}$$

where $B_i = \{B_1, B_2, ..., B_{n-1}\}$
$BB \equiv B_1 \vee B_2 \vee ... \vee B_{n-1}$

do_statement

$$\frac{\begin{array}{l} < b,p > \{I \wedge B_i\}\ S_i\ \{I\}, \\ \forall j,k : (< j,k > \{R\}\ \text{exit}\ \{\text{false}\} \wedge j = b) \rightarrow (R \rightarrow Q), \\ \forall j,k : (< j,k > \{R\}\ \text{next}\ \{\text{false}\} \wedge j = b) \rightarrow (R \rightarrow I) \end{array}}{< b,p > \{I\}\ \text{do}\ B_i \rightarrow S_i\ \Box^{+}\ \text{od}\ \{(I \wedge \neg BB) \vee Q\}}$$

where $B_i = \{B_1, B_2, ..., B_{n-1}\}$
$BB \equiv B_1 \vee B_2 \vee ... \vee B_{n-1}$

compound statement

$$\frac{\begin{array}{l} < b_1,p_1 > \{P\}\ S_1\ \{R\}, \\ < b_2,p_2 > \{R\}\ S_2\ \{Q\} \end{array}}{< b_1,p_1 > \{P\}\ S_1; < b_2,p_2 > S_2\ \{Q\}}$$

9 Working Examples and Models

9.1 A Secure Network Mail System Model

9.1.1 Introduction

A mail system is secure if it can protect mail from unauthorized access, unauthorized modification or unauthorized disclosure. This section proposes a secure network mail system to accomplish this goal. The proposed model is independent of the techniques used to implement the system. The model is based on the network mail system described by Owicki [Owi80], and on the secure military message systems proposed by Landwehr et al [LH*84].

The Owicki network mail system [Owi80] has a ring structure. Each server consists of three processes and three buffers. An incoming message from a neighboring node is read by the read process; the message and all other messages sent by local users are stored in the switch buffer. The switch process then retrieves messages from the switch buffer and deposits them into either the output buffer or the user buffer depending on the destination of the mail. The output process will read message from the output buffer and forward it to the next node. Local users receive mail from the user buffer. The system assumes that all messages sent out by a node are guaranteed to be received by the neighboring node. No security is imposed on the system except that messages are only delivered to the correct receivers. However, the system does guarar. ee delivery as long as the number of undelivered messages is less than the number of messages in the switch buffer and the output buffer.

The Landwehr secure military message systems model [LH*84] supports arbitrary network connection and captures the security policy that a military message system must enforce. The model describes multilevel secure systems that protect information of different classifications from users with different clearances. An object is a single-level unit of information and a container is a multilevel information structure. In general, a message is a container. In this model, a user may display, update, create, delete or release a message. Security is enforced by the system. Only an authenticated user may log in the system, i.e., by presenting a unique user ID and by passing the system authentication check. Once entering the system, a user may invoke an authorized operation, and may access to information that has passed the system's flow control check. For example, a user can view an entity with a classification less than or equal to the user's clearance and the classification of the output medium.

As noted in the introduction, the proposed architecture of each server in the mail system is similar to the Owicki mail server [Owi80], and the security model is derived from the security model of [LH*84], [LS86].

9.1.2 Overall Description

The proposed network mail system model supports arbitrary network connection and is a multilevel secure system model. Each node has one server with five processes and three buffers. The five processes are send, accept, switch, retrieve, and forward processes. The names of three buffers are switch buffer, local buffer and forward buffer (see Figure 1). Each process is assigned to top security clearance. Each buffer is a multilevel information structure and stores mail that is processed by the server. A mail is a single-level unit and consists of two parts: a header and text. A mail header contains sender ID, receiver ID, classification of the mail and timestamp that records when the mail is entered the system.

Each user and node are assumed to have a unique ID in the network. It is further assumed

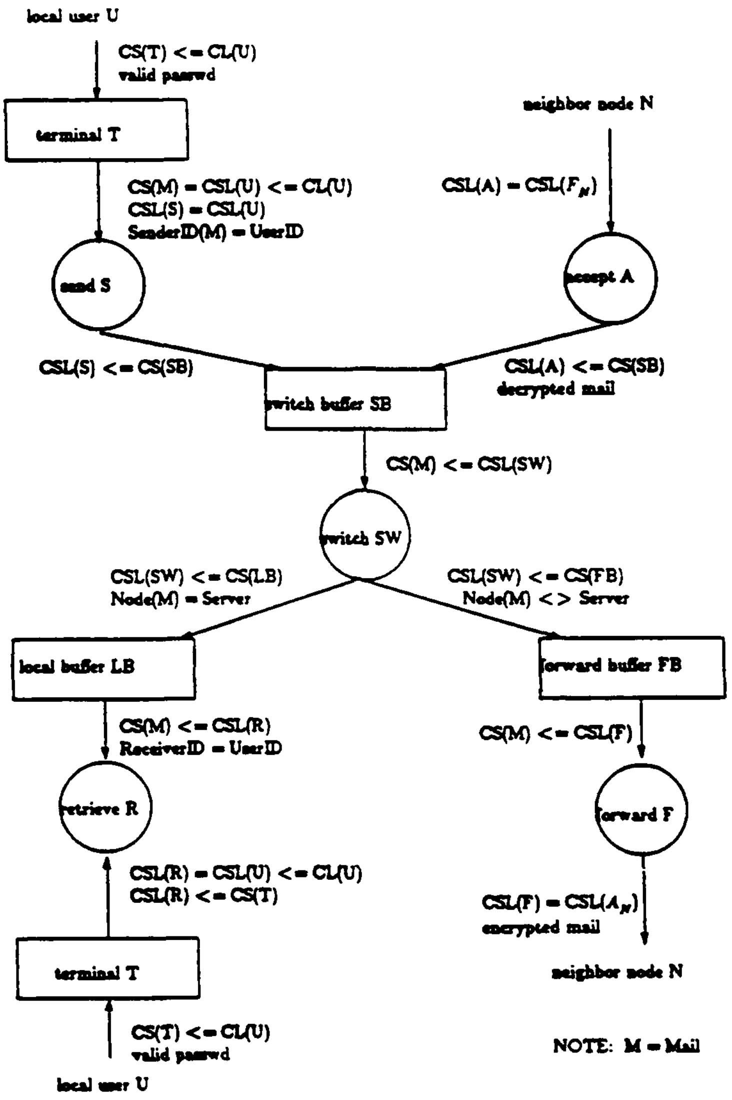

Figure 1: Mail Server Architecture

that a node knows how to forward mail to the proper destination, that all server processes can be trusted and that connections of network devices are secure. A user may send, retrieve, or save mail. However, sending and retrieving mail must be performed by the mail server on user's behalf.

Security is enforced by the system and consists of three parts: access control, flow control, and privacy and integrity control. Access control prevents unauthorized access to an entity connected to the network. Only authorized users may login the system, and only authorized server processes may read or write into server buffers. Flow control prevents unauthorized dissemination of mail stored in a buffer. A user may not read mail which has higher classification than the clearance of the setting. Privacy and integrity control prevents unauthorized disclosure or modification of mail that are being transmitted between nodes. This is to deter an eavesdropper from intercepting mail.

9.1.3 The Model

In order to describe the model in details, first of all, we present the definitions of terms used. Then we describe user's operations. Finally, we describe roles of server processes. Security issues are discussed as parts of each descriptions.

Definitions The following terms are used in this model and their definitions are provided as a basis for the model.

- *Entity* – a network resource (device, file, mail, server process, server buffer) or a legitimate user.
- *Object* – a passive entity that is acted upon by another entity. An object may be a device, a mail message, or a buffer.
- *Mail* – a single-level unit of information. A mail message contains a header and text.
- *Classification* – a security level attached to an object. A classification includes a sensitivity level and set of compartments.
- *Buffer* – a multi-level information unit containing mail.
- *Subject* – a subject may be a user or a server process acting on behalf of a user.
- *ID* – identifier. A string of characters names a unique entity.
- *Clearance* – a degree of trust associated with a subject.
- *Current Security Level* – the clearance of a subject that is currently being recognized. It must be less than or equal to the clearance of the subject.

Description of User Operations *Login:* A user can gain access to the mail system only by logging in. In order to login, the user provides a user ID and a password. The system will authenticate the information provided by the user, the login process completed successfully only if and when the system recognized the user as a legitimate subject and the clearance of the user is higher than or equal to the classification of the terminal that the user is using.

Send request: After a successful login, a user may compose a mail message. The user selects a current security level that must be equal to or less than the maximum clearance of the user. The clearance of the server is set to a level equal to the current security level and the created message will be classified at the same level. The user then enters the receiver ID and text. If the text is from a file, the classification of the file must be less than or equal to the current security level of the user.

Retrieve request: A user must first login the system. After a successful login, the user selects a current security level and requests server to retrieve mail. A mail message may only be displayed on a terminal if the message is intended for the user and the clearance of the terminal is higher than or equal to the classification of the mail.

Save request: The retrieved mail will be saved as a file and the file is classified at the same level as the mail.

Description of Server Processes *Send process:* When a user enters a send mail request, the server's send process sets its clearance to the current security level of the user. Before a mail message can be admitted into the network, the send process timestamps the message and deposits it into the switch buffer.

Accept process: In order to receive mail from a neighboring node, the clearance of the accept process must be equal to the clearance of the forward process of the neighboring node. The accept process will decrypt the received mail (link to link decryption) and then deposit the received mail into the switch buffer.

Switch process: The switch process reads mail from the switch buffer and deposits it into either the local buffer or the forward buffer depending on the destination of the mail. The local buffer contains mail intended for local users and the forward buffer contains mail intended for users of other nodes.

Retrieve process: The retrieve sets its clearance to the current security level of the user, and reads mail from the local buffer on user's behalf. The retrieve process may only read mail that is intended for the user and have a classification less than or equal to the clearance of the retrieve process.

Forward process: The forward process reads mail from the forward buffer, determines the next node to be routed and establishes connection with the accept process of that node. Before transmitting the mail to the next node, the forward process encrypts the mail. The current security level of the forward process and the accept process must be equal in order to establish the connection.

Note that *send process* and *retrieve process* are parts of the *send request* and *retrieve request* operations described below. The reason is that the server executes these operations on user's behalf.

9.1.4 Security Specification

This section describes the security assumptions, the security assertions and the security requirements of the proposed model. The security assumptions contains rules that are beyond mail server's control.

Security Assumptions *User behavior assumptions:*

a. The Network Security Officer assigns clearances, device classification properly.

b. Sender selects a correct current security level when sending mail.

c. User with higher security clearance may know the security level of user with lower clearance, but not vice versa.

d. User will follow the established security regulation.

Network/Server behavior assumptions:

server:

a. *login secure*

```
CS(T) <= CL(U)
valid authentication
```

b. *reclassify secure*: to set current security level of a subject.

```
CSL(U) <= CL(U)
```

c. *send secure*

```
CSL(U) <= CL(U)
CS(M) = CSL(U)
CSL(S) = CSL(U)
senderID = userID
```

d. *retrieve secure*

```
CSL(R) = CSL(U)
CSL(R) <= CS(T)
receiverID = UserID
```

e. *access secure*: to obtain mail from buffers.

```
CS(M) <= CSL(process)
access right
```

f. *store secure*: to store mail into a buffer or to display mail on terminal.

```
CSL(process) <= CS(object)
```

g. *forward secure*

```
CSL($F sub i$) = CSL($A sub j$)
encryption/decryption

where i,j are nodes that are connected.
```

h. *Server is secure* if and only if

```
1. login secure
2. reclassify secure
3. send secure
4. retrieve secure
5. access secure
6. store secure
7. forward secure
```

9.1.5 Discussion

Mail Delivery The model guarantees that users may only receive mail that are sent to them. This is done by the switching process and the retrieve process. Furthermore, a sender does not have to identify the security clearance of a receiver when he sends a mail to the receiver. If the sender classifies the mail correctly, the model will prevent receivers with lower security clearance from reading the mail. The model also protects users from exposing their security clearances from users with lower clearances.

The model also guarantees mail delivery if (1) there is a path between the sender's node and the receiver's node; (2) the routing information is available at each node; and (3) the number of undelivered messages is less than the number of messages in the switch buffer and the output buffer.

The integrity and privacy of mail is enforced by the link to link encryption and decryption,

e. Each network user has a unique ID and a password.

f. All devices in the network have comparable security classes.

g. Each server knows how to route mail to proper destination.

h. Reliable information transmission across the network is available.

i. A reliable user authentication mechanism and a good encryption/decryption mechanism are available.

Security Assertions *Mail server security assertions:*

A1. The classification of created mail must be equal to the current security level of sender.

A2. A sender may send mail to another user with security clearance equal to or less than the security clearance of the sender.

A3. Server process may read mail with classification less than or equal to the current security clearance of the process.

A4. The classification of retrieved mail is less than or equal to the current security level of the receiver.

A5. Information removed from an object inherits the object's classification.

Network security assertions:

A6. A user can use a terminal only when his security clearance is higher than or equal to the classification of that terminal.

A7. A terminal may connect to a server if clearance of the connected server is less than or equal to the clearance of the terminal.

A8. A server may forward mail to server of a neighboring node if their current security levels are the same.

A9. Any information transmitted over the network must be labeled with its classification and is in an encrypted format.

A10. Noise is inserted randomly in the network by the server.

Notations

```
Let U  denote user,
    T  denote terminal,
    S  denote server's send process,
    R  denote server's retrieve process,
    A  denote server's accept process,
    F  denote server's forward process,
    SW denote server's switch process,
    SB denote server's switch buffer,
    LB denote server's local buffer,
    FB denote server's forward buffer,
    M  denote mail,

    CS denote object's classification,
    CL denote subject's security clearance level,
    CSL denote subject's current security level.
```

Security Requirements The security requirements of the proposed mail system consists of the following rules. Figure 1 above contains the architecture and the security requirements of a mail and by the good encryption/decryption mechanism. Hence, an eavesdropper may not be able to comprehend content of an intercepted mail.

Covert Channel An eavesdropper may rely on convert channel to deduce useful information. For example, by observing the activity of the network link, an eavesdropper may be able to identify the identity of the sender; by inferencing the buffer overflow message, a user may be able to determine the traffic of a given class of mail in the network; by sending mail with different classifications to a particular user, a sender may be able to identify the security clearance of the receiver.

Since covert channel can not be eliminated completely, the proposed model attempts to reduce the bandwidth of leaked information. The bandwidth reduction is accomplished by the following four mechanisms:

First, all data sent across network link are encrypted at the sending node and are decrypted at the receiving node. By doing so, we can reduce the comprehensibility of an intercepted mail.

Second, noise is inserted randomly into the network by mail servers. The purpose of introducing noise into the network is to confuse an eavesdropper from distinguishing real mail from all intercepted data. In reality, the noise may contain useful information such as network topology.

Third, the model prevents users with lower security clearance from identifying users with higher clearance. This is corresponding with real life situation in which high security clearance personnel will often be aware the identity of personnel with lower clearance but not vice versa.

Fourth, all mail is stored in a multilevel buffer. Hence, if the buffer is full, a sender can not infer the traffic of a given class of mail.

9.1.6 Conclusions

In this section we have examined a secure network mail system model in which the correct mail delivery property is enforced by the access control mechanism, in which the authorized information dissemination property is enforced by the security clearance mechanism, and in which the integrity and privacy property is enforced by the encryption and decryption mechanism. Similar to the Owicki network mail system model, the guaranteed delivery property may be asserted if the number of undelivered mail is less than the number of messages in the switch buffer and in the local buffer and if the routing is possible.

Even though covert channels cannot be eliminated entirely, the model does reduce the bandwidth covert channel. The reduction is made possible by randomly inserting noise in the network, by encrypting mail-in-transit, and by placing mail with different classifications in the same buffer.

The model assumes that the server can be trusted and that mail delivery can be guaranteed by the routing mechanism.

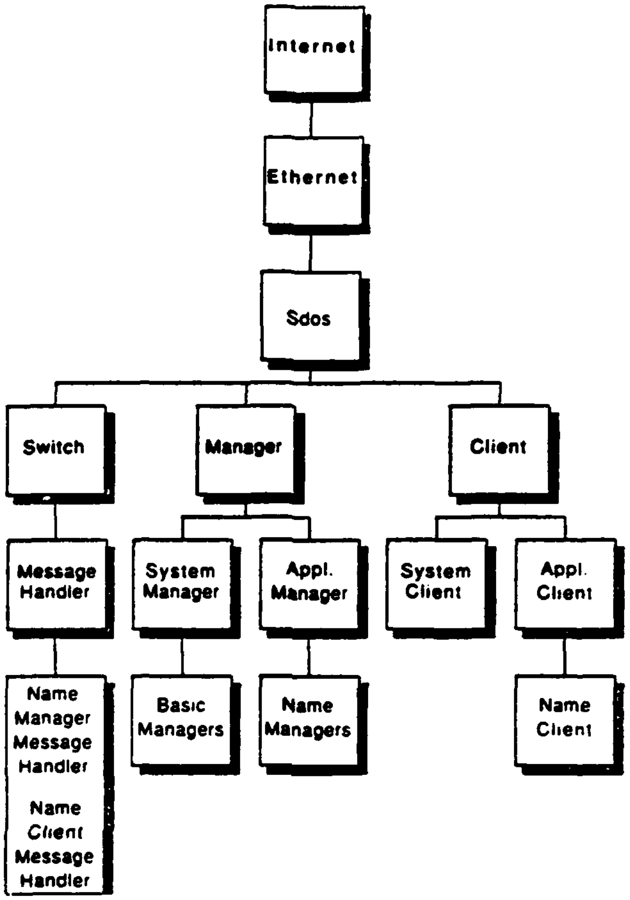

Figure 2: SDOS Object Hierarchy

9.2 Initial Results on Specifying SDOS

We have begun to apply the OBJ-based specification methodology described in Section 4.2 to a number of larger examples. One of these examples is a preliminary OBJC specification of the Secure Distributed Operating System (SDOS) described in [VCH88, TCVW*88].

Given the distributed structure of SDOS, OBJC is well-suited to express its overall specification. To date, we have only a major outline of SDOS specified in OBJC, but what we do have is quite promising as an indication of the practicality of OBJC for distributed system specification.

Figure 2 is a high-level diagram of the OBJC objects that comprise the specification. This OBJC object hierarchy accurately reflects the high-level organization of SDOS as described in [VCH88]. The details of the OBJC SDOS objects are given in subsection 9.2.1. Subsection 9.2.2 contains a discussion of how the OBJC specification of SDOS can be constructed to permit the verifcation of security properties using a state-machine approach, as suggested by the SDOS authors.

9.2.1 High-Level Outline of an OBJC Specification of SDOS

Selected objects from the SDOS specification outline are defined below in OBJC. Refer to Figure 2 for the hierarchical organization of the objects. The syntax and semantics of OBJC were discussed in Section 4.2 of the report.

```
object INTERNET is
  sort Internet .
  protecting ETHERNET .
  op internet : Ethernet Ethernet
     -> Internet .
endo

object ETHERNET is
  sort Ethernet .
  protecting SDOS .
  op ethernet : Sdos Sdos Sdos
     -> Ethernet .
endo

object SDOS is
  sort Sdos .
  protecting SWITCH + MANAGER + CLIENT .
  define Manager-List is LIST[Manager] .
  define Client-List is LIST[Client] .
  op sdos : Switch Manager-List Client-List
     -> Sdos .
endo

object SWITCH is
  sort Switch .
  protecting MESSAGE-HANDLER .
  define Message-Handler-List is
    LIST[Message-Handler] .
  op switch : Message-Handler-List
     -> Switch .
endo

object NAME-CLIENT is
  sort Name-Client .
  protecting MESSAGE[INT] .
  op new-name-client : -> Name-Client .
  op name-client : Message -> Name-Client .
  var N : MessageNum .
  eq new-name-client =
     name-client(send('NameLocal,0)) .
  eq name-client(sent('NameLocal,N,0)) =
    name-client(wait(N,0)) .
endo

object NAME-MANAGER is
  sort Name-Manager .
  protecting MESSAGE[INT] .
  op new-name-manager : Int
     -> Name-Manager .
  op name-manager : Message Int
     -> Name-Manager .
  var N : MessageNum .
  var M : Int .
  eq new-name-manager(M) =
     name-manager(receive('Name,0),M) .
  eq name-manager(received('Name,N,0),M) =
     name-manager(reply(N,M),M) .
  eq name-manager(replied(N,M),M) =
     name-manager(receive('Name,0),M + 1) .
endo
```

Given the object definitions above, the SDOS system behavior will be modeled by OBJC input and output terms of the form shown below. The example top-level input term shows an internet with two ethernet clients, each of which have in turn three sdos processes, with a single active client and manager. The sample switch, client, and manager terms depict further details of the initial input state of the system. Here a simple client-bound message with content "1234" is shown residing in the application manager.

Top-Level Input Term:

```
internet(
  ethernet(
    sdos(switch_c,nil,client),
    sdos(switch(nil),nil,nil),
    sdos(switch(nil),nil,nil)),
  ethernet(
    sdos(switch_m,manager,nil),
    sdos(switch(nil),nil,nil),
    sdos(switch(nil),nil,nil))) .
```

Switch, Client, and Manager Input Terms:

```
switch(
  message-handler(
    new-name-client-message-handler))

client(
  application-client(
    new-name-client))
```

```
switch(
  message-handler(
    new-name-manager-message-handler))

manager(
  application-manager(
    new-name-manager(1234))),
```

Shown below are the resulting output terms when the "1234" message has been transmitted, via the switches, to the application-client. The next message, "1235" is now shown as pending.

Switch, Client, and Manager Input Terms:

```
switch(
  message-handler(
    ncmh(receiving('NameLocal,0),0)))

client(
  application-client(
    name-client(waited((101).NzNat,1234)))

switch(
  message-handler(
    nmmh(receiving('NameGlobal,0),0)))

manager(
  application-manager(
    name-manager(receiving('Name,0),1235)))
```

9.2.2 Verifying Security Properties of the OBJC Specification of SDOS

The primary co-developers of OBJ3, J. Goguen and J. Meseguer, have also done foundational work on the verification of security [GM82, GM84]. In [GM82, GM84] they describe the Goguen-Meseguer methodology for security verification, and indicate how specifications written in a language such as OBJ3 fit into their methodology.

This subsection of the report investigates the connection between OBJC specifications and the Goguen-Meseguer security verifcation methodology. The general propostion put forth here is that if an OBJC specification structured as a secure state machine, then its security can be verified using the Goguen-Meseguer methodology. Details are presented of how to so structure the OBJC specification of SDOS.

Overview Since an OBJC specification is (intuitively) a lattice of objects, we begin by naming and informally describing the important objects in the lattice. Strictly speaking, the SDOS object should be at the top of the lattice; but in order to permit testing and demonstration, other objects are placed above and orthogonal to it.

SDOS is a distributed operating system: it runs on a network of host machines. This does not mean that each host is running a part of SDOS; it means that each host is running a copy of

SDOS and that a job started at one host can run (partially or wholly) on another host. Job sharing requires interSDOS communication, which is something more than interhost communication.

Goguen-Meseguer applies only to nondistributed concurrent systems. This is because a Goguen-Meseguer state machine can only model a single multiuser host — not multiple communicating hosts. In order to overcome this restriction, Goguen-Meseguer is not applied to the top of the lattice, thereby verifying the INTERNET object; it is applied to the sublattice at and below the SDOS object, thereby accomplishing the original goal. Plainly put: the interhost communication provided the communication primitives of OBJC is assumed correct and secure.

User Interface A Goguen-Meseguer state machine executes user commands. For the SDOS state machine, user commands are somewhat subtle. One problem is that the command set of a state machine is constant, but an SDOS user can write new programs that appear to be, or even replace, those originally provided. Another problem is that a state machine (even a Turing machine) cannot obtain additional input during computation. With these problems in mind, consider a user to be a person at a terminal and the command set to consist of the single command

```
    run <user-name> <program-name> <input-data>
```

What are the advantages of this perspective? Does it cause problems? This is considered below.

Logging on and off is no longer a special procedure. A terminal is always connected to SDOS. A session begins with

```
    run <user-name> login <password>
```

and ends with

```
    run <user-name> logout
```

and all user commands between logout and login (as well as a user's second login) are SDOS no-ops. Notice that multiple users can be logged onto a terminal simultaneously. More practically, a terminal can supply the first two fields of every command.

There is only one command and its semantics are constant.

During execution, the state machine does not interact with the user to obtain input data. The data either comes directly from the command line or from within SDOS as addressed by the command line. Such an address denotes data that is either in memory or the file system.

The file system is internal to SDOS. More specifically, every user has their own file system. File ownership is determined by a file's location. Together, the file systems satisfy a security policy at least as strong as that satisfied by SDOS.

Since a state machine has only a single-level command set, there are no "system calls" or "machine instructions" that can be called from a program. Therefore, a user-written program is simply a textual sequence of run commands. This has several ramifications. A user who writes programs has no new method of violating the security policy. Running a data file is no worse than typing the data at the keyboard. A suitable collection of provided programs renders other programming languages strictly unnecessary. More practically, a compiler whose output is a sequence of run commands can be provided or even written with run commands. An alternate approach is to have some programs accessible only from programs and not from the command line; but since all run commands, hence all programs, must satisfy the security policy, this is just an optimization.

A program that is run by a user other than the owner can do what the user, the owner, or some combination can do. This capability is controlled by SDOS and any user that can update the program. Since each command in a program has a <user-name> field, that is the granularity.

Import Lattice The objects at the upper nodes of the import lattice are described here. The objects they import are also listed.

INTERNET This object provides context for communication between hosts.

- HOST

HOST This object encapsulates the components of a host computer: its users and the operating system.

- USER
- SDOS

USER This object models a person at a terminal connected to a host running SDOS. Commands and results are transferred between user and host as OBJC messages. A user behaves as a client that runs programs by making run requests of an SDOS server.

- REQUEST-RESULT

SDOS This object encapulates the major components of SDOS. An SDOS behaves as a server. It accepts, services, and returns the results of run requests from users through the interface subsystem. Communication between user and host is accomplished with OBJC messages. The other subsystems operate as internal servers; they are not directly accessible to users.

- INTERFACE-SS
- SECURITY-SS
- RUN-SS
- FILE-SS
- VAR-SS

INTERFACE-SS This object is the user interface for SDOS. It accepts run requests from users, dispatches internal requests to service them, maintains information about active requests, and returns the result of serviced requests to the users. Before servicing a user's request, the login status and security level of the user is requested from the security subsystem. Requests are:

```
op run : UserName ProgramName InputData -> .
```

SECURITY-SS This object maintains security information about users and services requests concerning it. This information is initialised during startup. Requests are:

```
op login : UserName -> .
op logout : UserName -> .
op get-login : UserName -> .
op get-level : UserName -> .
```

RUN-SS This object runs programs. A program is either a builtin program or a file of run commands. The search-path variable resolves ambiguous program names. Requests are:

```
op run : UserName SecLev ProgramName InputData -> .
```

FILE-SS This object maintains user files. Each user has a separate set of files. Subject to security, a user can access another user's files by qualifying a file reference by the owner's name. Requests are:

```
op get-file : UserName SecLev FileName FilePos -> .
op put-file : UserName SecLev FileName FilePos FileData -> .
op rem-file : UserName SecLev FileName -> .
```

VAR-SS This object maintains user variables. Each user has a separate set of variables. Subject to security, a user can access another user's variables by qualifying a variable reference by the owner's name. Requests are:

```
op get-var : UserName SecLev VarName -> .
op put-var : UserName SecLev VarName VarData -> .
op rem-var : UserName SecLev VarName -> .
```

9.3 A Security Kernel for Distributed Systems

One of the longer-term research efforts we would like to undertake is the complete formal specification of a security kernel for a distributed computer system. This kernel will be the software portion of a trusted computing base satisfying the criteria for an A1 level secure system as defined by the Department of Defense.

In this section of the report we present our ideas on this kernel. We begin with a survey of related work followed by the discussion of our proposed kernel.

The aim of the the kernel is to provide a set of primitive services that manage resource allocation, communication, and process manipulation while ensuring the security properties of the system. Using the kernel, the system developer will be able to create software systems that are guaranteed to satisfy formally specified security policies.

9.3.1 Overview of Previous Research

Since distributed operating systems and computer system security are such important subjects, there has been a lot of research in these areas. This subsection surveys several of these systems in an attempt to give an overall view of the areas that have currently been explored.

Distributed Operating System Kernels One type of computer operating system is the distributed operating system. The following two systems are examples of Distributed Operating System Kernels. These systems help enumerate the particular issues being evaluated using this type of system:

- Security. In these systems security is considered a side issue. It appears that the other issues take precedence and once they have been developed into a standard methodology then security will become more central. Until then data integrity and minimal access controls are all the security seen in these systems.

- Integrity. In a distributed environment, where data caching is implemented for performance improvement and resource sharing is permitted, we need to worry about the integrity of the data. These systems try to develop methods for insuring that the data remains consistent yet performance is not hindered. The Alpha system even goes so far as to limit the loss of data when a portion of the system goes down.

- Resource Management. In a distribute system there is a wide variety of resources that must be managed fairly and efficiently. These systems try to develop the best methods for handling the resources.

- Support: These systems are very different in the applications they support. The Alpha kernel supports real-time applications developed for it specifically, where the V kernel provides a set of language tools to allow a wide variety of applications to run on it.

- Performance. Performance is one of the key issues in this area. The ability of the kernel to respond quickly to the dynamic nature of the distributed system, servicing requests and scheduling processes in a fair and efficient manner is addressed in both of these systems.

Alpha Kernel The Alpha Kernel [Nor87] is a decentralized[1] operating system kernel for a real-time command and control system. It is the initial phase of the implementation of the Archons projects at Concurrent Computing Corporation led by E. Douglas Jensen.

The kernel executes as a set of identical processes running on a system of connected homogeneous hosts. The hosts are meant to number from 10 to 100 and be separated by no more that on the order of 1000 meters. These processes cooperate to present to the client processes the appearance of a single computer, completely hiding the decentralized nature of the system.

Currently, to keep the project manageable, security has not be addressed in the Alpha kernel. It is a concept that will soon be looked at.

This kernel is meant to support a system that provides services for a single real-time command and control application. There is no concept of multiple users or a diversity of processes. The single application executes as a set of threads running through several objects (possible concurrently). These threads have real-time constraints on their performance that the kernel must be able to support.

V Distributed System The V distributed system is part of an ongoing research project to study issues in distributed systems. The project is lead by D.R. Cheriton at Stanford University and has produced numerous thesis topics and results over the last several years. It provides a "hands-on" capability for testing new methodologies and experimenting with different system parameters [Che88].

This system is based on a collection of workstations connected by a high-performance network. The primary configuration is that of diskless workstations connected to a set of file servers by the network [CZ83]. This configuration allows a process to be scheduled on different hosts during its lifetime. Since the process gets swapped out to a shared file server when not being executed, it can easily be swapped in on another machine thus "migrating" the process among the machines. The scheduler that handles this migration is designed to prioritize processes and then schedule the top n processes on the n machines in the system.

Although mentioned briefly in the literature, there appears to be no overt attempt to ensure security in this system. The main emphasis seems to be placed on performance measures and efficiency. Since the system is part of a continuous project, and the specifications of the system change with time, it seems that it would be very difficult to ensure anything more than some rudimentary security properties.

The system is designed to provide a high level of performance to a collection of applications developed using kernel specific run-time modules for popular programming languages (ie., Pascal, C). The system also supports co-location of servers with the hardware they serve to reduce communication slow down and thus permit the use of real-time control applications.

This system emphasizes issues that are also important to the kernel proposed in our research. The major difference is that the V system does not have a formal sense of security that is crucial to this our research.

Security Kernels Another form of secure operating system is the security kernel. This system is based on using a minimum set of secure operations in the kernel. These operations are then used to build the overall system.

[1] The author defines the system as decentralised to avoid any ambiguity with the popular term distributed. The system is decentralised since it executes as cooperating processes on physically separated hosts

The following systems are all examples of Security Kernels. These systems help enumerate the issues being explored in this type of system:

- Security: Each of these systems tries to maintain a certain security policy. This is the reason for the existence of the kernel and thus is the most important issue here. These kernels try to separate "Trusted" and "Untrusted" processes. The trusted processes are verified to maintain the security of the system and thus avoid the internal checks performed by the kernel on operations of untrusted processes.
- Integrity: Data integrity is a concern in these system. They use different techniques to maintain that changes to system data is a result of verified or trusted operations.
- Resource Management: These systems each take different approaches to management of the resources of the system. The differences occur in how the resources are allocated, how the access to the resources is managed and exactly what the resources are.
- Support: Another issue being looked at in these systems is support. In the systems below three of them are designed to support a UNIX-like interface executing on top of the kernel, the other system is meant to be a secure execution environment for ADA.
- Performance: Again performance is considered a major issue where the implementation of the kernel tries to keep the system efficient.

KSOS In the late 1970's there was a project at Ford Aerospace to develop a provably secure operating system [MD79]. This system, Kernelized Secure Operating System (KSOS), is intended to be a fully functional multilevel secure operating system with a Unix-like interface for larger minicomputers.

The system is based on the security kernel concept. The kernel provides the basic operating system functionality, and is enhanced by the inclusion of trusted and untrusted system support processes. These processes execute in three different modes.

In the kernel mode is the security kernel. This kernel must be a fully verified multilevel operating system kernel. It has complete access to the whole system and provides the interface between the system and the client processes. Each invocation of an operation supported by the kernel will be executed only if the calling process has the correct authorization to execute that operation. In other words, the invoked operation has to comply with the implemented security policy.

In the supervisor mode there is the Unix emulator (an untrusted process) that translates Unix-like system calls into kernel calls on behalf of the original invoking process. Also at this level is the trusted non-kernel security related software (NKSR). These processes are completely verified routines that perform necessary security related operations that possibly violate the strict definition of the security policy.[2]

This system provides a basis for execution of Unix processes in a secure environment by logically separating processes of different security classifications onto separate virtual machines. It also permits the development of different execution environments by allowing server processes that execute in the supervisor mode to emulate those environments.

[2] KSOS implements a secure file server at this level. The server stores files at several different security levels so it must be cleared for the highest of these levels. It has to permit lower level subjects to read their files and thus violates the *-Property.

Although this system supports data security and separation using the operating system kernel model, it does not support any concept of distributed programming, but rather connects the machines through standard network protocols.

LOCK (was SAT – was PSOS) The LOgical Coprocessing Kernel (LOCK) is a current research effort by Secure Computing Technology Corporation to develop a hardware-oriented solution to the problem of multilevel secure computer system development. The project is based on the SRI specification of the Provably Secure Operating System (PSOS) [NBF*80]. The hardware portion of the Trusted Computing Base specified by a Ford Aerospace attempt to implement PSOS [FOR80], was picked up by Honeywell in 1982 as the Secure Ada Target project (SAT). LOCK is the third phase of the SAT project to generate a "detailed design specification and a secure microcomputer prototype by 1990" [SK86].

The LOCK system is designed to attach a generic coprocessing unit (SIDEARM) onto a microcomputer bus. The unit will act as reference monitor for the CPU and its resources. It consists of one to four microprocessors with their own volatile and non-volatile memory and a separate long term storage device.

This system will maintain the non-interference model of Goguen and Meseguer [GM84] with respect to the TCSEC A1 level specifications. This is done through the management of the reference monitor and additional two-level encryption of secure data. This encryption enables the system to use standard hardware for communication lines and long term storage by ensuring that the data going on these devices is unusable to any unauthorized process or person.

The system is designed with a generic interface that will attach to a small machine-specific device called the host interface connector. This device and its functionality will have to be proven for each machine. But, since it is a small interface not much difficulty is seen here.

The initial prototype of the LOCK system will retrofit Kernel Extensions of the system into the security relevant portions of a UNIX operating system. The idea is that with this small adjustment (and a few others not yet determined to the operating system, the addition of the SIDEARM will create a secure system with little performance degradation (performance should be 80% of the original system).

This system is specifically designed for a single host with little mention of connecting it to a network, and no mention of a distributed system. [3] Placing a lot of the security relevant functions in hardware is a good way to improve performance and ensure reliability and integrity. With recent developments in microprocessors that have built in resource management units, such as the Intel 80386 [INT86], all this work may be overkill and actually perform less efficiently than a software based system that uses the microprocessors full potential.

Secure Xenix The Secure Xenix system is an "experimental system designed to run on IBM PC/AT workstations" [GCC*87]. This project is headed by V.D. Gligor at the University of Maryland.

The system is designed to meet the specifications of B2-A1 level as defined by TCSEC. To meet this classification level they add an Access Control List mechanism to the standard UNIX interface. This mechanism allows the system and users to implement mandatory and discretionary access controls and maintains the required audit functionality.

[3] Combining this system with the THETA system mentioned later may be an effective way to develop a secure distributed system if the interface between the two is similar enough.

In addition to the added access control the system includes a Secure Access Key (SAK) mechanism where, by a few special keystrokes, the user ensures an authenticated communication path directly connected to the kernel. This will permit the user to perform operations that may violate a precise interpretation of the security policy but not the spirit of it, and also to change their current classification level.

To maintain the security policy in the standardized UNIX file system, they create virtual directories for each security classification. Thus when a UNIX process writes to, or is swapped to the "tmp" directory, the data is considered secure.

The system also separates information by creating a separate process for each command of administrative users. These processes are further divided into trusted processes that do not require superuser privileges and those that do.

Overall, this system seems like a good concept, but they are trying to retrofit security into an existing system design. Regardless of the system, the designers inevitably have to "hack" something to make up for an assumption the original system made[4]. Such unconventional modifications will make the design and proof of the system much more difficult.

UCLA Security Kernel The UCLA Data Secure Unix operating system was developed in the late 70's as a demonstration that general purpose functionality and verifiable data security are attainable [PKK*79]. The UCLA Unix architecture is based on DEC PDP-11/45s and PDP-11/70s. The kernel executes as a single process in the hardware kernel mode on the host machine managing all client requests for machine resources.

Sitting above the kernel and executing in the hardware supervisor mode are a collection of modules. Two of the modules, the File Policy Manager and the Dialoguer are trusted processes. The other modules are instances of the Kernel Interface SubSystem (KISS).

The Dialoguer is a process to which the user terminal can be reliably connected. This connection occurs whenever the user types a predefined break sequence. Now the user and system can perform reliable authentication and modification of data security levels.

The File Policy Manager implements a shared file system while maintaining the security policy. All requests to the file system are sent to the kernel which then forwards them to the Policy Manager and passes its response back. Since the kernel manages all hardware access and thus defines memory as a collection of virtual memory pages, the Policy Manager implements the file system using these memory pages.

The KISS is a general interface to the kernel. It creates a more usable interface between general processes and the kernel. These processes do not directly communicate with the kernel at all. Each KISS is a separate interface for one process such as the scheduler, network interface nucleus or a Unix interface emulator. Each of these higher processes can then be interfaces to other processes running in the hardware user mode such as Unix user processes, or the Network Manager.

The security policy implemented by this kernel and the trusted processes is a capability based policy. All operations, whether memory or I/O related are governed by a central capability mechanism. Since I/O in the PDP-11 family is actually memory based and Unix devices are special files, all operations must go through the Policy Manager. Success of all operations revolve on whether the process has been granted the capability for that operation on that object. The granting of capabilities is actually handled only in the File Policy Manager and not in the kernel.

This system is a fully functional, verified operating system [Kem79]. Although the security policy is actually implemented outside the kernel it seems to maintain a fine grain protection

[4] The creation of the virtual directories is a good example of a hack

without loss of integrity or much performance degradation. Since the Policy Manager is the only process with the ability to Grant capabilities this system may prevent some operations that would be valid in the TCSEC specification. This limitation was left in place to simplify the proof and remove several problems found in other capability based systems.

Secure Distributed Operating Systems The following systems are examples of Secure Distributed operating Systems. They help enumerate the issues involved in this type of system:

- Security: Again security is a primary issue and the reason these systems exist. They are designed to provide a lot of functionality while still maintaining the security policy. Here an important issue is the model used for the system. Both of these systems suppose that a constituent operating system is executing on each host. THETA assumes that each host system has been verified as a multilevel system for that single host. The other system assumes the host systems are untrusted and thus each host is classified at a single security level. Assuming this they then develop different methods to connect the individual hosts together.
- Integrity: These systems let the constituent operating system worry about data integrity. Both ensure that data integrity during communication between hosts is maintained although THETA assumes the network is already in place that can do this.
- Resource Management: These systems both let the constituent operating system on each host actually control the resources. They then provide services to client processes to connect them to the resources.
- Support: These systems support the constituent operating system on their host and then the standardized protocol between hosts. Each system allows processes to execute on the host that do not communicate with the system in any way.
- Performance: Performance is an issue that must be addressed, although in these systems the tendency is to assume that the constituent operating system on each host will be able to manage the additional load with little performance penalties.

Distributed Secure System A cost effective and highly efficient system has been proposed by Rushby and Randell [RR83]. This system is composed of a network of standard UNIX systems connected thorough the "Newcastle Connection" and small trusted hardware devices.

The system is effectively a distributed multilevel system which acts, from the users vantage point, as a multilevel single host system. The main technique used in this system is that each host actually runs as a singlelevel system, thus implicitly maintaining the mandatory access controls.

The individual hosts are connected by a system called "UNIX UNITED" [BMR82] through a local area network. Between each host system and the is the heart of this system the "Trustworthy Network Interface Unit" (TNIU). The TNIU unit is a hardware system that guarantees via label checking and encryption that messages sent across the network maintain the mandatory access policies. In other words, a machine classified as Top-Secret can not send any messages to a machine with a lower security clearance such as Secret.

This system supports single level hosts and connects them together to develop a distributed multilevel system. It supports whatever constituent operating systems exist on the hosts. The distributed nature of the system is limited by the inter-host communication protocol established by the TNIU.

The TNIU is a very good idea in that it develops a secure network interface and network communication by encrypting the messages in the hardware before sending them out. Our proposed kernel will use this idea (slightly modified) to establish a trusted communication path between separate hosts.

Trusted HETergeneous Architecture (THETA – was SDOS) The THETA system is an experimental prototype for a B3 level *secure distributed operating system* [VCH88, TCVW*88]. The system is designed to allow connection of a network of hosts with a heterogeneous hardware and software base.

The main premise behind the THETA system is that the network meets the requirements for B3 level classification, and each host runs a secure multilevel constituent operating system (COS) which also meets the requirements for B3 level classification. Using these assumptions and the "hook-up" requirement of McCullough [McC87] THETA combines the COSs and network to provide a B3 level distributed operating system.

THETA exists on the host machine as a collection of programs running as separate processes on the host COS. Using THETA, a client process can either execute on the host COS or as a client of an THETA process. Either way, all of the local operating system functions are performed by the COS. But, whenever a client process wants access to an THETA object or operation it must go through the THETA object manager process. The COS guarantees that the local objects and operations provided by the THETA processes are protected from use by any non-THETA processes.

One of the THETA programs executing on the local host is called the switch. This is a multilevel process and exists on each host. All operation invocations must be processed by the local switch. If the operation involves communication with a remote host then the switch handles the necessary network communications and eventually returns the reply to the invoking system manager to pass on to the client.

Since THETA runs as a client on the host COS it should be much easier to port the programs to new architectures. The verification of each port may be limited to some machine/COS specific code with a large portion of the THETA code untouched. This means that the verification of the new THETA system should be much easier that the initial verification of the whole system. Unfortunately THETA does assume the existence of a verified single host COS existing on every system.

As the previous system, THETA is limited to the security level of the COS. Communication between THETA clients and objects is managed by THETA but is restricted by the abilities of the COS. Although THETA appears to have more of a distributed nature and includes security as a primary focus it is a fuller, and thus more limited system than a kernel would be.[5]

9.3.2 Our Kernel

The kernel proposed here targeted for a distributed operating system that will meet the A1 classification level as defined by the TCSEC. It executes as a separate process on each host and communicates through a trusted network to the kernels on other hosts. Each of these kernels can be viewed as a general resource manager, such that a client process executing under control of the kernel uses the operations provided by it to access any resources needed. The manager will control access to all resources and guarantee that the security policy is maintained during these

[5] A fuller system has already made several protocol and design decisions which limit the flexibility of the system.

accesses. To develop this secure general resource manager as implemented by the kernel we must first understand the general classes of resources which this manager must control.

Resources

- *Single-Process Non-preemptive*: This resource type is allocated to a single process at a time such that it has exclusive access to that resource until it deallocates the resource. Examples of this type of resource are a terminal, printer, and outside communications port.
- *Single-Process Preemptive*: These resources are also allocated to a single process at a time, but the process may temporarily loose control of the resource to another process. Examples of this type of resource are a CPU, and some coprocessors.
- *Multiple-Process Non-Preemptive*: These resources may be allocated to more than one process at a time. This may be accomplished by partitioning the resource into multiple single-process subresources or simply sharing the same resource. The processes are guaranteed access to the resource until they deallocate it. Examples of this type of resource are long term storage devices, message passing channels (possibly limited to two processes), and virtual memory.
- *Multiple-Process Preemptive*: These resources are also allocated to multiple processes at a time. The difference here is that the process may temporarily loose control of the resource to another process. An example of this type of resource is physical RAM in a page swapping environment.

Resource Manager Now that we know the types of resources that the manager must control we should determine what the resource manager actually does. According to the TCSEC specification, at the higher levels of classification the manager must contain only security relevant operations. It must also manage the security policy of the system including auditing controls. Following this definition and the description of the resource types given above we see that there are some immediate considerations that the resource manager must address.

1. For the preemptive resources we must develop a scheduling policy, possibly a separate policy for each resource. For example the physical RAM would probably be controlled by a scheduler to minimize page swapping, while the CPU scheduler would probably be a circular scheduler with possibly multiple priority queues to ensure fairness.
2. Security of resources is also important. Each resource must be classified at a certain security level and this level may be changed during the lifetime of the resource. If use of a resource may possibly violate the security policy then it must be monitored by a trusted resource server. This is obviously a security policy concern and thus is managed by the resource manager.
3. Duplicate resources should also be considered for the preemptive resources. There may be multiple copies of a resource or a resource may be partitioned into identical subresources. In either of these cases a preempted user should be able to be rescheduled to any of the identical resources with no difficulty or detectable difference. Although this appears primarily to be a performance issue, knowledge and control of individual resources may be a source of covert channels and thus should be handled by the resource manager.
4. The resource manager must handle the audit mechanisms as specified by the TCSEC.

Model Given the above requirements for the resource manager the kernel can still implement it many different ways. Each of these implementation methods is just a different model of the resource manager but result in the same overall functionality. The following list consists of the models that are currently being considered.

- *Separation Kernel:* The kernel provides a basic set of services to allow the development of several virtual machines. Each machine will be a virtual distributed entity that is allocated a set of private resources (storage) and is assigned a security classification. All users and processes using this machine will be cleared for the same classification level as the machine. A Need-to-Know policy will be implemented on each virtual machine with a mandatory policy governing communication between machines and to peripherals.

- *Policy Kernel:* This is currently a separate idea, but may be eventually merged into the above concept. A policy kernel provides a basic set of services that permits the policy manager to develop a statically defined security policy for the system. The services are general enough that any policy can be implemented using them. The security verification of the policy would then be made easier by using the services as secure high-level operations.

- *Capability Kernel:* Here the kernel is similar to the UCLA Security Kernel. It provides simple operations that allow the user to manipulate capabilities. Thus, the capability to read or write an object has to be explicitly granted to the user allowing the system to directly control the rights given to a particular user.

- *Access Control Kernel:* Here the kernel maintains the access control lists for all objects. Requests to access an object must be granted by the kernel. Anyone with the appropriate privileges in the access control lists may modify the access rights to an object. Here we control the distribution of data by guarding these access privileges.

Basic Assumptions Regardless of the model used, the development of the kernel as a resource manager will incorporate the following assumptions:

1. *Trusted Users*: It is assumed that the users of the system are trustworthy up to the level of clearance they have obtained. These users will not deliberately violate security properties at this level.

2. *Trusted Configuration*: It is assumed that the configuration information, security classification of devices and users, is accurate. The system will guarantee that only a trusted security officer will be able to modify the configuration information.

3. *Trusted Identification*: It is assumed that there exists some trusted method of verifying the identity of the user at login time.

4. *Network Security*: It is assumed that the communication network is Secure from an active wiretapper. Rushby and Randell [RR83] developed a Trusted Network Interface Unit (TNIU) to insure secure network communication. A similar hardware device, with slight specification changes could be used here. The communication will probably be based on public key encryption to permit a known host to come on-line and be authenticated.

5. *Physical Security*: It is assumed that the physical security of devices is guaranteed. Terminals, processors, servers and various peripherals are physically located in areas whose security level is that of the levels of the devices. (ie., an outside modem line could not be considered anything but Unclassified while terminals in the Pentagon Security Room may be Top Secret). These devices must also have been cleared to their clearance level.
6. *Hardware Integrity*: The hardware used is guaranteed to operate according to its specifications.
7. *Processor Support*: The processors used must be able to support virtual memory capabilities, and separation of memory addresses. We do not want an untrusted user program to gain access to any resources not allocated to it by the security kernel.

Kernel Services In this section I define the basic services provided by the kernel that are available to the general user. Since the distributed nature of the kernel is transparent to the user, the distributed environment services are not available to the general user.

The following is a list of possible services that the kernel should provide. The kernel maintains strict control over the physical parameters of the system. Although this is not necessarily a complete list, it is pretty comprehensive. Some additional services may be added if the kernel model is an Capability or Access-Rights kernel to manage these additional resources.

- *Resource Allocation*: Exactly how this is performed and what the allocation actually means depends on the type of resource. For example the kernel will allow users to allocate and deallocate blocks of memory. All references to this memory will be relative to the user's memory space. Also under consideration is the idea of allowing the user to define memory segments, each of which will be a separate virtual entity with associated privileges. This would allow for the creation of virtual machines in which user processes will be confined.
- *Resource Read and Write*: These services allow the user to communicate with any resource allocated to the user.
- *Process Control*: The kernel will allow users to create, and destroy processes. Currently the kernel will manage process scheduling, but the scheduling of user processes may later be relegated to a specific (possibly unverified) user supplied system process. If so, the security of the interface will have to be verified.
- *I/O Control*: The kernel will provide services to handle I/O to peripheral devices. There will probably be two methods of handling I/O. The first will be a set of "trusted" device drivers that are accessed through specific calls and interrupts. The kernel will ensure that the privileges assigned to the devices are not exceeded by the data sent to the driver. The second method would be to allow a user to exclusively control a device – this method is still under consideration.
- *Communication Control*: The kernel will provide services that will carefully monitor all inter-process and inter-machine communications. The exact mechanism and policy for this is still under consideration. The final services will maintain the security policy but may be more restrictive. This communication will consist of intra-system communication between processes in the same distributed system and inter-system communication between processes in different distributed systems.

9.3.3 Summary

Kernel The kernel will execute as an independent process on each host. It will have complete control of the hardware and system resources. When connected to a secure network[6] the kernel will communicate with other identical kernels such that it will be able to access the resources the other kernel manages.

This kernel will provide the basic functionality expected of an operating system. This will include process creation, deletion and scheduling along with resource management. The kernel will implement a resource manager by providing services to client processes that will permit them to gain access to the resources and use them.

The kernel will maintain the security policy of the system. In the most likely implementation, objects and subjects will be classified with respect to a particular security compartment. Within a compartment, the rights for a subject to access an object will be explicitly specified. No subject outside the compartment will have rights to the object.

The primary research issues involved here are summarised below:

- Security: This issue is by far the most important one here since this is why the system is being designed. Here I will assume that if each kernel on an individual host can maintain the security policy, if the kernels communicate using a standard trusted protocol and the communications network is secure, then we can ensure that the security policy is maintained over the whole system. This concept has been formalized by McCullough [McC87] and is used in the THETA system.
- Integrity: Data integrity is an essential part of computer security and thus is also a key issue in this system. I intend to design the system to guarantee changes in data are authorized and loss of data due to node failure is minimal.
- Resource Management: As seen earlier, the kernel is effectively an implementation of a general resource manager. The kernel will have complete control of the resources and provide services to the client processes for use of these resources.
- Support: This general services provided by this kernel can be used by client processes to develop an execution environment that is compatible with popular environments that currently exist (ie., UNIX).
- Performance: Although performance is an important issue, this system will sacrifice performance to ensure the security policy if no more efficient method is available.

Specification and Verification The details of the system will be specified by defining the objects of the system and the operations performed on those objects. These objects will include system specific objects that are not available to client processes. The specification will include all the information necessary to implement the TCSEC security policy but not restrict the implementation to a particular method.

The verification will include proof that specified kernel objects maintain certain security policies. These policies will include those specified by the TCSEC [DOD83] and probably the "hook-up" properties of McCullough[McC87]. These properties ensure that secure single host systems are connected correctly to form a secure multi-host system.

[6] A secure network is considered one which guarantees that messages are delivered only to the proper host and that active wiretappers can not interfere with or understand the messages.

Beyond this verification the system will have to be analyzed to determine if any covert channels exist. If any are found that can not be removed, the bandwidth of the channels must be determined and possible methods to reduce the bandwidth will be presented.

Overall, a system as large and complex as a security kernel will provide [illegible]cellent test bed for the specification and verification of secure distributed systems. In addition to research results on the kernel itself, the kernel project will drive our methodology and tool research.

10 Conclusions

The goal of this project was to investigate techniques for the verification of the security of distributed systems. In carrying out the work, we surveyed the following:

- Techniques for the specification and verification of concurrent and distributed programs
- Existing mechanized theorem provers, mostly to evaluate their suitability for verifying concurrent and distributed programs
- Work towards the development of exemplary secure distributed systems, even in cases where verification was not a design goal

An outgrowth of our work was a design of an environment suitable for the specification and verification of secure distributed systems, and descriptions of several distributed architectures that would be amenable to verification.

Our conclusions consider the following:

- The need for secure distributed systems
- A security model appropriate to distributed systems
- A suitable abstraction of a system design that is verified with respect to the security model – A1 certification
- The feasibility of carrying the verification below the level of an abstract design, moving towards beyond-A1 certification
- Mechanizations of the verification process using extensions of current theorem provers.
- Exemplary distributed systems that could serve as vehicles for an initial attempt to produce an A1 (or beyond) distributed system.
- Other research topics in support of secure distributed systems.

Informally, a system is secure if the information it stores is protected against release or modification by unauthorized users. The Multilevel Security Policy (MLS) as described in the Orange Book [DOD83] associates security levels with users and objects, and requires that a user gets to see the contents of objects at his or lower levels; that is information can flow to higher levels but never lower levels. A more abstract model of security that avoids the need to consider objects has been formulated by Goguen and Meseguer [GM84][GM82], Feiertag, Levitt and Robinson [FLR77], and McCullough [McC87]. In these models the information a user observes is to be dependent only on the actions of users at his level or lower. That is, the actions of higher level users cannot be observed by lower level users. This is the requirement of a MLS system, including a distributed system. Generalizations of MLS can also be modeled with these systems through the use of event systems whose elements are input, output or transition events. The events any user sees can be specified as a function of his *view*, a view being his security level, his group, what objects others have granted him access to, etc.

The burden of security falls on the operating system, although appropriate hardware support can minimize the impact of security features on system performance. The LOCK system is an example of a system with extensive hardware support, although all systems have some hardware support, e.g., a memory management unit.

In simple terms an operating system that satisfies the MLS policy (or other policies) must enforce access control: processes have access to objects in accordance with the security policy. In addition, the operating system itself must not be a channel for the communication of information not in accordance with the security policy. Such unwanted information flow can potentially occur through objects managed by the operating system and to which more than one-users have access; the term covert channel is often used in referring to such objects.

There have been attempts to develop systems that implement the MLS policy, most for single host/multiple user systems such as mainframes or shared workstations.

In recent years there has been increasing interest in distributed systems and, naturally, in secure distributed systems. For our purposes in this report a distributed system consists of hosts (which could be workstations, mainframes, personal computers), servers (which could be repositories for objects accessible to multiple hosts, such as files, directories, names, passwords, etc.), and a network through which the hosts and servers communicate with each other. Security is perhaps more important for distributed systems in that such systems are likely to have many hosts and many users with different authorizations. The security policy for a distributed system is identical to the (abstract) version of MLS: A user is not to observe the actions of users except those at its level or lower. A user can be associated with a host or might be an eavesdropper on the network – passive or active.

There have been several proposals for secure distributed systems, and they vary according the services offered by the system. In the simplest case, each host can support a single user or, slightly more generally, users operating at the same level. In this case, Rushby and Randell [RR83], the burden of assuring security falls on the network, which can mediate all communication between hosts to assure only those intended to communicate with each other do so. Indeed, the fact that the users are permitted to communicate only through a few well defined interfaces makes the attainment of security for this simple (albeit useful) distributed distributed system easier than for a multiuser mainframe. More complex distributed systems would include multilevel file servers, as proposed by Rushby and Randell.

A more general distributed system would support multilevel hosts. More complex distributed systems permit the sharing of memory or of hosts across the system; hosts could be shared through process migration. For such systems the attainment of security requires secure hosts in addition to secure interhost services.

The Orange Book defines a rating for a secure system according to the system's services in support of security and the extent of the certification of the system with respect to a security policy. The highest ratings, A1 and beyond-AI, are granted to systems that have been formally verified to satisfied the policy. A1 certification requires that the system design be verified, while beyond-A1 is more stringent in that it requires the verification of the system's implementation. For a single-host system the design is considered to be the specification of the functional behavior of each service provided by the system, e.g., system calls and ordinary instructions accessible to user processes. A few systems have been verified according to the A1 criterion. We are aware of no distributed systems that have been certified to be A1. A goal of this work was to make progress towards an A1 distributed system, through the definition of an interface for a secure distributed system and a specification of a design.

Once the interface specification of a system has been verified, it remains to verify the implementation. Of course, a system cannot be considered to be verified unless the executable code is verified. However, many errors that could render a system insecure can be eliminated through verification of design decisions that can be formulated in stages of development well before code is produced. One approach to staging the development and verification of a distributed system is to consider it to be the interconnection of large self-contained components, each of which has a specification. McCullough has developed a methodology for the verification of systems at this level, which he calls hookup verification; the security policy is called restrictiveness. Through the methodology the hookup of a set of components is concluded to be secure provided each of the components (represented by its specification) is verified to satisfy restrictiveness and their hookup is shown to satisfy particular properties,

Our work extends, mechanizes, and applies the McCullough methodology. Our extension involves writing specifications for a class of generic building blocks, the class including filters of various kinds. Next we define a large class of components that are instantiations of these filters and show that the instantiations satisfy restrictiveness. These components include queues, transformers, multiplexors, de-multiplexors, and switches. A secure distributed system can be configured using these components as building blocks, these blocks providing the links through which untrusted components communicate. To demonstrate the utility of the methodology, we show how a set of our components can be connected to produce a verified Rushby-Randell distributed system design.

All proofs are mechanically checked using the Higher Order Logic (HOL) system developed at Cambridge [Gor88]. Of the existing mechanical proof systems, we found HOL to be the best suited to reasoning about the interconnection of components with respect to a security policy. HOL was selected for this project based on its support for higher order logic, generic specifications, and polymorphic type constructs - all in support of writing and reasoning about general classes of components.

Once the interconnection of a system of high-level building blocks has been established to satisfy the security policy, it remains to verify the implementation of the blocks. We found it convenient to divide the implementation of the basic building blocks into two basic abstractions: (1) the low level mechanisms that provide the abstraction of processes, especially process separation, and (2) secure applications that involve the interaction of processes. The first abstraction is successfully achieved through conventional secure operating system design and verification.

The second is best achieved through programming with a concurrent programming language, and consequently requires reasoning about concurrent programs. Our approach to the verification of concurrent programs involves reasoning about the specifications of interacting programs and then verifying the specifications. For the former, we favor algebraic specifications in the style of OBJ, but reflecting the concurrent activity. We have developed an extension to OBJ (called OBJC) which uses rewrite rules to represent concurrent programs that share state. Security properties, among others, can be expressed in OBJC and the OBJ rewrite rules can be shown to satisfy such properties. Although any theorem prover could be used, we have found it convenient to build the theories necessary to accomplish proofs using OBJ.

To carry the proof down to the level of executable code, we suggest using a concurrent programming language that runs on the process separation kernel; although we have yet to establish the formal connection, the kernel can be the runtime system for the concurrent programming language. We suggest using the language SR (Synchronizing Resources) for the programming of secure applications. Towards demonstrating the utility of SR we have developed its axiomatic semantics and showed how it could be used to verify specifications of concurrent programs expressed in the algebraic style of OBJ.

The main focus of the project was on the verification of secure distributed systems. However, we also considered two aspects of secure system development that are important in the general context of software engineering. These are the testing of secure systems and the rapid prototyping of such systems.

Regarding testing, we have developed a planning system for the intelligent testing of secure software. The system, called TPlan, is based on the ideas of STRIPS and GPS. A goal is presented to TPlan that represents a particular state one hopes the program under test will not enter; with respect to security, the state is one where an user has unauthorized access to objects or where information has leaked through covert channels. TPlan can do an exhaustive search, but can take advantage of heuristics such as: particular operations that might represent flaws in the system and how many processes should be involved in exposing the flaw (each process requiring a context switch to become active). TPlan produces a plan which can be interpreted as a test sequence that exposes a flaw.

Regarding rapid prototyping, we have noted that many operating systems share structure. We have developed a template for the rapid prototyping of operating systems, the template consisting of 4 levels of specification in an executable specification language. Using the template, a designer can propose policies for access control, process switching, scheduling, memory and file management in terms of rules that are interpreted as programs. Currently, we are using our template to produce a rapid prototype of the MINIX operating system. The implementation of MINIX requires 12.000 lines of C and assembly code; our rapid prototype will require approximately 1000 lines of specification.

References

[A*84] G. Almes et al. Edmas: a locally distributed mail system. *International Conference on Software Engineering*, 56–66, 1984.

[A*86] G.R. Andrews et al. An overview of the SR language and implementation. *Journal of Distributed Computing*, 1:133–149, 1986.

[AA87] M. Abrams and A.K. Agrawala. Automated measurement and prediction of unconditionally synchronizing distributed algorithms. *IEEE Conference on Distributed Computer Systems*, 498–505, 1987.

[AD83] K.R. Apt and C. Delporte. An axiomatization of the intermittent assertion method using temporal logic. In *Automata, Languages and Programming: lncs, Volume 154*, pages 15–27, Springer-Verlag, 1983.

[AF*87] D.P. Anderson, D. Ferrari, et al. A protocol for secure communication and its performance. *IEEE Conference on Distributed Computer Systems*, 473–480, 1987.

[AK86] B. Auernheimer and R. Kemmerer. RT-ASLAN: a specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12(9):879–888, 1986.

[Alf85] M. Alford. SREM at the age of eight; the distributed computing design system. *IEEE Computer*, 1(4), 1985.

[And86] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, 1986.

[AO*86a] G.R. Andrews, R. A. Olsson, et al. An overview of the sr language and implementation. *Journal of Distributed Computing*, 1:133–149, 1986.

[AO86b] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *Distributed Computing*, 1, 1986.

[AO88a] G.R. Andrews and R.A. Olsson. The evolution of the sr language. *ACM Transactions On Programming Languages And Systems*, 10(1):51–86, 1988.

[AO88b] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *ACM Transactions On Programming Languages And Systems*, 10(1):51–86, 1988.

[AOC*88] G. Andrews, R. Olsson, M. Coffin, I Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[AP87] Schlichting R.D. Hayes R. Andrews, G.R. and T.D.M. Purdin. The desing of the Saguaro distributed operating system. *IEEE Transactions on Software Engineering*, 13(1):104–118, 1987.

[Apt86] K.R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions On Programming Languages And Systems*, 8(3):388–405, 1986.

[AS83a] Gasser M. Ames Jr., S.R. and R.R. Schell. Security kernel design and implementation: an introduction. *IEEE Computer*, 16(7):14–22, 1983.

[AS83b] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):UNKNOWN, 1983.

[B*82] H. Berg et al. *Formal Methods of Program Verification and Specification.* Prentice-Hall, Inc., 1982.

[Bac86] M. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

[Bad86] D.Z. Badal. The distributed deadlock detection algorithm. *ACM Transactions On Computer Systems*, 4(4):320–337, 1986.

[Bar85] H. Barringer. *A Survey of Verfication Techniques for Parallel Programs.* Volume 191, Springer-Verlag, 1985.

[BB79] T.A. Berson and G.L Barksdale Jr. KSOS—Development methodology for a secure operating system. In *Proceedings of the AFIPS National Computer Conference*, pages 365–371, 1979.

[BC84] M. Boari and S. Crespi-Reghizzi. Multiple-microprocessor programming techniques: MML, a new set of tools. *IEEE Computer*, 1(1), 1984.

[Bes85] E. Best. Concurrent behavior: sequences, processes and axioms. In *Seminar on Concurrency: lncs, Volume 197*, pages 221–245, Springer-Verlag, 1985.

[BF*87] F. Baiardi, A. Fantechi, et al. Distributed implementation of nested communicating sequential process: communication and termination. *Journal of Parallel and Distributed Computing*, 4, 1987.

[BG*82] D.M. Berry, C. Ghezzi, et al. Language constructs for real-time distributed systems. *Computer Languages*, 7, 1982.

[BG85] T.A. Budd and A.S. Gopal. Program testing by specification mutation. *Computer Languages*, 10, 1985.

[BGHL87] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: a formal specification. *Symposium on Operating System Principles*, 21(5):94–104, 1987.

[BH86] B. Edupuganty Bryant, B.R. and L.S. Hull. Two-level grammar as an implementable metalanguage for axiomatic semantics. *Computer Languages*, 11(3):173–191, 1986.

[BH*87] F. Bastani, W. Hilal, et al. Efficient abstract data type components for distributed and parallel systems. *IEEE Computer*, 1(10), 1987.

[Bir85] A.D Birrell. Secure communication using remote procedure calls. *ACM Transactions On Computer Systems*, 3(1):1–14, 1985.

[BL75] D.E. Bell and L.J. LaPadula. *Secure Computer systems: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, July 1975.

[BL88] B.N. Bershad and H.M. Levy. A remote computation facility for a heterogenous environment. *IEEE Computer*, 1(5), 1988.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[BM81a] R. Boyer and J. Moore. *The Correctness Problem in Computer Science. Intl Lecture Series in Computer Science*, Academic Press, 1981.

[BM81b] R. S. Boyer and J. S. Moore. A verification condition generator for fortran. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, 1981.

[BMR82] D.R. Brownbridge, L.F. Marshal, and B. Randell. The newcastle connection, or unixes of the world unite! *Software — Practice and Experience*, 12:1147–1162, December 1982.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions On Computer Systems*, 2(1):39–59, 1984.

[BO*85] R.G. Babb, K. Orr, et al. Workshop on models and languages for software specification and design. *IEEE Computer*, 1(3), 1985.

[Boe85] H. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions On Programming Languages And Systems*, 7(4):637–655, 1985.

[Boe88] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 1(5), 1988.

[Bok87] S.H. Bokhari. Multiprocessing the Sieve of Eratosthenes. *IEEE Computer*, 1(4), 1987.

[BR83] G.V. Bochmann and M. Raynal. Structured specification of communicating systems. *IEEE Transactions on Computing*, 32(2), 1983.

[Bro85a] S.D. Brookes. In an axiomatic treatment of a prallel programming language. In *Logics of Programs: lncs, Volume 193*, pages 41–60, Springer-Verlag, 1985.

[Bro85b] S.D. Brookes. On the axiomatic treatment of concurrency. In *Seminar on Concurrency: lncs, Volume 197*, pages 1–34, Springer-Verlag, 1985.

[BS86] T.P. Blumer and D. P. Sidhu. Mechanical verification and automatic implementation of communication protocols. *IEEE Transactions on Software Engineering*, 12(8):827–836, 1986.

[BT84] L.A. Bergstra and J.V. Tucker. The axiomatic semantics of programs based on Hoare's logic. *Acta Informatica*, 21, 1984.

[BT85] T. Bensel and D. Tavilla. Trusted software verfication: a case study. *IEEE Conference on Security and Privacy*, 14–24, 1985.

[Bur84] F. W. Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions On Programming Languages And Systems*, 6(2):159–174, 1984.

[BW82] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18, 1982.

[BY87] W. Bevier, W.and Hunt and W. Young. Toward verified execution environments. *IEEE Conference on Security and Privacy*, 106–116, 1987.

[C*85a] J. S. Crow et al. *SRI Verification System Version 2.0 Specification Language Description*. Technical Report, SRI International, November 1985.

[C*85b] J. S. Crow et al. *SRI Verification System Version 2.0 User's Guide*. Technical Report, SRI International, November 1985.

[C*86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[Cam85] M. Campbell. Computer security: a status report. *Hawaii Systems Sciences Conference*, 2:742–755, 1985.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions On Programming Languages And Systems*, 8(2):244–263, 1986.

[CF*86] V. Carchiolo, A. Faro, et al. A LOTUS specification of the PROWAY highway service. *IEEE Transactions on Computing*, 35(11), 1986.

[CGM87] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *HDL Descriptions to Guaranteed Correct Circuit Designs*, Elsevier Scientific Publishers, 1987.

[CH81] Z.C. Chen and C.A.R. Hoare. Partial correctness of communicating sequential processes. *Proc. International Conference on Distributed Computing*, UNKNOWN(UNKNOWN):UNKNOWN, 1981.

[Cha85] E. Chang. Verification of non-terminating concurrent programs. *IEEE Conference on Distributed Computer Systems*, 411–415, 1985.

[Che88] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[CHL*89] G.C. Cohen, M.J. Healy, K. Levitt, P. Windley, S. Kalvala, A. Jasuja, J. Pan, J. Alves-Foss, J. Buffenba, and M. Sievers. Formal verification with hol – an aerospace perspecitive. August 1989. NASA Contract NAS1-18586 Report (First Draft).

[CKM*88] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltink. m-EVES a tool for verifying software. In *10th International Conference on Software Engineering*, IEEE, 1988.

[CL*84] W.W. Chu, M. Lan, et al. Estimation of intermodule communication (IMC) and its applications in distributed processing systems. *IEEE Transactions on Computing*, 33(8), 1984.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions On Computer Systems*, 3(1):63–75, 1985.

[Cla79] E.M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiomatic systems. *Journal of the ACM*, 26(1):129–147, 1979.

[CM81] Gasser M. Huff G.A. Cheheyl, M.H. and J.K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, 1981.

[CM84a] K. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions On Programming Languages And Systems*, 6(4):632–646, 1984.

[CM84b] J. Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions On Computer Systems*, 2(3):251–273, 1984.

[CM87] M.F. Coulas and G.H. MacEwen. Rnet: a hard real-time distributed programming system. *IEEE Transactions on Computing*, 36(8), 1987.

[Coh86] N.H. Cohen. *Ada Axiomatic Semantics: Problems and Solutions*. Technical Report 223, SofTech, 1986.

[Coh88a] A. Cohn. *Correctness Properties of the Viper Block Model: The Second Level*. Technical Report, University of Cambridge Computer Laboratory, 1988.

[Coh88b] A. Cohn. *Correctness Properties of the Viper Block Model: The Second Level*. Technical Report, University of Cambridge Computer Laboratory, 1988.

[Coh88c] A. Cohn. A proof of correctness of the VIPER microprocessor: the first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988.

[Coh88d] A. Cohn. A proof of correctness of the viper microprocessor: the first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988.

[Con85] R. Constable. Constructive mathematics as a programming logic I: some principles of theory. *Annals of Discrete Mathematics*, 24, 1985.

[Coo81] R.P. Cook. Abstractions for distributed programming. *Computer Languages*, 6, 1981.

[Cra84] D. Craigen. Ottawa Euclid and Eves: a status report. *IEEE Conference on Security and Privacy*, 114–124, 1984.

[Cra87] D. Craigen. *m-EVES*. Technical Report CP-87-5402-21, I. P. Sharp Associates Limited, 1987.

[Cra88] D Craigen. An application of the m-EVES verification system. In *Second Workshop on Software Testing, Verification, and Analysis*, ACM/SIGSOFT and IEEE/CS, 1988.

[CV84] M. Clint and C. Vincent. The use of ghost variables and virtual programming in the documentation and verification of programs. *Software — Practice and Experience*, 14(8):711–737, 1984.

[CZ83] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Ninth Annual ACM Symposium on Operating System Principles in Operating Systems Review*, 17(5):129–140, 1983.

[CZ85] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions On Computer Systems*, 3(2):77–107, 1985.

[dBM87] J.W. de Bakker and J.J.C. Meyer. Order and metric in the stream semantics of elemental concurrency. *Acta Informatica*, 24, 1987.

[DE82] R. Dannenberg and G. Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, 8(1), 1982.

[DE*87] S.A. Dart, R.J. Ellison, et al. Software development environments. *IEEE Computer*, 1(11), 1987.

[Den82] D.E. Denning. *Cryptography and Data Security.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DOD83] *Department of Defense Trusted Computer System Evaluation Criteria.* Department Of Defense Computer Security Center, August 1983.

[ECD73] Jr. E.G. Coffman and P.J. Denning. *Operating Systems Theory.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

[ELH83] B. Elspas, K. N. Levitt, and D. Hare. *Verification of Jovial Programs.* Technical Report, SRI International, 1983.

[EP*83] U. Engels, U. Pletat, et al. An operational semantics for specifications of abstract data types with error handling. *Acta Informatica*, 19, 1983.

[Fei80] R. J. Feiertag. *A Technique for Proving Specifications are Multilevel Secure.* Technical Report CSL-109, SRI International, January 1980.

[Fin86] R. Finkel. *An Operating Systems Vade Mecum.* Prentice-Hall, 1986.

[Flo67] R.W. Floyd. Assigning meanings to programs. *American Mathematical Society*, 19–32, 1967.

[FLR77a] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 57–65, ACM, 1977.

[FLR77b] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 57–65, ACM, 1977.

[FOR80] *Provably Secure Operating System (PSOS) Final Report.* Ford Aerospace and Communications Corporation, June 1980.

[Fra83] L.J Fraim. Scomp: a solution to the multilevel security problem. 16(7):26–53, 1983.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[FW86] D.A. Fisher and R.M. Weatherly. Issues in the design of a distributed operating system for Ada. *IEEE Computer*, 1(5), 1986.

[G*86] V. Gligor et al. A new security testing method and its application to the secure Xenix kernel. *IEEE Conference on Security and Privacy*, 40–50, 1986.

[Gan83] H. Ganzinger. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Transactions On Programming Languages And Systems*, 5(3):318–354, 1983.

[GC84] N. Gehani and T. Cargill. Concurrent programming in the Ada language: the polling bias. *Software — Practice and Experience*, 14(5):413–427, 1984.

[GCC*87] V.D. Gligor, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hecht, W. Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. Design and implementation of Secure Xenix. *IEEE Transactions on Software Engineering*, 13(2):208–221, 1987.

[Geh82] N. Gehani. Specifications: formal and informal — a case study. *Software — Practice and Experience*, 12:433–444, 1982.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions On Programming Languages And Systems*, 7(1):80–112, 1985.

[GH78] J. Guttag and J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHM78] J. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.

[GKM87] J. Goguen, C. Kirchner, and J. Meseguer. *Concurrent Term Rewriting as a Model of Computation.* Technical Report SRI-CSL-87-2, SRI International, May 1987.

[GM82] J.A. Goguen and J. Meseguer. Security policies and security models. *IEEE Conference on Security and Privacy*, 7(5):11–20, December 1982.

[GM84] J.A. Goguen and J. Meseguer. Unwinding and inference control. *IEEE Conference on Security and Privacy*, SE-10(5):75–86, 1984.

[GM86a] N. Gehani and A.D. McGettrick. *Software Specification Techniques. International Computer Science Series*, Addision-Wesley Publishing Company, 1986.

[GM86b] J. Goguen and J. ¿Meseguer. Extensions and foundations of object-oriented programming. *ACM SIGPLAN*, 21(10):153–162, 1986.

[GM87a] J.I. Glascow and G.H. MacEwen. The development and proof of a formal specification for a multilevel secure system. *ACM Transactions On Computer Systems*, 5(2):151–184, 1987.

[GM87b] J.I. Glasgow and G.H. MacEwen. The development and proof of a formal specification for a multilevel secure system. *ACM Transactions on Programming Languages and Systems*, 5(2):151–184, 1987.

[GM87c] J. Goguen and M. Moriconi. Formalization in programming environments. *IEEE Computer*, 1(11), 1987.

[GMT*80] S. Gerhart, D. Musser, D. Thompson, D. Baker, R. Bates, R. Erickson, R. London, D. Taylor, and D. White. An overview of affirm: a specification and verification system. In S. Lavington, editor, *Information Processing 80*, North-Holland, 1980.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. In *lncs, Volume 78*, chapter 2, spver, 1979.

[GNS88] D.K. Gifford, R.M. Needham, and M.D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.

[Gog80] J. Goguen. How to prove inductive hypotheses without induction. In *Lecture Notes in Computer Science, Volume 87*, pages 356–373, Springer-Verlag, 1980.

[Gog86] J.A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 1(2), 1986.

[Gog88] J. Goguen. *OBJ3 as a Theorem Prover with Applications to Hardware Verification*. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming*. Volume 130, Springer-Verlag, 1982.

[Gor83] R. L. Gordon. Experience with a distributed system testbed. *Hawaii Systems Sciences Conference*, 1:356–366, 1983.

[Gor87] M. Gordon. *A Proof Generating System for Higher-Order Logic*. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

[Gor88] M. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 73–128, Kluwer Academic Press, 1988.

[GS86a] K. Geihs and M. Seifert. Automated validation of co-operation protocol for distributed systems. *IEEE Conference on Distributed Computer Systems*, 436–443, 1986.

[GS86b] S. Graf and J. Sifakis. A logic for the specification and proof of regular controllable processes of ccs. *Acta Informatica*, 23, 1986.

[GT86a] J. Goguen and J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, Addison-Wesley, 1986.

[GT86b] J. Goguen and J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, Addison-Wesley, 1986.

[Gut75] J. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Department of Computer Science, October 1975.

[Gut87] J. Guttman. Information flow and invariance. *IEEE Conference on Security and Privacy*, 67–77, 1987.

[GW88a] J. Goguen and T. Winkler. *Introducing OBJ3*. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[GW88b] J. Goguen and T. Winkler. *Introducing OBJ3*. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[H*86] J. Halpern et al. Muse - a computer assisted verification system. *IEEE Conference on Security and Privacy*, 25–35, 1986.

[Hal84] J.Y. Halpern. A good Hoare axiom system for an Algol-like language. *Eleventh Annual ACM Symposium on Principles of Programming Languages*, 262–271, 1984.

[Ham88] D. Hammerslag. *Treemacs Manual.* Technical Report UIUCDCS-R-88-1427, University of Illinois at Urbana-Champaign Department of Computer Science, May 1988.

[Hay87] I. Hayes. *Specification Case Studies.* Prentice-Hall, Inc., 1987.

[HD86] M.E. Hull and G. Donnan. Contextually communicating sequential processes — a software engineering environment. *Software — Practice and Experience*, 16(9):845–864, 1986.

[Her87] M. Herlihy. Concurrency versus availability: atomicity mechanisms for replicated data. *ACM Transactions On Computer Systems*, 5(3):249–274, 1987.

[HG85] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transactions on Computing*, 34(10), 1985.

[HI85] T. Hikita and K. Ishihata. A method of program transformation between variable sharing and message passing. *Software — Practice and Experience*, 15(7):677–692, 1985.

[HM83] C. Heitmeyer and J. McLean. Abstract requirements specification: a new aprpoach and its application. *IEEE Transactions on Software Engineering*, 9(5), 1983.

[HM*84a] T. Higashino, M. Mori, et al. An algebraic specification of HDLC procedures and its verification. *IEEE Transactions on Software Engineering*, 10(6):825–847, 1984.

[HM84b] M.E. Hull and R. McKeag. Communicating sequential processes for centralized and distributed operating system design. *ACM Transactions On Programming Languages And Systems*, 6(2):175–191, 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa75] C.A.R. Hoare. Parallel programming: an axiomatic approach. *Computer Languages*, 1, 1975.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa81] C.A.R. Hoare. A calculus of total correctness for communicating processes. *Science of Computer Programming*, 1(1):49–72, 1981.

[Hoa87] C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, 1(9), 1987.

[Hof85] D. Hoffman. The trace specification of communications protocols. *IEEE Transactions on Computing*, 34(12), 1985.

[HR*86] S. Hariri, C.S. Raghavendra, et al. Reliability analysis in distributed systems. *IEEE Conference on Distributed Computer Systems*, 564–571, 1986.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[INT86] *80386 System Software Writer's Guide*. 1986.

[Jal87] P. Jaloate. Synthesizing implementations of abstract data types from axiomatic specifications. *Software — Practice and Experience*, 17(11):847–858, 1987.

[Jan87] R. Janicki. A formal semantics for concurrent systems with a priority relation. *Acta Informatica*, 24, 1987.

[JK86] T.A. Joseph and P.B. Kenneth. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions On Computer Systems*, 4(1):54–70, 1986.

[JL*87] J. Joyce, G. Lomow, et al. Monitoring distributed systems. *ACM Transactions On Computer Systems*, 5(2):121–150, 1987.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions On Programming Languages And Systems*, 5(4):596–619, 1983.

[Jon86] C. Jones. *Systematic Software Develpment using VDM. International Computer Science Series*, Prentice-Hall, Inc., 1986.

[Joy88a] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988.

[Joy88b] J. Joyce. Using higher-order logic to specify computer hardware and architecture. In D. Edwards, editor, *Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture*, North-Holland, 1988.

[JT88] D.M. Johnson and F.J. Thayer. Stating security requirements with tolerable sets. *ACM Transactions On Computer Systems*, 6(3):284–295, 1988.

[Kar84] R.A. Karp. Proving failure-free properties of concurrent systems using temporal logic. *ACM Transactions On Programming Languages And Systems*, 6(2):239–253, 1984.

[KB70] D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297, Pergamon Press, New York, 1970.

[Kem79] R.A. Kemmerer. *Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs*. PhD thesis, University of California, Los Angeles, 1979.

[Kem85] R.A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–42, 1985.

[Kem86] R.A. Kemmerer. *Verification Assessment Study Final Report, Volume III, The Affirm System*. Technical Report, Department of Computer Science, University of California, March 1986.

[KJA85] S.N. Kamin, S. Jefferson, and M. Archer. *The FASE System of Executable Specifications of Data Types*. Technical Report, University of Illinois, October 1985.

[Kle85] L. Kleinroch. Distributed systems. *IEEE Computer*, (1), 1985.

[Klu83] W.E. Kluge. Cooperating reduction machines. *IEEE Transactions on Computing*, 32(11), 1983.

[KP86] S. Kaplan and A. Pnueli. *Specification and Implementation of Concurrently Accessed Data Structures: An Abstract Data Type Approach*. Technical Report CS86-23, Weismann Institute of Science, 1986.

[Kra87] B. Kramer. SEGRAS - a formal and semigraphical language combining Petri nets and abstract data types for the specificatin of distributed systems. *International Conference on Software Engineering*, 116–126, 1987.

[Kro87] F. Kroger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[KS86] J. Kerridge and D. Simpson. Communicating parallel processes. *Software — Practice and Experience*, 16(1):63–86, 1986.

[Kut84] S. Kutti. Why a distribute kernel? *Operating Systems Review*, 18(4):5–11, 1984.

[L*83] K. N. Levitt et al. *Investigation, Development and Evaluation of Performance Proving for Fault-Tolerant Computers*. Technical Report, SRI International, 1983.

[L*85] Clarke L. et al. A comparison of data flow path selection criteria. *International Conference on Software Engineering*, 244–254, 1985.

[Lad87] P. Ladkin. Specification of time dependencies and synthesis of concurrent processes. *International Conference on Software Engineering*, 106–116, 1987.

[Lam82] L. Lamport. *What Good is Temporal Logic*. Technical Report, Digital Research Systems Center, November 1982.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions On Programming Languages And Systems*, 5(2):190–222, 1983.

[Lam86a] L. Lamport. *Control Predicates Are Better Than Dummy Variables for Reasoning About Program Control*. Technical Report, Technical Report, Digital Research System Center, May 1986.

[Lam86b] L. Lamport. *Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control.* Technical Report, Digital Research System Center, May 1986.

[Lam86c] L. Lamport. On interprocess communication (Part I: basic formalism). *Distributed Computing*, 1, 1986.

[Lam86d] L. Lamport. On interprocess communication (Part II: algorithms). *Distributed Computing*, 1, 1986.

[Lan81] C.E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, 1981.

[Lan83] C.E. Landwehr. The best available technologies for computer security. *IEEE Computer*, 16(7):86–100, 1983.

[LBF85] T.J. Le Blanc and S.A. Friedberg. HPC: a model of structure and change in distributed systems. *IEEE Transactions on Computing*, 34(12), 1985.

[Lel88] W. Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[Len88] P.M. Lenders. A generalized message-passing mechanism for communicating sequential processes. *IEEE Transactions on Computing*, 37(6), 1988.

[Lev80] K. N. Levitt. *A Secure Operating System.* Technical Report, SRI International, 1980.

[LG86] B. Liskov and J. Guttag. *Abtraction and Specification in Program Development.* MIT Press, 1986.

[LHM84] C.E. Landwehr, C.L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Transactions On Computer Systems*, 2(3):198–222, 1984.

[LM87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computing*, 36(4), 1987.

[Loe85] K. Loepere. Resolving covert channels within a B2 class secure system. *Operating Systems Review*, 19(3):9–28, 1985.

[LRS79] K. N. Levitt, L. Robinson, and B. Silverberg. *The HDM Handbook, Volumes I, II, III.* Technical Report, SRI International, 1979.

[LS84] L. Lamport and F. Schneider. The Hoare logic of CSP, and all that. *ACM Transactions On Programming Languages And Systems*, 6(2):281–296, 1984.

[LS86] W. Lu and M. Sundareshan. A model for multilevel security in computer networks. In *Proceedings 7th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1095–1104, March 1986.

[LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification.* B. G. Teubner, 1987.

[Lub84] B.D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 21, 1984.

[Lv85] D.C. Luckham and W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, 1985.

[LW86] B. Liskov and W. Weihl. Specifications of distributed programs. *Distributed Computing*, 1, 1986.

[Mas88] I.A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10(2):177–210, 1988.

[MB83] P. Merlin and G. Bochmann. On the construction of submodule specifications and communication protocols. *ACM Transactions On Programming Languages And Systems*, 5(1):1–25, 1983.

[MB*84] Schroeder. M.D., A.D. Birrell, et al. Experience with Grapevine: the growth of a distributed system. *ACM Transactions On Computer Systems*, 2(1):3–23, 1984.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4), 1981.

[McC87] D. McCullough. Specifications for multi-level security and a hook-up property. *IEEE Conference on Security and Privacy*, 161–166, 1987.

[McH85] J. McHugh. An information flow tool for Gypsy. *IEEE Conference on Security and Privacy*, 46–56, 1985.

[McL87] J. McLean. Reasoning about security models. *IEEE Conference on Security and Privacy*, 123–133, 1987.

[MD79] E.J. McCauley and P.J. Drongowski. KSOS—The design of a secure operating system. In *Proceedings of the AFIPS National Computer Conference*, pages 345–353, 1979.

[Mel88] T. Melham. *Automating Recursive Type Definitions in Higher-Order Logic*. Technical Report 146, University of Cambridge Computer Laboratory, September 1988.

[Mey85] B. Meyer. On formalism in specification. *IEEE Software*, 2(1):6–27, 1985.

[MF86] D.A. Mundie and D.A. Fisher. Parallel processing in Ada. *IEEE Computer*, 1(8), 1986.

[MG83] P. McMullin and J. Gannon. Combining testing with formal specifications a case study. *IEEE Transactions on Software Engineering*, 9(3), 1983.

[Mil76] J.K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, 1976.

[Mil83] A. Mili. Proving programs by stuctural induction: a new perspective on the sometime method and the subgoal induction method. *Hawaii Systems Sciences Conference*, 1:112–117, 1983.

[Mil87] J.K. Millen. Covert channel capacity. *IEEE Conference on Security and Privacy*, 60–66, 1987.

[MJ83] P. Mateti and J. Jaffar. A correctness proof of an indenting program. *Software — Practice and Experience*, 13:199–226, 1983.

[MLF84] W.N. McKusick, M.K.and Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions On Computer Systems*, 2(3):181–197, 1984.

[MM83] A.R. Meyer and J.C. Mitchell. Termination assertions for recursive programs: completeness and axiomatic definability. *Information and Control*, 56(1):112–138, 1983.

[MM85] Girgis M. and Woodward M. An integrated system for program testing using weak mutation and data flow analysis. *International Conference on Software Engineering*, 313–323, 1985.

[MM86] J. McHugh and A. Moore. A security policy and formal top level specification for a multi-level secure local area network. *IEEE Conference on Security and Privacy*, 34–44, 1986.

[Mur86] T.P. Murtagh. Eliminating proofs of non-interference from Levin-Gries CSP program proofs. *IEEE Conference on Distributed Computer Systems*, 494–499, 1986.

[Mus80] D. Musser. Abstract data type specification in the affirm system. *IEEE Transactions on Software Engineering*, 1:24–32, January 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions On Programming Languages And Systems*, 6(1):68–93, 1984.

[NBF*80] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. *A Provably Secure Operating System: The System, its Applications, and Proof.* Technical Report CSL-116, SRI, May 1980.

[ND*86] V. Nguyen, A. Demers, et al. A model and temporal proof system for networks of processes. *Distributed Computing*, 1, 1986.

[Nem85] R.M. Nemes. Modular verification of distributed systems. *IEEE Conference on Distributed Computer Systems*, 396–410, 1985.

[Nes87] D.M. Nessett. Factors afecting distributed system security. *IEEE Transactions on Software Engineering*, 13(2):233–248, 1987.

[Nic85] R. De Nicola. Two complete axiom systems for a theory of communicating sequential processes. *Information and Control*, 64(1):136–172, 1985.

[Nor87] J.D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems The Alpha Kernel.* Academic Press, Inc., Orlando, Florida, 1987.

[NS78] R.M. Needham and M.D. Schroder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[NZ88] Black A.P. Lazowska E.D. Levy H.M. Sanislo J. Notkin, D. and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, 1988.

[OC*88] J.K. Ousterhout, A.R. Cherenson, et al. The sprite network operating system. *IEEE Computer*, 1(2), 1988.

[OH86] B.R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23, 1986.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions On Programming Languages And Systems*, 4(3):455–495, 1982.

[Oss83] M. Ossefort. Correctness proofs of communicating processes: three illustrative examples from the literature. *ACM Transactions On Programming Languages And Systems*, 5(4):620–640, 1983.

[Owi80a] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. Garland Publishing, Inc., 1980.

[Owi80b] S. S. Owicki. Specification and verification of a network mail system. In *Seminar on Concurrency: lncs, Volume 197*, pages 198–234, Springer-Verlag, 1980.

[Pag79] F.G. Pagan. Semantic specification using two-level grammars: blocks, procedures and parameters. *Computer Languages*, 4, 1979.

[Pau83] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pau87] L. Paulson. Logic and computation: interactive proof with cambridge lcf. In *Cambridge Tracts in Theoretical Computer Science 2*, Cambridge University Press, 1987.

[Pea82] M Pease. The Byzantine Generals problem. *ACM Transactions On Programming Languages And Systems*, 4(3):382–401, 1982.

[Pet85] N Petschenik. Practical priorities in system testing. *IEEE Software*, 2(5):18–23, 1985.

[PKK*79] G.J. Popek, M. Kampe, C.S. Kline, A. Stoughton, M. Urban, and E.J. Walton. UCLA Secure Unix. In *Proceedings of the AFIPS National Computer Conference*, pages 355–364, 1979.

[Pra84] V.R. Prasad. Interference-freedom in proofs of CSP programs. *IEEE Conference on Distributed Computer Systems*, 79–86, 1984.

[Pro85] N. Proctor. The restricted access proessor, an example of formal verfication. *IEEE Conference on Security and Privacy*, 49–59, 1985.

[PS*86] N. Prywes, Y. Shi, et al. Supersystem programming with Model. *IEEE Computer*, 1(2), 1986.

[QK*85] D. Quammen, J.P. Kearns, et al. Efficient storage management for temporary values in concurrent programming languages. *IEEE Transactions on Computing*, 34(9), 1985.

[Ram83] A. Ramsay. A distributed programming assistant. *Software — Practice and Experience*, 13:983–992, 1983.

[Rei82] W. Reisig. Deterministic buffer synchronization of sequential processes. *Acta Informatica*, 18, 1982.

[Rem84] J.H. Remmers. A technique for developing loop invariants. *Information Processing Letters*, 18(3):137–139, 1984.

[Ric80] R.P. Rich. Mechanical proof testing. *Computer Languages*, 5, 1980.

[RK83] K. Ramamritham and R. Keller. Specification of synchronizing processes. *IEEE Transactions on Software Engineering*, 9(6), 1983.

[RM84] G. Roman and Day M. Multifaceted distributed systems specification using processes and event symchronization. *International Conference on Software Engineering*, 44–54, 1984.

[Rom87] G. Roman. Specifying software/hardware iteractions in distributed systems. *International Conference on Software Engineering*, 126–136, 1987.

[RR83] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, 1983.

[RR86] R.R. Razouk and M.T. Rose. Verifying partial correctness of concurrent software using contour/transition-nets. *Hawaii Systems Sciences Conference*, 2A:734–743, 1986.

[RS*85] D. Rotem, N. Santoro, et al. Distributed sorting. *IEEE Transactions on Computing*, 34(4), 1985.

[Rus81] J.M Rushby. Design and verification of secure systems. *Eighth Annual ACM Symposium on Operating System Principles in Operating Systems Review*, 15(5):12–21, 1981.

[Rus82] J.M. Rushby. Proof of separability. a verification technique for a class of security kernels. *International Symposium on Programming, Lecture Notes in Computer Science*, (137):352–367, 1982.

[Rus84] J. Rushby. The security model of enhanced hdm. In *Proceedings of Seventh DoD/NBS Computer Security Conference*, pages 120–136, DoD Computer Security Center, 1984.

[S84a] Ntafos S. An evaluatino of required element testing strategies. *International Conference on Software Engineering*, 250–260, 1984.

[S*84b] E.M. Clarke Sistla, A.P. et al. Can message buffers be axiomatized in linear temporal logic? *Information and Control*, 63(1):88–112, 1984.

[San84] D.T. Sannella. A set-theoretic semantics for Clear. *Acta Informatica*, 21, 1984.

[Sch82a] F.B. Schneider. Synchronization in distributed programs. *ACM Transactions On Programming Languages And Systems*, 4(2):179–195, 1982.

[Sch82b] J. Schwarz. Using annotations to make recursion equations behave. *IEEE Transactions on Software Engineering*, 8(1):21–35, 1982.

[SD*82] C.A. Sunshine, D.H.Thompson, et al. Specification and verification of communication protocols in AFFIRM using state transition methods. *IEEE Transactions on Software Engineering*, 8(5), 1982.

[SdR87] F.A. Stomp and W.P. de Roever. A correctness prrof of a distributed minimum-weight spanning tree algorithm. *IEEE Conference on Distributed Computer Systems*, 440–447, 1987.

[Sha84] S.M. Shatz. Communication mechanisms for programming distributed systems. *IEEE Computer*, 1(6), 1984

[Sil83] J.M. Silverman. Reflections on the verification of the security of an operating system kernel. *Ninth Annual ACM Symposium on Operating System Principles in Operating Systems Review*, 17(5):143–154, 1983.

[SK85] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions On Computer Systems*, 3(4):344–349, 1985.

[SK86] O.S. Saydjari and T.W. Kremann. A standard notation in computer security models. *Proceedings kof the 9th national Computer Security Conference*, 1986.

[SL83] A.U. Shankar and S.S. Lam. An HDLC protocol specification and its verification using image protocols. *ACM Transactions On Computer Systems*, 1(4):331–368, 1983.

[SL87] S. Sluizer and S. Lee. Using executable specifications to model user requirements. *Hawaii Systems Sciences Conference*, 2:41–49, 1987.

[SM85] K.H. Sears and A. E. Middleditch. Software concurrency in real time control systems: a software nucleus. *Software — Practice and Experience*, 15(8):739–759, 1985.

[Sou84] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions On Programming Languages And Systems*, 6(4):647–662, 1984.

[Sou86] N. Soundararajan. Total correctness of CSP programs. *Acta Informatica*, 23, 1986.

[SS84] R. Schlichting and F. Schneider. Using message passing for distributed programming: proof rules and disciplines. *ACM Transactions On Programming Languages And Systems*, 6(3):402–431, 1984.

[SS88] A.E.K. Sobel and N. Soundararajan. A proof system for distributed processes. *Acta Informatica*, 25, 1988.

[SSM85] R. E. Shostak, R. L. Schwartz, and P. M. Melliar-Smith. RSTP: a mechanical logic for specification and verification. In , editor, *Proceedings Sixth Conference on Automated Deduction*, June 1985.

[ST88] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25, 1988.

[Sta84] J.A. Stankovic. A perspective on distributed computer systems. *IEEE Transactions on Computing*, 33(12), 1984.

[SW87] S.M. Shatz and J. Wang. Introduction to distributed-software engineering. *IEEE Computer*, 1(10), 1987.

[Sza87a] A. Szalas. Arithmetical axiomatization of first-order temporal logic. *Information Processing Letters*, 26(3):111–116, 1987.

[Sza87b] A. Szalas. A complete axiomatic characterization of first-order temporal logic of linear time. *Theoretical Computer Science*, 54(3):199–214, 1987.

[TA87] K. Tai and S. Ahujs. Reproducible testing of communication software. *International Computer Software and Applications Conference*, 331–341, 1987.

[Tak87] T. Takaoka. A decomposition rule for the hoare logic. *Information Processing Letters*, 26(4):250–208, 1987.

[Tay83] R.N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19, 1983.

[TCVW*88] Jr. T.A. Casey, S.T. Vinter, D.G. Weber, R. Varadarajan, and D. Rosenthal. A secure distributed operating system. *IEEE Conference on Security and Privacy*, 27–38, 1988.

[TE81] D. Thompson and R. Erickson. *AFFIRM Reference Manual.* Technical Report, USC Information Sciences Institute, February 1981.

[TWW82] J. Thatcher, E. Wagner, and J. Wright. Data type specification: parameterization and the power of specification techniques. *ACM Transactions On Programming Languages And Systems*, 4(4):711–732, 1982.

[UNK85] UNKNOWN. *Ada for Specification: possibilities and limitations.* Cambridge University Press, 1985.

[UNK86] UNKNOWN. *Program Specification and Transformation.* North Highland, 1986.

[UUD85] S. Urban, J. Urban, and W. Dominick. Utilizing an executable specification language for an information system. *IEEE Transactions on Software Engineering*, 11(7):598–608, 1985.

[VB85] T. Vickers Benzel. Verification technology and the a1 criteria. In *Proceedings of VERkshop III, in ACM Software Engineering Notes, 20(8)*, pages 108–109, 1985.

[VCH88] S.T. Vinter, T.A. Casey, and K.A. Huber. *The Secure Distributed Operating System Design Project.* Technical Report, BBN Laboratories Inc., 1988.

[vdm87] vdm. *VDM '87 VDM - A Formal Method at Work. Proceedings.* Volume 252, Springer-Verlag, 1987.

[Vel84] P.A.S. Veloso. A sound and complete methodology for abstract data type specification: correctness proof and example. *Hawaii Systems Sciences Conference*, 1:399–405, 1984.

[VM87] R.A. Volz and T.N. Mudge. Timing issues in the distributed execution of Ada programs. *IEEE Transactions on Computing*, 36(4), 1987.

[WE86] H. Weber and H. Ehrig. Specification of modular systems. *IEEE Transactions on Software Engineering*, 12(7), 1986.

[Wed85] H.F. Wedde. A formal basis for correct implementations of distributed programming languages. *IEEE Conference on Distributed Computer Systems*, 476–485, 1985.

[Wey84] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12), 1984.

[WH84] R. Williamson and E. Horowitz. Concurrent communication and synchronization methods. *Software — Practice and Experience*, 14(2), 1984.

[Win87] J. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, 1987.

[Wit85a] B.I. Witt. Communicating modules: a software design model for concurrent distributed systems. *IEEE Computer*, 1(1), 1985.

[Wit85b] B.I. Witt. Parallelism, pipelines, and partitions: variations on communicating modules. *IEEE Computer*, 1(2), 1985.

[WLds] R. J. Waldinger and K. N. Levitt. Reasoning about programs. *Artificial Intelligence*, 5(3):235–316, 1974 (also appeared in Studies in Automatic Programming Logic, American Elsevier, 1976, Manna and Waldinger (eds.)).

[WN86] J. Wing and M. Nixon. Extending Ina Jo with temporal logic. *IEEE Conference on Security and Privacy*, 2–12, 1986.

[WP*83] M. Wirsing, P. Pepper, et al. On hierarchies of abstract data types. *Acta Informatica*, 20, 1983.

[WS83] P. Wegner and S. Smolka. Processes, tasks, and monitors: a comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering*, 9(4), 1983.

[YC83] S. Yau and M. Caglayan. Distributed software system design representation using modified Petri nets. *IEEE Transactions on Software Engineering*, 9(6):733–740, 1983.

[YM87] W. Young and J. Mchugh. Coding for a believable specification to implementation mapping. *IEEE Conference on Security and Privacy*, 140–150, 1987.

[Zav86] P. Zave. Case study: the PAISLey approach applied to its own software tools. *Computer Languages*, 11, 1986.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.