

DTIC FILE COPY

4

RADC-TR-88-127
Final Technical Report
June 1988



AD-A204 402

THE SECURE DISTRIBUTED OPERATING SYSTEM DESIGN PROJECT

BBN Laboratories Inc.



Stephen T. Vinter, Thomas A. Casey, Kathleen A. Huber

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

89 2 6 004

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

RADC-TR-88-127 has been reviewed and is approved for publication.

APPROVED: *Emilie J. Siarkiewicz*
EMILIE J. SIARKIEWICZ
Project Engineer

APPROVED: *Raymond P. Urtz, Jr.*
RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:

John A. Ritz
JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) 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.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) BBN Report No. 6144			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-127		
6a. NAME OF PERFORMING ORGANIZATION BBN Laboratories, Inc.		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)		
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238-0001			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0056		
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 35167G	PROJECT NO. 1069	TASK NO. 01
11. TITLE (Include Security Classification) THE SECURE DISTRIBUTED OPERATING SYSTEM DESIGN PROJECT					
12. PERSONAL AUTHOR(S) Stephen T. Vinter, Thomas A. Casey, Kathleen A. Huber *					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM May 85 TO Sep 87		14. DATE OF REPORT (Year, Month, Day) June 1988	
15. PAGE COUNT 402					
16. SUPPLEMENTARY NOTATION *Subcontractors: Odyssey Research Associates - Authors are D.G. Weber, Rammohan Varadarajan, David Rosenthal, Bob Lubarsky, Stanley Perlo, Daryl McCullough (over)					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Multilevel Secure Systems , Trusted Operating Systems Distributed Operating Systems . JES		
12	05				
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This report discusses some issues in distributed system security in the context of the design of a secure distributed operating system (SDOS). The design is targeted for an A1 rating, as per DoD 5200.28-STD. Some new developments in formal verification methods are reported. Distributed system security is contrasted with single-host and network security, and described in the context of the "Trusted Network Interpretation" (NCSC-TG-005). Problems unique to distributed system security are discussed. An argument is made for implementing security features in higher protocol layers, corresponding roughly to the Session through Application layers of the OSI model. A new security policy, based on message-passing rather than reads and writes, is presented. The formal model, functional description, and a formal top level specification are also presented.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Emilie J. Siarkiewicz			22b. TELEPHONE (Include Area Code) (315) 330-2158		22c. OFFICE SYMBOL RADC/COTD

UNCLASSIFIED

Block 16 (Continued)

BBN Communications Corporation - Author is John Linn

1. NAME	J
2. ADDRESS	
3. CITY	
4. STATE	
5. ZIP	
6. PHONE	
7. FAX	
8. E-MAIL	
9. OTHER	
10. COMMENTS	
11. A-1	

UNCLASSIFIED

Contents

Executive Summary	xi
Revision History	xiii
1 Introduction	1
1.1 Overview	1
1.2 Security	2
1.3 Project Direction	3
1.3.1 Phase I: Narrowing the Orientation of the Project	4
1.3.2 Phase II: Decomposing the Problem	4
1.3.2.1 Security Requirements	5
1.3.2.2 Security and Distributed System Functionality	6
1.3.2.3 Choice of Mechanisms	6
1.3.3 Phases III and IV: Policy and Design	7
1.3.3.1 The Policy	7
1.3.3.2 The Design	9
1.3.4 Phase V: Formalization	10
1.3.4.1 Formal Model	11
1.3.4.2 Formal Design Specification	11
1.4 Distributed Operating Systems	12
1.5 Distributed System Security Problems	14
1.5.1 Objects and Managers	14
1.5.2 Communication and Identification Integrity	14

1.5.3	System Environment	15
1.5.4	Configuration and Security Management	15
1.5.5	Feedback Applications	15
2	Policy	17
2.1	The Security Policy	17
2.1.1	Motivation	19
2.1.1.1	The C2 Internet Experiment	20
2.1.1.2	Threats and Policy Requirements	21
2.1.1.3	Assurance	23
2.1.2	A Generic Policy for Multi-Level Security	24
2.1.2.1	Bell-LaPadula and Message-Passing	24
2.1.2.2	A Policy for Message-Passing Operations	26
2.1.2.3	A Policy for MLS Entities	27
2.1.2.4	Additional Rules and Comments for Mandatory Policy	31
2.1.3	Instantiated Policy for SDOS	33
2.1.3.1	SDOS Entities	34
2.1.3.2	Mandatory Policy	36
2.1.3.3	Discretionary Policy	38
2.1.3.4	Configuration Policy	40
2.2	Assigning Values to the Security Policy Parameters	48
2.2.1	Life-Cycle Phases	48
2.2.1.1	Development	48
2.2.1.2	Installation	49
2.2.1.3	Normal Operation	50
2.2.1.4	Modification	50
2.2.1.5	De-installation	50
2.2.2	Assigning Security Labels	51
2.2.2.1	Security Levels	51
2.2.2.2	Integrity Levels	52

2.2.2.3	Level Sets for MLS Entities	55
2.2.3	Strict vs. Flexible Assignments	56
2.3	The Formal Model	57
2.3.1	Approach	57
2.3.2	The Language of the Formal Model	58
2.3.2.1	Types	58
2.3.2.2	Functions	59
2.3.2.3	Policy	60
2.3.3	Implementation and Verification	61
2.3.4	Notation	62
2.3.5	Model	64
2.3.5.1	Types and Restrictions on Types	64
2.3.5.2	Functions and Restrictions on Functions	68
2.3.5.3	Policy	77
3	Design	83
3.1	The Functional Description	83
3.1.1	Introduction	83
3.1.1.1	SDOS Implementation Strategy	83
3.1.1.2	System Environment	87
3.1.1.3	The Approach	89
3.1.2	Principles Underlying SDOS Functions	91
3.1.2.1	Introduction	91
3.1.2.2	Object Management	93
3.1.2.3	Process Management	94
3.1.2.4	Interprocess Communication (IPC)	95
3.1.2.5	Secure Information Transfer	96
3.1.2.6	Discretionary Access Control	97
3.1.2.7	Symbolic Naming	100
3.1.2.8	User Interfaces	101

3.1.2.9	Controlled Software Development	103
3.1.2.10	Configuration, Monitoring and Control (CMC)	104
3.2	Assured COS Support	105
3.2.1	Alternative Approaches to Integration	106
3.2.1.1	Approach I	106
3.2.1.2	Approach II	107
3.2.2	Examples of SDOS Integration with an MLS COS	108
3.2.2.1	Hosting SDOS on GEMSOS	109
3.2.2.2	Hosting SDOS on KeyKOS	109
3.3	Communications Layering	110
3.3.1	OSI Reference Model	110
3.3.2	SDOS Protocol Hierarchy	114
3.4	Network Security	114
3.4.1	Security Services and Mechanisms	116
3.4.1.1	Security Services	117
3.4.1.2	Security Mechanisms	120
3.4.2	SDOS Network Security Approach	123
3.4.2.1	SDOS Network Security	123
3.4.2.2	Separate Versus Embedded Security Module	123
3.4.2.3	Traffic Flow Confidentiality	124
3.4.2.4	TCB Components	124
3.4.2.5	Relationship With The Trusted Network Interpretation Document	124
3.5	Descriptive Top-Level Specification	125
3.5.1	Message Switch System Calls	125
3.5.2	Type Host	127
3.5.2.1	Security Database	127
3.5.2.2	Object Database	128
3.5.2.3	Host Monitoring and Control	130
3.5.3	Type Process	131

3.5.4	Type Principal	133
3.5.5	Type Project	134
3.5.6	Type Object	135
3.6	Design Specification	136
3.6.1	Reasons for Design Decisions	136
3.6.1.1	DAC in Managers	136
3.6.1.2	Mandatory Security	137
3.6.2	The SDOS Trusted Computing Base	139
3.6.3	Detailed Description of the Major TCB Components	143
3.6.3.1	The Message Switch	143
3.6.3.2	The Process Table	144
3.6.3.3	The Locator	144
3.6.3.4	The Host Type	145
3.6.4	Discretionary Access Control	147
3.6.4.1	Client Identities	148
3.6.4.2	Identification	152
3.6.4.3	Access Authorization	153
3.6.4.4	Initial Access Control Lists	155
3.6.4.5	Discretionary Control Assurance	156
4	Formal Methods	159
4.1	The Formal Top-Level Specification	159
4.1.1	Overview	159
4.1.2	The Kernel	160
4.1.2.1	Introduction	160
4.1.2.2	The Security Database	161
4.1.2.3	The Object Database	168
4.1.2.4	Message Switch	170
4.1.2.5	Locator	173
4.1.2.6	Process Manager	175

4.1.2.7	Process Table	177
4.1.3	The File Manager	178
4.1.3.1	Introduction	178
4.1.3.2	Operations	179
4.1.4	The Catalog Manager	182
4.1.4.1	Operations	184
4.1.4.2	Other Remarks	189
4.1.5	Authentication	189
4.1.5.1	The TIP and Authentication Manager	189
4.1.5.2	Achieving Security	191
4.1.5.3	Authentication and System Security	192
4.2	Verifying MLS Properties	192
4.2.1	A New Security Methodology	193
4.2.2	Basic Theory	194
4.2.2.1	Notation	194
4.2.2.2	Processes	195
4.2.2.3	Security Properties	196
4.2.2.4	Buffers	198
4.2.2.5	WNI and Determinism	199
4.2.2.6	Strengthening WNI	203
4.2.3	Applying the Theory to Gypsy	206
4.2.3.1	Verifying WNI using Gypsy	208
4.2.3.2	Gypsy Subprocedures	215
4.2.4	Extensions to the Theory	220
4.2.4.1	"True" Levels for messages	220
4.2.4.2	Input-Limited Restrictive Hookup Theorem	221
4.2.4.3	Limited Insecurity	224
4.3	Verification of the FTLS	231
4.3.1	Overview	231
4.3.2	Verifying the File Manager design	232

4.3.2.1	A Brief Recap of the design	232
4.3.3	Verifying the Catalog Manager Specification	234
4.3.4	Verifying the Kernel specification	236
4.3.4.1	Intuitive understanding of kernel security	236
4.3.4.2	Reasons for not conducting formal verification	237
4.3.5	Proof of processes involved in Authentication	238
4.3.5.1	TIP specifications	238
4.3.5.2	Authentication Specifications	241
4.3.5.3	Message Switch Assumptions	242
4.3.5.4	External Assumptions	242
4.3.5.5	Authentication Manager Proof	243
4.3.5.6	Proof of restriction for the TIP (including the Filter). . .	244
4.3.5.7	Implementation Considerations	246
4.3.5.8	Composability of the User and the TIP	246
4.3.6	Concluding Remarks	247
4.3.6.1	Limitations on users	247
4.3.6.2	Conclusion	248
5	Final Report	249
5.1	Project Goals and Accomplishments	249
5.1.1	Distinction Between Network and DOS Security	251
5.1.2	Contrasts Between Single-host and DOS Security	252
5.1.2.1	TCB Boundaries	252
5.1.2.2	Object References	253
5.1.3	Distributed TCB	254
5.1.4	SDOS Covert Channels	255
5.1.5	Problems Arising from Heterogeneity	257
5.1.5.1	Heterogeneous Networks	257
5.1.5.2	Heterogeneous Hosts	257
5.1.6	Problems Related to Object Replication	258

5.2	Tasks and Lessons Learned	260
5.2.1	SDOS Security Policy	260
5.2.1.1	Read-down and Write-up	261
5.2.1.2	Object Replication	261
5.2.1.3	Restriction: Hook-up Security	262
5.2.1.4	Configuration Policy	263
5.2.2	Design	263
5.2.2.1	Overview of Design	263
5.2.2.2	Enforcing Security	265
5.2.2.3	Host Operating System Security	267
5.2.2.4	Network Security	268
5.2.3	Formal Methods	269
5.2.3.1	A New Security Methodology	269
5.3	Possible Future Directions	271
5.3.1	Prototype Implementation	271
5.3.2	Research Into Layering	271
5.3.3	Research into formal methods	272
A	The Gypsy Specifications	275
A.1	Notes on the Gypsy Specification	275
A.1.1	Conventions used in the Gypsy FTLS	277
A.2	Global Type Declarations	278
A.3	Global Function Declarations	282
A.4	The File Manager Specification	283
A.4.1	Local Function, Procedure and Type Declarations	283
A.4.2	Main Procedure	290
A.5	The Catalog Manager Specification	299
A.5.1	Local Function, Procedure and Type Declarations	299
A.5.2	Main Procedure	309
A.6	Authentication	314

A.6.1	The Authentication Manager Specification	314
A.6.2	The Terminal Interface Process Specification	320
A.7	The Kernel Specification	326
A.7.1	Local Function, Procedure and Type Declarations	326
A.7.2	The Message Switch Specification	327
A.7.3	The Security Database Specification	337
A.7.4	The Object Database Specification	345
A.7.5	The Process Manager Specification	352
A.8	The System Specification	356
B	Transformed Specifications for the File Manager	358
C	Glossary	375
	Bibliography	379

Executive Summary

This report is the final revision of a working document describing the Secure Distributed Operating System Design Project. It presents the project goals, the approach we have adopted for solving the technical problems we have encountered, and the project results. We have prepared this report with the expectation that it will be read by a diverse group of people. Its organization isolates different aspects of the project, such as expected results, preliminary results, and technical issues, thereby facilitating the identification of the key areas of interest to its various readers. The reader who is interested in a moderately-detailed technical summary of the project is directed to Chapter 5. Final Report. It is addressed to a reader who has some familiarity with computer security but who has not read the other chapters of this report.

Chapter 1 introduces and defines the problems that we are addressing. Its sections discuss security and distributed systems in general, and the relationship between the two, and they outline our approach to the problem.

Chapter 2. Policy, discusses the security policy in both informal and formal terms. Section 2.1 contains the security policy for the SDOS system. The security policy section is divided into three parts: motivation, a generic security policy, and an instantiated security policy. The motivation presents factors that played an important role in the formation of the policy, such as the perceived threats to the system and the important security features the system needs to have.

The generic policy is a set of rules that provide an abstract definition of security. The policy is based on the proposition that message passing is an appropriate model for describing interactions in a secure distributed computer system. The abstract nature of the generic policy is useful for gaining an intuitive understanding of secure information flow. The core of the generic policy is based on the property called *restrictiveness*. It places a constraint on possible information flow passing through multi-level secure entities in terms of histories of message-passing operations. Entities that satisfy the restrictiveness property separately are assured to satisfy the property as a group of interacting entities. This *hook-up* property allows the system policy to be decomposed into the policy enforced by individual components of the system.

The instantiated policy is the generic policy applied to a particular computer system, in this case an object oriented secure distributed operating system. The instantiated policy augments the information flow rules of the generic policy with rules constraining the use and control of abstract resources. The rules that make up the instantiated policy are organized into three groups: a mandatory policy, a discretionary policy, and a configuration policy. Section 2.2 describes the assignment of values to the parameters of the different policies.

The formal model (Section 2.3) is the formalization of the rules stated in the instan-

tiated security policy. The model is intended to state the properties of the system being formally verified. The formal model attempts to express policy constraints in terms of the extrinsic properties of the system (i.e., its external behavior) rather than its intrinsic properties (i.e. internal state). By describing behavior rather than internal states we are consistent with the philosophy behind data abstraction, and we are placing less constraints on the system's design.

Chapter 3 contains the design for a system that implements the security policies of Chapter 2. It contains a functional description, discussions of host and network security, a Descriptive Top-Level Specification (DTLS), and internal design details. The functional description provides a high-level look at the components of SDOS providing both discretionary and mandatory access control. Included in this section is a description of the security kernel and several *object managers* that provide symbolic naming, authentication, and configuration management services.

Chapter 4, Formal Methods, contains the Formal Top-Level Specification (FTLS) and discussions of the verification methodology and problems unique to verification of distributed systems.

Chapter 5, Final Report, briefly restates the problem and highlights some of the problems (and possible solutions) that are unique to distributed system security. It goes on to summarize Chapters 2, 3, and 4, and it concludes with some suggestions for future work in this area. As stated above, it is intended to be readable by persons having some familiarity with computer security, who have not read the other chapters of this report.

The actual Gypsy specifications are contained in Appendix A. Appendix B contains the transformed specification of the file manager. The transformation is a technique described in Chapter 4 to verify the FTLS. Appendix C contains a glossary of acronyms used in this report.

Revision History

Initial version, February 1986. Revision 1.1, July 1986:

A description of the Formal Model was added as Section 7 (causing subsequent sections to be renumbered). A summary of the formal model was added to Section 3. The contents of Appendix B were moved to Section 6.3.4.4. Significant technical changes were made in Sections 6.2 (A Generic Policy for Multi-level Security), and 6.3 (Instantiated Policy for SDOS). Miscellaneous changes were made in Section 5, Appendix A, and References.

Revision 1.2, April 1987:

The Functional Description was added as Section 7 (causing subsequent sections to be renumbered). A new section, "Assigning Values to the Security Policy Parameters" was added as Section 9 (causing subsequent sections to be renumbered). A significant change to the policy and formal model was made, eliminating automatic reclassification and the request-to-read machinery. This caused Section 6.2.3 and Appendix A to be eliminated, and numerous changes to be made throughout Sections 6 and 8 (formerly 7), especially Sections 6.2, 6.3, 8.1, 8.2.3, 8.3, and 8.5.

Revision 1.3, June 1987:

A section on Implementation Strategy (7.1.1) was added to the Functional Description. Editorial changes were made throughout the report, to fix typographical errors, and to bring all references and examples into consistency with the policy and formal model changes that were made in Revision 1.2.

Revision 2.0, October 1987:

This revision was produced with a different text-formatter than the previous revisions; the new formatter has features more suitable for the mathematical notation used in the formal specifications. Another major change is that the report has been reorganized into five chapters plus an appendix. The sections were renumbered as subsections within chapters, and some reordering has taken place. Significant new material has been added in the areas of formal methods (FTLS and Gypsy specifications), network and single-host security, and design details, both external and internal. Old Sections 1 through 5 make up the new Chapter 1. Old Sections 6, 8, and 9 make up the new Chapter 2. Old Section 7 is part of the new Chapter 3, along with much new material. Old section 10 was completely replaced by the new Chapter 5. Chapter 4, Formal Methods, and Appendix A, Gypsy Specifications, are all new material.

Revision 2.1, January 1987:

This last revision was generated at the conclusion of the project. Appendix C containing a list of acronyms was added, and a number of clarifications and corrections were made. No new sections were added.

Chapter 1

Introduction

1.1 Overview

There has been a considerable amount of both theoretical and applied work done in the area of multi-level computer security. The majority of this effort has been directed at the development of security policies and mechanisms for centralized computer systems. The little work in the area of security in distributed computer systems has either been focused on a small portion of the problem, or has been concerned with a very low level of abstraction, such as the secure transmission of messages in a network.

The Secure Distributed Operating System Design Project, funded by the Rome Air Development Center, was a 28 month investigation of multi-level security issues that relate to the development of a secure distributed operating system (SDOS) using the object-oriented Cronus DOS as a baseline. The project was a collaborative effort between Bolt Beranek and Newman Laboratories Incorporated, who have experience in the development of distributed operating systems, and Odyssey Research Associates, whose expertise is in the area of verification and computer security.

The goals of the project were twofold. First, we hoped to improve our understanding of the problems of developing an SDOS that demonstrates a set of high level requirements, such as availability and scalability. There are a large set of problems that need to be addressed in this context, many of which are interrelated and cannot be considered in isolation. We intended to qualitatively evaluate the relative importance and difficulty of addressing each of these problems using four criteria: performance, functionality, feasibility, and usability. We expected that such an analysis will yield insight into the trade-offs faced by the designer of any such system.

This project was viewed as a first step in the development of an SDOS that incorporates advanced distributed system technology, including high-level resource management and structured software architectures. Therefore, the second goal of the project was to produce the preliminary documents required for the system implementation. These documents include the *security policy* (or policies), which describes the access and flow of

information in the distributed system; the *formal model*, the mathematical formalization of the security policy; the *formal top-level specification*, developed from the system's design and used to verify that the functionality of the design is consistent with the assertions of the model; a *functional description* of the system, describing the major components of the system, their architecture, and our approach toward its design; and the *design* that forms the basis for a subsequent implementation.

This report serves as a repository for documents produced by the SDOS project. As new work was produced it was included in this report. In addition, three papers, [Casey et al. 88], [Weber87], [Vinter 88], were published describing the results of this effort.

1.2 Security

A secure computer system is a computer system in which all access and use of information and other system resources requires authorization, and there is assurance that authorization is performed correctly. The rise of interest in secure computer systems and the form these systems currently take are a result of four factors:

- Information sensitivity:
there are applications in which information is sensitive, and access to that information needs to be controlled.
- Benefits of computers:
many of the benefits of using computer systems for processing non-sensitive information are common to the the processing of sensitive information.
- Paper security:
a set of procedures are in place for specifying and using sensitive information in the paper world (as opposed to the electronic world), and most secure computer systems are modeled after these procedures.
- Automation:
computers are commonly used to automate tasks previously performed by people; many of these tasks are critical and there must be a high level of assurance that they are performed correctly by the computer.

Computer systems with security as a property have two advantages over other systems. First, they enhance the likelihood that information will not be compromised or corrupted (note that information security, like software correctness, can never be absolute in a complex system considering the major role humans play in the use of secure information). Second, secure systems provide greater precision in specifying and constraining legitimate information use. This precision contributes to overall software correctness by decreasing the number of possible ways that information can be used illegally.

We see the user of a secure computer system (i.e., the person responsible for selecting the system and/or developing applications for it) as being concerned with two aspects of the system. First, the user is interested in the adequacy of the system to protect the sensitive information maintained by the applications running on the system. An understanding of the definition of security in the context of this particular system is needed, along with knowledge of the level of assurance that the security-related portions of the system are correctly implemented. The definition of security is found in the security policy of the system (see sections 1.3 and 2.1). Level of assurance is difficult to measure quantitatively, and is derived by demonstrating that the formal specification (implementation) of the system meets the formal model (security policy requirements).

The second aspect of a secure computer system relevant to a user is the functionality of the system; that is, the degree to which the system can be used to implement a software system that allows the desired use of sensitive information. This is the general problem shared by users of all computer systems. As we will demonstrate throughout this report the security aspects of the system have a major affect on overall system functionality.

To determine the adequacy of security in a computer system, people commonly considered the threats anticipated to the system and mechanisms that are used to prevent those threats. Both hardware and software mechanisms can prevent information corruption and compromise by enforcing at least two sets of rules: *mandatory access rules* and *discretionary access rules*. These rules restrict the conditions under which information may be accessed. Mandatory controls restrict access based on the designated sensitivity of the information and the level of clearance of the user. Systems that allow the manipulation and storage of information which have varying degrees of sensitivity are referred to as *multi-level secure*. Discretionary controls specify how information can be accessed by a particular user. Discretionary access privileges change dynamically at the discretion of the subjects in the system; mandatory controls are a predefined set of rules constraining access to system resources.

There are several secondary features that secure computer systems commonly have which relate to mandatory and discretionary controls. These include rules concerning how information containers are reused, how sensitivity level labels are associated with information, how authentication is performed, and how accesses are audited. The DoD Trusted Computer System Evaluation Criteria [DoD Criteria 85] is the authoritative document on the classification of centralized secure computer systems based on their features and methods of development. Though no evaluation criteria document existed for secure distributed systems as the beginning of this effort, the Trusted Network Interpretation [NCSC TNI 87] arose later. This is briefly addressed in section 3.4.

1.3 Project Direction

We conceived of the project as consisting of five partially overlapping phases:

1. Narrowing the orientation of the project
2. Decomposing the project into separate areas and enumerating the important issues in each area
3. Developing a security policy for the SDOS
4. Developing an informal system design and system/subsystem specification
5. Formalizing the policy, producing the formal model and the formal top level specification.

Next we provide an overview of each phase.

1.3.1 Phase I: Narrowing the Orientation of the Project

Given the large scope of this project, our earliest efforts were directed at narrowing the orientation of the project. Distributed operating systems vary widely in the set of requirements they satisfy, the functionality they provide, and the software architectures they support. The security problems that need to be addressed with each system depend on the properties of the system. For example, it is easier to ensure the secure access to information in a simple DOS that provides only fundamental services such as message passing than in a DOS that provides a high degree of functionality, simply because of the difference in complexity of the two systems.

We believe that developing an arbitrary SDOS will have little value because of the heavy impact the characteristics of a DOS have on the security problems that are relevant to the system. A more fruitful endeavor is to identify the particular DOS requirements of interest, and address the security problems faced in the development of the system that satisfies those requirements.

The Rome Air Development Center has been supporting the development of the Cronus DOS at BBN since 1981. The first step in the Cronus project was to identify a set of initial system requirements that were considered important by RADC and necessary to exploit the potential benefits that distributed computer systems have over centralized systems. These requirements, along with a discussion of the purpose and value of a DOS, are the subject of section 1.4. We have narrowed the orientation of the SDOS project by limiting ourselves to only consider systems that satisfy these distributed system requirements.

1.3.2 Phase II: Decomposing the Problem

After narrowing the orientation of the project, we decomposed the project into a set of distinct areas that could initially be independently investigated. The purpose of this approach was to identify the issues relevant to the project without attempting to filter

them based on our evaluation criteria. In doing so, we treated the problem as a search space and we began a breadth first search of the possible issues that might be relevant. The results of this search form the basis for section 1.5. The decomposition yielded three separate areas, as described in the following sections.

1.3.2.1 Security Requirements

There were two sources of ideas for the security requirements of a DOS: the results of the work of experts in the security field, and the security requirements of a class of applications which is representative of the applications that the DOS is intended to support. We next look at these ideas in more detail.

A large body of work describing the important security properties of secure computer systems is available. Several perspectives have been taken including the formation of security policies to describe the secure use of information, the development of formal models that can be used in the verification of a computer system, and the examination of threats to systems and how they can be thwarted.

An effort of significance to this project is the DoD Computer Security Center attempt to develop a trusted network evaluation criteria. The 1985 National Workshop on Network Security [Workshop 85] was organized to provide input toward this effort by over 70 of the nation's security experts. The results of the workshop were a list of issues and position papers; these results provide a valuable snapshot of the most up-to-date views of security experts on security in distributed systems. This work contributed to the eventual generation of the Trusted Network Interpretation (TNI) [NCSC TNI 87] of the Trusted Computer Security Evaluation Criteria [DoD Criteria 85]. Other relevant work includes secure operating systems designed or developed, such as KSOS [McCauley and Drongowski 79] (see [Landwehr 83] for a complete list) and security policies, including Landwehr's MMS policy [Landwehr et al. 84], Sytek's MLO policy [Sullivan 86], and a non-interference style hook-up security policy developed at ORA [McCullough 87].

The purpose of developing an SDOS is to allow applications to use the resources that span the entire computer system in a secure fashion. Occasionally some applications have their own security requirements, realized in an application security policy, that are different than those of the operating system. The degree to which the SDOS security policy matches the application security policy in part determines the usefulness of the DOS for the application. Ideally, the SDOS security policy subsumes the application policy. If this is not the case, the SDOS should be flexible enough to allow additional security mechanisms to be built on top of it to satisfy more stringent application requirements.

A set of Command and Control (C2) applications potentially needed by the Air Force were chosen as representative applications that the SDOS is intended to support [Berets et al. 85]. The reasons for choosing the C2 application are that they span many types of computer systems and require survivability, scalability, and interoperability.

Second, they involve diverse aspects of the use of secure information including collection, selection, aggregation, and analysis. Additionally, these applications involve monitoring and controlling physical devices that collect and use secure information (e.g., radar). Finally, the complexity and size of automated command and control systems require a software methodology to support modular and evolvable system development.

Our results indicated two unusual aspects of these applications. First, automated data collection yields massive amounts of data whose sensitivity is frequently time dependent. An automatic downgrading facility would minimize the overclassification of this data. Second, a full command and control system is naturally a feedback system, consisting of four parts: collection, aggregation, analysis, and control. We hope to either develop a policy that accommodates a command and control system of untrusted components or develop a feasible system architecture for the application where trust is well specified and highly constrained.

1.3.2.2 Security and Distributed System Functionality

Another area that we investigated is the impact that security requirements and mechanisms will have on the wide range of functions required to make a distributed computer system fully operational. The most obvious effect of incorporating security mechanisms into an operating system is severe degradation of performance. For this reason, performance is a criterion for evaluating the set of requirements and problems to be addressed, and the set of mechanisms to be adopted.

One property secure systems should have is uniform and thorough access authorization of all uses of system resources. Such authorization is possible only by using precise discretionary access control within the operating system itself. An example of the need for precise controls occurs in the monitoring and controlling of the major components of the distributed computer system. A key component of this service is called configuration management (different from configuration management of the entire software development lifecycle of the secure computer system). The configuration manager is responsible for controlling the attributes and the state of hosts in the system, including the system services that execute on those hosts. This component plays an important role in a secure system, since it determines the initial state of a host and how the services that host provides can change over time.

There are several other areas where security needs have a major impact on the existing functionality of distributed system. These areas are discussed in detail in section 1.5.

1.3.2.3 Choice of Mechanisms

The set of mechanisms adopted in an implementation are selected for a wide range of reasons. They include ease of implementation, anticipated performance, system model, functionality, simplicity, development speed, and the desire to experiment. There are

trade-offs between any set of mechanisms. The choice between capabilities and access control lists is an example of such a trade-off for discretionary access controls. Capabilities are more complex to implement than access control lists, but they allow easy temporary privilege transfers. Access control lists make privilege revocation easy and facilitate the review of privileges needed to access an object on a per object basis. Which mechanism is best depends on the system requirements and the constraints placed on the development project.

In this project we identify many of the trade-offs between the mechanisms that are relevant to secure system development. We used the existing Cronus system as a basis for selecting mechanisms, for three reasons. First, since we implemented these mechanisms we understand them better than other mechanisms. Second, a version of Cronus was operational; this provides a head start in determining the viability of secure system alternatives. Third, the mechanisms in Cronus were adopted because they satisfy the basic system requirements that Cronus shares with the SDOS system being investigated.

1.3.3 Phases III and IV: Policy and Design

The policy and design phases were begun simultaneously, for two reasons. First, the abstract nature of a security policy makes it difficult to determine the feasibility of implementing the policy. This is important because of performance problems that have characterized secure systems. We expected that the additional insight that an early design can offer will be useful in the policy development process.

The second reason for beginning the two phases at the same time was due to the existence of the Cronus system that could serve as the basis for the SDOS design. Cronus represents available distributed operating system technology that, ideally, could be applied to the SDOS development. It was imperative that we determine the viability of using Cronus as a starting point for the system design at the earliest possible time. With a rough version of the policy and a preliminary design of the secure system that has evolved from the Cronus system, it would be possible to uncover any fundamental incompatibilities that might exist.

The difficulty of concurrent policy and design development is ensuring that they remain consistent. Therefore, an essential part of the development process was frequent comparisons of the most recent versions of the policy and design. Next we look at the policy and design phases separately.

1.3.3.1 The Policy

The purpose of a system's security policy is to define the meaning of security within the system. Policies are defined at many different levels of abstraction. The higher the abstraction, the closer it corresponds to our intuitive understanding of the meaning of security; the lower the abstraction, the greater its complexity and the more readily the policy can be implemented and demonstrated to be consistent with that implementation.

Since both understandability and precision/enforceability are important, we have produced a security policy with two parts: a high-level *generic* part that is independent of any particular system, and a low-level *instantiated* part. Though a policy can generally have many implementations, the details of the instantiated policy greatly constrain the designs that could be securely implemented. Therefore, the low-level policy is useful for developing a system design and later demonstrating that the design enforces the security policy.

Security is defined in a security policy through the description of an abstract system of entities and operations, and rules governing the conditions under which the operations may be legitimately used. The policy is formed, in part, by using an intuitive understanding of the meaning of security. As we have described, the security rules in the SDOS policy include mandatory and discretionary access rules. These rules roughly correspond to rules constraining who may communicate and what information is communicated, respectively, in the system.

Other factors in the formation of the SDOS policy included:

- Knowledge of the problems occurring with the implementation and use of other security policies.
- Understanding of how information will be used in the system, which we have based on our familiarity with other secure systems and our use of C2 Internet as the application domain.
- Insight into the technical distributed system issues that relate to security.

The SDOS security policy is used by application developers for expressing the security aspects of their application and formulating its design. The application developer uses the high-level security policy as a guide to associate security attributes with information to be manipulated within the application and to define who and under what conditions that information can be accessed. The low-level policy can be used during the process of designating the software modules that comprise the application and specifying their interaction.

How is a security policy judged to be adequate? What distinguishes a good security policy from a poor one? These questions have not been adequately addressed in security literature, and they remain open to discussion. When viewing a system security policy as the foundation for secure applications, it should have two characteristics. First, the policy should incorporate the most widely shared features of application security policies, in order to reduce redundant mechanisms. Second, the policy should incorporate those features needed by applications that can only be provided at or below the level of the operating system. We hoped to look in detail at methods of evaluating policies. Section 2.1 provides the current version of the security policy, and explains why this policy differs from the well known Bell-LaPadula model.

1.3.3.2 The Design

A system's design is the document used to guide the system's implementation. There are two major aspects of the design: to present a decomposition of the system into its functional elements, and to describe the manner in which the elements perform their function and interact. The decomposition of a system is typically called the system's *functional description* (see section 3.1); the manner of operation is called a set of mechanisms. Choice of both functions and mechanisms greatly determine the performance, usability, feasibility, and capabilities of the system's implementation.

The process of creating the design involves identifying the major functions of the system, assigning the functions to modules, determining the interaction patterns of modules, and choosing mechanisms to implement the functions. We chose to drive the system functionality by the basic set of DOS requirements rather than by the security policy.

The development of a design is an iterative process, requiring experimentation with different implementation ideas. We used performance, usability, and functionality as the criteria for judging these ideas. Performance corresponds to expected efficiency; measures of performance include the amount of interprocess communication, remote communication, context switches, and rough estimates of execution time.

Usability refers to how easy a system is to use, either from the standpoint of a software developer, a system administrator, or a user. Traditionally, secure systems have included features that detract from the usability of the system. For example, the AFDSC Multics system required users to log off and log back on simply to read electronic mail. There is generally a trade-off between the usability of a system and the system's functionality.

Feasibility refers to the ease with which a design can be implemented. The value of a design that cannot be implemented is dubious. Because of our interest in future implementation, we made design decisions based on how feasible they are to implement. A common example exists in secure system development. Many operating systems and applications have been designed which require major portions of the systems to be verified in order to satisfy the prespecified security requirements. However, the current state of verification tools makes it untenable to rely heavily on verification of large parts of a system.

Almost a decade of distributed system research and development has led to a fairly firm understanding of the needed functionality of distributed systems. One example is availability through data and functional redundancy. Another example is data consistency through coordinated distributed state changes and failure recovery techniques. What remains to be investigated are how security requirements and support for secure information will affect DOS functionality.

The purpose of the design phase is to generate a design document. However, we also expect this phase to yield an equally important document describing the trade-offs

pertaining to the major design alternatives uncovered during this phase. This document will be useful to future designers of secure systems and will provide insight into the implementation problems faced by developers of secure distributed systems.

The approach we took in developing the design was to isolate the mandatory security mechanisms within two facilities: the mechanisms that pass messages between components on or across hosts, and the mechanisms that allow the storage of objects in the object database. Discretionary access checks will be performed both within object managers (for abstract operations) and between object managers and the object database (for simple read/write object operations).

1.3.4 Phase V: Formalization

Formalization is the process of using the language of mathematics to express and reason about a system that was previously expressed informally. There are several steps taken in the formalization process. First, the policy must be formalized: this is called a *formal model*, and is presented in section 2.3. Second, the design must be formalized: this is usually called the *formal specification*. Last, a formal proof of correspondence between the formal model and the formal specification must be constructed. This proof shows that the (formal) design correctly implements the (formal) policy.

The effort to develop an SDOS is significantly different from the design of other distributed operating systems due to the need to have greater assurance that the system's design meets its requirements. The need for greater assurance is based on the importance placed on the system security requirements. As is commonly the case in the development of secure systems, we will gain a greater precision in our analysis of the system design by using formalization and formal verification techniques.

Conceptually, the process of formalization should begin after the system policy and the design have been completed. In this project, however, we began formalization at an earlier point. After the system policy was formulated, we had an informal definition of security in the context of the DOS and a clear idea of the direction of the design. Beginning formalization at this point was largely independent of the policy's initial development. In contrast, the formal specification should proceed concurrently with the informal design development, since the two affect each other: the design drives the formalization of the design, and the resulting formalization and attempts at proof identify design security flaws and other design problems that must be corrected.

The relationship between design and policy is formalized by stating them in formal mathematical languages that allow a precise analysis of their interrelationship. There are many languages for representing the policy and design (for example, the formalization of the security policy can be expressed with the mathematical language of sets and relations), but few languages have sophisticated automated tools to aid in the development of proofs and the checking of their validity. This automated support is vital since the design of a large system, and correspondingly the proof that it is consistent with a policy, is complicated. We chose to use the Gypsy system [Good et al. 78] because it

provides a well developed environment for automated support and is widely used. Of the widely known verification systems, Gypsy currently offers the greatest flexibility and the most complete set of tools. This flexibility allowed the formal specification to closely follow the actual design of the system.

1.3.4.1 Formal Model

Security is defined in terms of a system's abstract entities and its abstract operations. The system is secure if the histories of abstract operations in which it may possibly engage and the possible states of its abstract entities satisfy a set of constraints. The formal model is a formal statement of these constraints.

Since the constraints of the formal model are exactly those properties to be proved about the formal specification of our specific system, the model must involve concrete entities and operations of the system. The formal model must therefore be a formalization of the instantiated, system-specific, security policy. This tends to make the model less directly understandable than the generic policy; our approach was to argue that the formal model is the appropriate set of constraints since it is derived from the more intuitive security policy. Of course, since many concrete system entities are not known at the time the policy is formulated, as much abstraction is retained in the model as is possible.

The formal model may include both security properties and other properties of interest. Security properties usually concentrate on restrictions on information flow and on various types of access control. Other properties might include no-denial-of-service and assertions about survivability. We chose to treat these properties as options to pursue if the basic security properties could be demonstrated within the time bounds of this effort. No-denial-of-service properties, in particular, are not easily stated and proven within Gypsy.

1.3.4.2 Formal Design Specification

The system design was recast in a formal language to produce the formal specification. The main issue involved in this translation is the level of detail of the specification. We expect that the design and specification would incorporate high-level algorithms and interface specifications between the major system components, but not the implementation-level code. Algorithms can be expressed in Gypsy. We expect that the more detailed levels of the specification would bear a strong syntactic resemblance to the informal design. However, the top level of system specification, which describes the DOS as consisting of processors executing in parallel and communicating over some medium, cannot be directly expressed in Gypsy. The limited constructs Gypsy provides for concurrency required that artifices be used to model the system architecture and high level of functionality.

Since the Cronus system was used as a baseline for modeling the SDOS, we intended

to include three fundamentally different kinds of hosts in the formalized design: native SDOS hosts; hosts supporting SDOS on a constituent operating system with known properties; and hosts with no known properties connected to the communication medium via a secure access machine. In the first two cases, security depends on properties of the underlying system: the native SDOS host relies on properties of the hardware architecture to support security, while the SDOS running on a constituent operating system depends crucially on the properties of that operating system. These underlying properties must appear as part of the formal proof.

The Cronus system emphasizes the operability between hardware and software components in a system. Since SDOS, like Cronus, is expected to evolve, the details of supporting hardware and constituent operating systems cannot be determined in advance. It was not our purpose to have the details of the design reflect a particular architecture. The formal specification specifies a "generic" hardware architecture and a generic constituent operating system architecture to which the formal design could be related.

1.4 Distributed Operating Systems

The purpose of a distributed operating system is to promote and manage resource sharing among interconnected computer systems by providing coherent and integrated tools. Coherency and integration allow the development and use of distributed applications that are able to exploit the advantages that multiple interconnected computer systems have over centralized systems, without concern for the details of each computer in the system.

The field of computer science has over a decade of experience in working with interconnected computer systems. Despite this experience, there has been relatively little progress in distributed application development. There are several reasons for this lack of progress. Many networks consist of a heterogeneous set of computers that have incompatible devices, communication protocols, system software, and data representations. The complexity of distributed software running on these systems far exceeds the complexity of centralized software with similar functionality. Yet heterogeneous systems are a necessary result of the need for specialized computers and the desire to keep up with the quickly changing hardware technology. Distributed system applications also cover an extremely wide range of problem areas, and no single system is appropriate for the wide variety of system requirements.

From these observations we drew two conclusions. First, uniformity and abstraction are essential attributes of a distributed computer system that reduces the complexity of the system and hides the details of each computer in the system. In particular, we were concerned with the complexity of such basic services as interprocess communication, access control, naming, data storage, and data retrieval. A general framework for structuring and interconnecting components throughout the system is an essential means of simplifying the system. An open system design, where applications are constructed

within the same model as operating system services, further enhances overall system uniformity.

Second, no single, comprehensive distributed system architecture will be appropriate for all applications. Rather, a specific set of properties are necessary to serve as the system requirements that informally guide the system design and development. Based on the set of applications identified by the Rome Air Development Center, we have adopted the following properties that the distributed operating system should exhibit:

- **Survivability:** the integrity of data and overall system operability should not be affected by partial failures.
- **Scalability:** the system architecture should accommodate the scaling of system resources.
- **Global system resource management:** system resources, such as operating system services and devices, should be controlled through a single, uniform facility for the entire DOS.
- **Interoperability:** it should be possible to integrate software in a machine-independent fashion, allowing the underlying hardware to change without redesigning and reimplementing applications.

There are few software architectural methodologies. In the initial Cronus system design work we discovered that only one methodology was rich enough to support all of the properties listed above and comprehensive enough to address all of the basic services we were concerned with in a uniform and abstract way. This model is called the *object model* [Jones 78] and was adopted as the framework for structuring software in Cronus. Although particular Cronus mechanisms can easily be replaced by new ones, the model is a fundamental, inseparable part of the Cronus system. SDOS is also based on the object model.

Objects are instances of abstract data types and correspond to logically addressable resources, such as data and physical devices. The type of each object defines the set of operations on the object; these operations are the only means of accessing the object. Object operations are implemented by object managers, which hide the representation of the object from its accessors.

Abstraction is inherent in the model by hiding the representation of objects in modules having precisely defined, high-level interfaces. Uniformity derives from using the object to represent all resources in the system including data, system services, and devices. All interprocess communication is achieved by performing operations on objects, independent of their location, managers' identities, or representation. Discretionary access control is made uniform by mapping operations to access rights and performing authorization on each operation invocation. Since all accessible entities are objects on all machines, it is possible to create a global name space at both the system and user levels. Data storage and retrieval are confined within managers, and made uniform

through manager development tools and a well defined interface to an object database repository. Last, software extensibility derives from the creation of new object types from existing types, with application specific interfaces.

1.5 Distributed System Security Problems

After examining literature in the security field, exploring the security needs of the C2 Internet project, and considering the impact of security on the major components of a DOS, we were able to collect a list of problems that needed to be addressed by this project. This collection, the result of Phase II, was then organized into a set of problem areas. Each of the following sections corresponds to one problem area.

1.5.1 Objects and Managers

Objects are abstract entities that are accessed with operations defined on a type-by-type basis. In contrast, information flow rules are commonly expressed as low-level operations, such as read and write, between any pair of entities in the system. The major problem is how to enforce information flow constraints in an efficient manner without expressing each operation on an object in terms of reads and writes. Our early work indicated that it may be possible to limit information flow in different ways in two different places: between clients and managers and between managers and the object database.

Related issues include the definition and implementation of multi-level secure entities (these "trusted" components are intuitively defined as having the ability to make sensitive information available to clients under controlled circumstances), how discretionary access controls are enforced, and how multi-level objects and types can be supported. We believe there is an important relationship between the correct implementation of operations on objects and the correct implementation of discretionary access controls. Software development tools that automate much of the programming of applications may contribute to the correct implementations of both operations and access control.

1.5.2 Communication and Identification Integrity

Secure communication between processes on separate hosts requires that the operating system maintain the integrity of messages and information about their source of transmission. The major issues are what information must be transferred with integrity, and what techniques are available to help establish message integrity. Encryption is one means of ensuring host-to-host message integrity, and will be an important component in the solution to the integrity problem.

1.5.3 System Environment

There are two aspects of the distributed computing environment: heterogeneity caused by the types of hosts in the system, and operating system integration caused by having the secure, distributed operating system co-resident with existing, centralized operating systems on particular hosts. The heterogeneity problem must be addressed in the policy, by defining a logical mapping between the security entities (such as object labels) in one system and comparable entities in other systems, and in the implementation. These issues require that we define the nature of interactions between machines with different security policies, and the resulting system-wide security policy.

Operating system integration within a single host has been successful only in non-secure systems. The feasibility of co-residency is dependent on the hardware support available to separate the two systems and the assurance that the policy of the secure operating system cannot be compromised by the other system's presence.

1.5.4 Configuration and Security Management

Distributed operating systems commonly result in the distribution of control. System-level control is evident in two ways: the management of security information and the management of system services. These controls are commonly vested in the security administrator and system administrator, respectively. The security administrator is responsible for defining and maintaining the security attributes of system services, objects, and users. The system administrator is responsible for installing, monitoring, controlling, and removing system services. Facilities are needed to explicitly state and enforce these roles. Issues include how machines are downloaded, the dynamicism required, and how applications are distinguished from system services in both the policy and the design.

1.5.5 Feedback Applications

Many of the applications of interest to the Air Force are command and control applications, where sensors are used to detect conditions in an environment, and this information is collected and analyzed. The results of the analysis are then used to command remote sensing and weapons systems. The command aspect of such applications are feedback loops, and present a difficult problem in the context of application on a secure computer system. The problem stems from the necessary flow of information downward in security levels in the control phase of the application. This downward flow violates the widely accepted Bell-LaPadula information flow rules that only allow upward information flow. What is needed is a precise characterization of how information can flow downward as part of the controlling phase of a feedback system. The issues here are how to designate this particular instance of downward information flow as legal, and how this designation is related to the discretionary access controls which determine who

can issue the command and control operations that result in the downward information flow.

Chapter 2

Policy

2.1 The Security Policy

The purpose of a security policy is to define the meaning of security within a system. A policy presents an abstract view of a system, and defines, in terms of that abstraction, the properties that the system is intended to enforce. As much as possible, a security policy should list the intended system properties but avoid describing mechanisms that implement those properties. Loosely, properties describe what a system must do or not do, whereas mechanisms tell how tasks are accomplished. Although there is no solid dividing line between properties and mechanisms, a policy aims to express the goals of a system in the most all-encompassing manner that is still enforceable within the computer system. Note that a policy can state enforceable properties for many aspects of system behavior. Since this document is specifically a *security policy*, it is the document that defines the breadth of the interpretation of *security*.

The terms *property* and *rule* will be taken as roughly synonymous in referring to specific elements of this overall security policy. The terms *goal* and *requirement* will be used to describe the high-level motivations behind specific rules and properties. No attempt at formality has been made. A formal expression of the policy in mathematical language is provided in the *formal model* (see section 2.3). The security policy is a statement of the system's security rules, described in non-mathematical language, that can be referred to during the later system design and formalization phases. Strategies for demonstration and verification, formal or otherwise, of the desired properties are also discussed later (see section 4.2) and are not included as part of the policy.

Many properties are expected of distributed operating systems; some are related to security and many others are not. It is not clear *a priori* which of these properties should be included in a security policy. One approach is to adapt the security policy of another system. However, we knew of no instance of an already existing security policy for a DOS that would be entirely adequate. Instead, we examined a distributed application currently under development in order to motivate our definition of security.

We combined this motivation with our knowledge of existing security practice in systems other than DOSs, and with our knowledge of distributed systems technology. The result is a security policy that should be applicable to a wide range of DOS designs.

A security policy may be defined at many different levels of abstraction. The more abstract the level, the closer the policy corresponds to our intuitive understanding of the meaning of security; the less abstract the level, the greater the policy's complexity and the greater the contact between it and the secure system's implementation. Since both understandability and detail are important, we produced a policy with two parts:

1. A high-level, generic part that is independent of any particular system;
2. A lower-level, system-specific instantiation, applying the generic part to the particular case of SDOS, and including other system-specific policy requirements.

Rather than use the well-known and widely-used Bell-LaPadula generic policy of multi-level security [Bell and LaPadula 76], we chose to adapt the multi-level security policy of McCullough [McCullough 87]. The resulting generic policy has substantial differences from Bell-LaPadula. The salient features of our generic policy that distinguish it from the Bell-LaPadula policy are:

- No distinction is made between the activity and passivity of system components in the generic policy; any system component able to interact with other components will be called an entity.
- Instead of using read and write operations to describe information flow in the system, the generic policy uses a single send-message operation. This operation comes closer to modeling network interactions between entities (e.g., hosts, processes, and devices) in the distributed computer system.
- System entities which are expected to handle data of many security levels will satisfy the McCullough *restrictiveness* property. This property emphasizes control of information flow, rather than access control, and is greatly distinguished from Bell-LaPadula for this reason. In addition, it is also a "hook-up property", meaning that a collection of communicating entities satisfying the restrictiveness property will form a larger entity which is also restrictive. By repeated hook-up of system entities, a constraint on information flow for the entire system can be built up from constraints on individual components. This method of decomposing global security into local security properties is very useful for analysis of distributed system security.

The instantiated security policy for a particular system consists of three parts: a mandatory policy, a discretionary policy, and a configuration policy. The mandatory policy is a straightforward application of the generic policy to the entities defined for SDOS. The discretionary policy limits the access to abstract resources on the basis of the identity of the accessor. The configuration policy specifies the parameters used to

configure system security and the rules governing the security dynamics of the network. The salient features of the instantiated security policy are:

- Secure information flow between entities is maintained for all object-oriented interactions among users and system resources.
- It is permissible in some cases to extend the functionality of the system after its initial configuration by adding new trusted and untrusted software to the system.
- Specially designated users are provided with the privilege to change security relevant parameters governing the creation, labeling, and actions of entities. As a result, the security policy can be customized before, during, and after the installation of the system.
- The spread of discretionary rights is curtailed, controlled by a select group of users for each resource.
- Critical operations on system resources can be designated as requiring manual execution only, thereby ensuring user intervention for the initiation of critical functions.

In section 2.1.1 we provide a rationale for the security policy and for some of the policy decisions we made. This rationale shows that many of the rules of the security policy are motivated by threats to the secure operation of SDOS and of applications which it supports. It also argues that the rules meet these threats in many cases. The generic policy is described in section 2.1.2. In section 2.1.3 we present the instantiated security policy, composed of rules governing information flow, access to and control of abstract resources, and system configuration for an object oriented distributed operating system.

2.1.1 Motivation

In the development of SDOS, we followed this progression:

1. Study a conceptual model of the system to find threats to its proper functioning;
2. Formulate a policy that prevents those threats;
3. Give assurance that the system, as designed, meets its policy.

To judge which threats are of concern for SDOS, we considered a prototypical application developed for an existing DOS. The Command-and-Control (C2) Internet Experiment, implemented at BBN, is a demonstration application intended to run under the Cronus distributed operating system [Berets et al. 85]. There are security considerations for C2 Internet that potentially affect its design, and the design of many similar

applications. Since SDOS could conceivably be used to support C2 Internet, these security considerations for C2 Internet may be used to motivate some design choices for SDOS. In the next section, we will briefly describe the purpose of C2 Internet. In section 2.1.1.2 we will identify some of the threats to its security, and indicate what sort of security policy for SDOS could be used to meet these threats. In section 2.1.1.3 we will discuss assurance of policies for secure systems, including SDOS.

2.1.1.1 The C2 Internet Experiment

The C2 Internet Experiment is intended to model some possible distributed Command-and-Control applications. Using C2 Internet, one can demonstrate and experiment with the features of the Cronus distributed operating system that support distributed applications. In particular, the Experiment models the collection, integration and control of data on military ground force movements. It uses that data in several ways: to identify targets automatically, to assess how best to apply resources to them, and to provide both raw and processed data for commanders to make their own assessments. Although the human involvement in the battle management process is vital, the development of C2 Internet Experiment is being driven by the need to automate parts of the process. The quantity of data that can be collected is too great to allow human processing of all of it. Therefore, automatic data processing is used to reduce the volume of data reaching the user. Automatic data processing should also result in shorter response times to changing battlefield conditions.

The C2 Internet collection of applications is divided into four phases. Each phase is typical of one aspect of command-and-control systems.

1. Data is collected from various sensors. These are generally conceived as airborne collectors of radar and infrared images. Typically, C2 Internet will exploit its distributed environment by allocating several different processors to the task of collecting sensor data.
2. Data from various sensors is fused together into a coherent, multi-sensor view of the external world. Although data from each sensor may first be analyzed in isolation, providing a sensor-oriented description of the battlefield, eventually data will be combined to produce a target-oriented description, with concise target reports. The data fusion process will also make use of intelligence data not necessarily collected from C2 Internet sensors: weather data, prior knowledge of terrain, and prior knowledge of infrared or radar signatures of targets.
3. Target reports are analyzed to aid in battle management functions and planning. These activities include: extrapolating from the current battlefield situation toward the future, evaluating the threat posed by observed targets, and generating recommendations for efficient scheduling of available C2 resources.
4. Battle-management decisions are used to control future allocation of C2 Internet resources. This results in a feedback loop in which the current analysis of the

battlefield situation determines how C2 resources are used, and the effectiveness of the current plan is assessed to determine whether it needs modification.

2.1.1.2 Threats and Policy Requirements

2.1.1.2.1 Unauthorized Access: Multi-level Security Some of the data contained in any command-and-control application such as the C2 Internet Experiment will be classified. For example, intelligence data may be highly sensitive. Data contained in the form of programs may also be classified: from them, one may be able to infer the capabilities of friendly forces. Yet other data, possibly the weather reports, will be unclassified. The simplest means for preventing unauthorized access is to consider all data to be of the highest classification known to the system, and to require all users to be cleared for that classification. The problem with this solution is the extra cost of maintaining large amounts of classified data, and of clearing personnel to perform C2 functions.

The alternative chosen for this policy, the multi-level security (MLS) solution, has the computer system enforce trustworthy access controls to its resources. Containers for data will be tagged with a security classification, and processes, users, and external devices will be tagged with a clearance level. A policy restricting access on the basis of classification and clearance levels is called a *mandatory policy*. The security policy requires that information will flow only to equal or greater classification levels. For example, it implies that the C2 Internet data fusion process, which combines information from sources possibly at different levels, will need a clearance level at least as great as each of its sources.

2.1.1.2.2 Preventing Overclassification Enforcing a multi-level security policy will at least eliminate the basic cause of overclassification: a need to change all security labels of information in a system to the highest security level. However, overclassification will still be common if *upgrading*, the relabeling of an entity to place it at a higher security level, is frequently used to restrict access to data to a smaller group of people. A partial solution to this tendency uses more fine-grained classification and clearance levels. Typically, *category sets*, which reflect need-to-know for certain kinds of information, are included as part of the policy for multi-level security.

A significant amount of the data processed by C2 applications becomes outdated within a short time, hours or days at most, due to changing battlefield conditions. If such data is archived rather than destroyed, it may not retain its original sensitivity, and therefore it incurs the extra expense of being maintained at the overclassified level. *Downgrading*, the relabeling of an entity to place it at a lower security level, on a selective basis is a solution to this problem. Because downgrading conflicts with the basic MLS policy rule that data flow only to equal or greater levels, policies for selective violation of this rule are needed.

2.1.1.2.3 Manual Control One use of computer systems is to accept, manage, and display information. When that information is sensitive, requiring use of secure computer systems, rules controlling information flow and preventing overclassification are important. Another important use of computers is to control other systems external to the computer system itself. Examples of computer controlled systems in the C2 application include sensor and weapons systems. The control of sensors is expected to be fully automated. The control of weapons systems, in many cases, will be partially automated (e.g., for controlling direction) and partially manual (e.g., for launching). On the other hand, many actions that have critical importance with respect to resource usage or destruction (such as destroying archived data) may require initiation by people in order to be performed with adequate safety. It is imperative in these situations that this manual control not be circumvented. The policy must include rules allowing uncircumventable manual control over resources accessible through the secure system.

2.1.1.2.4 Preventing Denial-of-Service One primary reason for automating C2 applications is to increase throughput and decrease the time from data collection to output of battle management planning. Threats to responsiveness are called *denial-of-service*.

The possibility that C2 application data could be corrupted or destroyed before it is used in battle management threatens denial-of-service. Integrity markings will be used to prevent unauthorized writing to or destruction of objects. The standard Biba extension to Bell-LaPadula adds these integrity markings to the classification labels of objects. The security policy will incorporate integrity markings that will be used to enforce the overall intent of the Biba extension: information flows only to the same or lower levels of integrity. Thus, for example, a process of low integrity will be prevented from destroying high-integrity data if each is properly labeled.

The Biba integrity property clearly solves only a small part of the denial-of-service problem. It is possible to satisfy the Biba property with a system that does no information processing. In general, preventing denial-of-service means that eventually, or within a certain amount of time, certain events must happen. This policy will not address the general denial-of-service problem further.

2.1.1.2.5 Discretionary Access Control Discretionary access controls are controls on system functions based on revocable authorizations that depend on the identities of users or processes. They are valuable for controlling the unauthorized use of information, the modification of that information, the use of external devices and systems and for confining the damage caused by accidental or malicious software or operator errors. Discretionary controls are a flexible method for isolating separate SDOS applications without assigning distinct and incompatible multi-level security labels.

The development of C2 Internet is tightly related to the philosophy of its underlying Cronus system, which stresses object-oriented design and operating system extensibility. This extensibility means that C2 Internet and other applications will define new abstract

operations unknown at the time the underlying DOS is built (although C2 applications will generally need less frequent changes of their discretionary authorizations than will general-purpose or text-processing systems). Because DOS extensibility is needed, we generalized our discretionary access controls to apply to any abstract operation that can potentially be defined.

2.1.1.2.6 Configuration Policy No practical security policy with only mandatory and discretionary rules can be complete. One still must guarantee that any system-specific policy parameters are properly initialized. Some examples follow:

- An MLS policy depends on proper initialization. A mandatory policy that bases security on classification levels and clearances assumes that those levels are responsibly assigned. Therefore, this policy must limit the means available for changing those levels, and for changing other security-relevant system information such as passwords and log-in IDs.
- The security policy will have configuration parameters other than classification levels that allow it to be modified to suit a particular installation. These parameters and rules for their modification will be fixed, system-specific features. An example of a configuration parameter is the circumventability of the policy itself: in an experimental mode of operation the policy is circumventable; in a tamperproof mode it is not.
- Since C2 Internet and SDOS are both distributed systems, new rules are needed to govern changes in the configuration of the distributed network. If new hosts are added, or the network topology is altered, the state of security enforcement must be re-initialized after each such change.

These special rules are defined and grouped together in a *configuration policy*.

2.1.1.3 Assurance

The previous section dealt with threats to secure functioning of a C2 application by listing the components of a policy that can block those threats. This section discusses the confidence one may have that a policy is correctly implemented.

Two documents issued by the National Computer Security Center, the *Trusted Computer System Evaluation Criteria* (TCSEC) [DoD Criteria 85], and the *Trusted Network Interpretation* (TNI) [NCSC TNI 87], are frequently mentioned in regard to evaluating how *trusted* a system is. Both of these documents define a linearly ordered set of criteria for "trust", each one more strict than the last. To evaluate "trust", both the TCSEC and TNEC combine an evaluation of a system's security policy with an evaluation of assurance that that policy is met. The word "trust", used in this sense, acts both to describe the policy, and to describe whether an implementation meets the policy. We will avoid

confusion by using the term "trust" as little as possible. Instead, we will characterize policies by the kinds of properties they enforce, and we will use the term *assurance* to refer to the confidence gained that a particular system is a correct implementation.

Every large system satisfies properties that have little assurance. For example, discretionary access control in SDOS may apply to abstract operations whose use does not need to be controlled in a given application. Various degrees of assurance may be needed for various operations. However, our policy itself makes no mention of degrees of assurance. An entity is either assured or it is not; its degree of assurance was the concern of the formalization phases of the project.

The assurance of SDOS as a whole will depend on the assurance of some subset of its total software. Elements of this subset will have an extra attribute that distinguishes them. This attribute will become a parameter of the security policy, just as the classification labels of data are a parameter of the Bell-LaPadula policy. The setting and modifying of these assurance parameters will be governed by rules of the configuration policy.

2.1.2 A Generic Policy for Multi-Level Security

In this section, we will first summarize the Bell-LaPadula policy, and note some problems that arise in its application. Next, we will discuss an access control policy for message-passing. The message passing policy will distinguish between assured, multi-level secure (MLS) entities and single-level entities with no assurance. We then state the McCullough policy which we will apply to every MLS entity. Finally, some additional properties of the mandatory policy are given.

The McCullough security policy, called *restrictiveness* here, forms the core of the mandatory policy. It places a constraint on possible information flow through multi-level secure entities, whether those entities are taken individually or collectively. Because restrictiveness is defined in terms of histories of message-passing operations, we will discuss message-passing first. At the same time, we develop special rules on message-passing which are similar to the Bell-LaPadula *-property. These special, additional rules are required because not every SDOS entity will be assured for every possible level of the system. The complete generic policy is the combination of restrictiveness and the additional rules governing message-passing.

2.1.2.1 Bell-LaPadula and Message-Passing

The most widely used policy for multi-level security has been the Bell-LaPadula policy. Bell-LaPadula defines security in terms of abstract entities, operations, and attributes of a system. The entities are *objects*, which are the passive containers of information, and *subjects*, which are the active agents that invoke operations on the objects. The operations are *read*, under which data flows from an object to a subject, and *write*, under which data flows from a subject to an object. Each subject and each object has

an attribute generically called a *level*; the possible levels are partially ordered, with an ordering called *dominates*. The rules of this policy are:

- A subject may not have read access to an object unless the subject's level dominates the object's in the partial-ordering (simple-security property).
- A subject may not have write access to an object unless the object's level dominates the subject's (*-property)

In combination, these properties support the higher-level notion of security that information should flow upward only in level: the first rule prevents *read-ups* (reading information above a subject's security level or at an incomparable level), and the second rule prevents *write-downs* (writing information below a subject's security level or at an incomparable level).

For several reasons, the Bell-LaPadula policy is less than ideal for application to SDOS. First, Bell-LaPadula makes a distinction between active and passive entities. However, in a distributed computing system, many system entities (such as network hosts themselves) are both active and passive. A policy that does not immediately distinguish between activity and passivity may be more appropriate for SDOS.

Second, the operation of reading becomes difficult to interpret for interactions between network hosts, since it is obviously *not* a primitive operation. The Bell-LaPadula policy permits one host to read information from another host which is at a lower level. But for loosely-coupled processors (ones which have no shared memory), the actual implementation of this read involves the sending of at least two messages: the first to request the read; the second to transmit the information requested. The request to read information from a lower level is a write-down, and therefore insecure under Bell-LaPadula rules; yet it should be possible to build workable and secure systems which allow this in some cases. The direction taken in this policy is to abandon read as a primitive operation, and to define the properties needed in order that requests to read lower-level information be secure.

In fact, the design of distributed systems, and in particular of distributed operating systems, has often seized upon one kind of communication, *message passing*, as the fundamental operation from which most other operations can be built. Therefore, to define security for distributed systems, it should be useful to begin with a definition of secure message passing. This will produce a security model in which the basic operations are not *read* and *write*, as they are in Bell-LaPadula, but instead the single operation of *send-message*.

The third and most important difficulty is that Bell-LaPadula is a policy on access control, whereas what is desired is a policy on controlling information flow. Of course, access controls are a vital mechanism for implementing controls on information flow, but the two are not the same. It is widely recognized that even Bell-LaPadula controls on access are not always sufficient to eliminate information flow via covert channels.

A policy which does control certain kinds of information flow, and which eliminates non-timing covert channels, is presented in section 2.1.2.3.

2.1.2.2 A Policy for Message-Passing Operations

Consider a system of communicating *entities*. These entities will have both active and passive characteristics, and therefore they can act as Bell-LaPadula subjects, objects, or both. Examples of entities are hosts on a network, processes, files, and human users. The basic activity of a system is message-passing among its entities. A secure system will restrict in some way the possible information flows which result.

A single, atomic, communication between entities will sometimes be referred to as a *send-message operation* or *event*.

The goal of the policy is to ensure that human users receive only information for which they are authorized. As in Bell-LaPadula, this is accomplished by extending the concept of authorization to all system entities, and ensuring that no entity contains information for which it not authorized. Every entity will be assigned an attribute called a *label*. A label will in general consist of a set of *levels*, each of which designates information sensitivity. Intuitively, a label is correctly assigned if the sensitivity of each piece of information contained in the entity is dominated by some level within the label. Part of the definition of a secure send-message operation will then depend on a comparison of the labels of sender and receiver entities.

In addition to its level, each entity is assigned a binary attribute: multi-level/single-level. A multi-level secure (MLS) entity is assured to distinguish various levels of information within itself, as long as the level of sensitivity of that data is within the MLS entity's label. In contrast, single-level entities are not assured to distinguish varying levels of information they contain; therefore, all data within a single-level entity must be considered to be classified at that entity's one and only level. Special assurance must be given for multi-level entities to prevent unauthorized mixing of information of information at different levels. A policy to prevent this will be discussed further in the next section. MLS entities maintain proper labeling of data by specifying for each send-message event the level of the message sent.

In terms of these abstract entities and their attributes, the basic rules governing send-message operations can be given as follows: Every system operation is an instance of a send-message event from entity A with label l_A to entity B with label l_B . The sender, A , chooses a level m for the message. The following rules must hold for this operation:

1. there is a level in l_A which equals m ;
2. there is a level in l_B which dominates m .

Rule 1 (given the special properties of MLS entities) ensures that each message is

labeled correctly. Note that if A is single-level, its label contains only one level, and that level must equal the level of every message sent by A .

Rule 2 ensures that if the level of a message is correct, then its receiver is authorized to know its contents. Note that if the sender and receiver are both single-level, then the restriction in rule 2 has the same form as the Bell-LaPadula *-property.

2.1.2.3 A Policy for MLS Entities

Whenever an MLS entity initiates a send-message event, there must be some justification for the security level it chooses for that event. This justification is expressed as a security policy for MLS entities. We would like to guarantee that a message's level dominates the sensitivity of all information gained when the message is received. This can be guaranteed if, within each MLS entity, the content of a message, and even its existence, is based only on information from equal or lower levels. Equivalently, we require that information does not flow within an MLS entity from messages received at level x to messages sent at level y unless y dominates x .

What does information flow mean? Although there may be many possible answers to this question, we will use a definition of information flow based on deducibility. Information flows from level x to level y if knowing the history of events directly seen at level y allows one to deduce something about the history of events directly seen at level x . Put another way, information does not flow from x to y if even full and complete knowledge of events at level y implies nothing about the events at x .

The demand that nothing be implied about events at level x is too strong. There are two basic reasons. First, knowledge of the history at y may automatically imply something about history at x , regardless of the design of the MLS entity in question. For example, if the sets of events at x and at y overlap, then knowledge of one automatically implies something about the other. This could easily happen if, say, both sets contained all public send-message events. Our definition of information flow should be weakened to eliminate these automatic and *a priori* deductions.

Second, suppose that a particular MLS entity is designed to upgrade messages, to receive a message at a low level and later send the same message content out at a higher level. Knowing the design of this entity and a history of events at the lower level, anyone can deduce that there must be events happening at the higher level. Yet we would like to consider this MLS entity secure. Our definition of information flow must again be weakened to cover just deducibility of facts about the history of messages received at higher levels, since it is these inputs which are the source of higher-level information flowing into the entity. Information does not flow from x to y if the history of events at y implies nothing about the history of messages received at x .

Treating information flow as deducibility in the above sense will still be too strong for our purposes. An event may be considered to include the time of its occurrence. However, requiring non-deducibility security in the presence of timing information makes

most MLS designs with shared processing resources insecure. The McCullough policy for multi-level security does not explicitly consider deducibility based on timing information, nor do we know of multi-level security models which do. Therefore we remove the time component of an event, and replace the set of events with the sequence of events linearly ordered by time. This retains deducibility based on the relative order of events in a history.

We now state the McCullough policy for MLS entities. To clarify our statement, some notation must be introduced.

Let E be a set (the send-message events) with distinguished subsets I (the input events received), and O (the output events sent). E^* , I^* , and O^* are the sets of finite sequences of events from E , I , and O respectively. Let T be a subset of E^* (the set of possible histories of send-message events of an MLS entity). If α and β are sequences, then $\alpha \wedge \beta$ is their concatenation. For any sequences, if $\alpha \wedge \beta$ is in T , then so is α .

Let L be a partially ordered set (the security levels). Each event e from E has a unique level $l \in L$. If α is a sequence in E^* , then $\alpha \uparrow l$ is the subsequence of α of all of its events with levels dominated by l . In general, if S is a set of events, $\alpha \uparrow S$ is the subsequence of α obtained by removing all events not in S .

Particular sequences of events will be written in angle brackets, e.g., $\langle e_1, e_2, e_3 \rangle$.

The policy for MLS entities applies only to those entities which are always ready to accept inputs. ("Accepting" an input may not also require that the input be acted on; in other words, it does not eliminate entities which may fail for some reason.) The entity described by T is **input-total** if for all α in T and all i in I , $\alpha \wedge \langle i \rangle$ is in T .

An input-total process is **restrictive** if for all levels l , all sequences α and γ in E^* , and all input sequences β and β' in I^* , whenever

$$\begin{aligned} \alpha \wedge \beta \wedge \gamma &\in T \text{ and} \\ \beta \uparrow l &= \beta' \uparrow l \end{aligned}$$

then there is a sequence γ' in E^* such that

$$\begin{aligned} \alpha \wedge \beta' \wedge \gamma' &\in T \text{ and} \\ \gamma \uparrow l &= \gamma' \uparrow l \text{ and} \\ \gamma' \uparrow I \uparrow \text{not} - l &= \langle \rangle. \end{aligned}$$

(The operator $\uparrow \text{not} - l$ removes from a sequence all events with levels dominated by l .) The above property can now be re-stated. Let the *visible* events be those with levels dominated by l . Given a possible history of send-messages for an entity, if a block of inputs to that history is altered without altering the visible inputs, then the history with the new inputs can be continued into a possible history containing the original sequence of events, and without making any new invisible inputs.

For an entity which is restrictive, a user or other entity which can see only the visible events is limited in what can be deduced about invisible inputs. Suppose that

the history $\alpha^{\wedge}\beta^{\wedge}\gamma$ is the actual behaviour of the entity. The user may conjecture that some behaviour $\alpha^{\wedge}\beta'^{\wedge}\gamma'$, produced by an altered sequence of inputs, did not happen. But this cannot be ruled out since it would have produced the same sequence of visible events. Therefore the user can make no deductions which eliminate such altered input histories. It may at first appear that only special inputs may be altered (those in β). However, β may be placed at any juncture in the history, and possible histories with any given sequence of invisible inputs may be constructed by repeated application of the restrictiveness property. It is this limit on deducibility which allows us to claim that a restrictive entity prevents information flows which downgrade.

The property of restrictiveness, first studied in [McCullough 87], is slightly stronger than required simply for non-deducibility. It is also a *hook-up* property. Two entities may be considered hooked together to form a new, larger entity if some input events of one entity are identified with some output events of the other, and vice-versa. A property is a *hook-up* property if it holds for two hooked-up entities when it holds for each entity alone. (For a more precise statement of "hook-up", see section 4.2.) Since restrictiveness is a hook-up property, if two restrictive processes communicate with one another, then together they form a process that is itself restrictive.

The property of restrictiveness is similar in some ways to the Goguen-Meseguer non-interference policy [Goguen and Meseguer 82]. Each is concerned with keeping a user at level l from deducing information about higher levels. We have chosen to use restrictiveness instead as our basic policy for several reasons.

- The Goguen-Meseguer policy is given in terms of internal system states. This means that the portion of the system state which contains information at level l must be identified. While this may always be possible, it means that the policy cannot be completely given until many parts of the system design have already been decided. Instead, we have identified information at levels higher than l by identifying its source: send-message events that are inputs and whose levels are not dominated by l .
- "Hook-up" is not well-defined in the Goguen-Meseguer model, and the histories of events possible in the model are somewhat more limited than in the McCullough model.
- The non-deducibility property implicit in the Goguen-Meseguer model is not a hook-up property, while restrictiveness is.

Our policy, then, is that each MLS entity be restrictive at every level l . Because restrictiveness is a hook-up property, the collection of all MLS entities in SDOS will also be restrictive. This fact shows that SDOS will control information flow through the assured part of the system. Information flow through the part of the system without assurance is controlled by the rules of the message-passing policy.

For a diagram showing possible message passing routes in a simple system, see Figure 2.1.

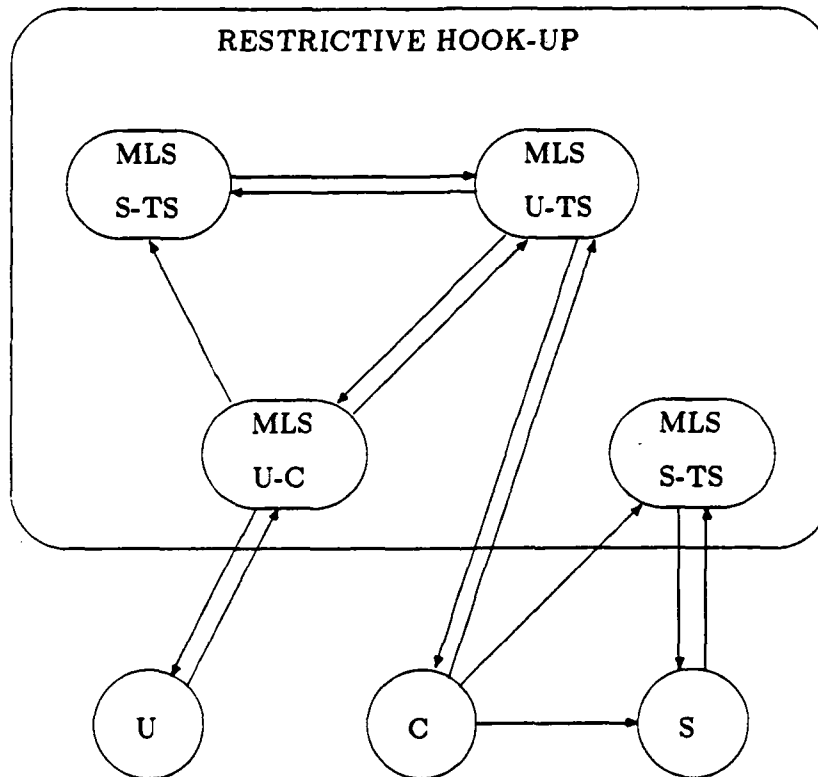


Figure 2.1: Some possible message passing in a simple system. Levels (U, C, S, and TS) and assurance attributes are shown. Each of the four MLS entities is restrictive, as is their hook-up.

2.1.2.4 Additional Rules and Comments for Mandatory Policy

2.1.2.4.1 Authentication The message passing policy requires that an entity supply a proper level for each message it sends. If the sender is MLS, we have also required a policy which constrains the choice of level for each send-message. However, nothing has been said about the means of enforcing the message passing policy when a message sender is single-level, or the means of enforcing that an MLS sender use only levels within its label. The choice of mechanism is an implementation issue, and can be decided differently in different cases:

- a sender may be assured to enforce the policy;
- a receiver may be assured to enforce the policy;
- hardware domains or other partitions may be used to prevent unauthorized message passing;
- messages may be marked with the level of a port or other channel through which they pass;
- a receiver may be assured to use authentication of a sender to determine a proper level for the sender's messages.

2.1.2.4.2 Integrity The outline of secure message passing operations given above does not explicitly mention integrity. We implicitly include *integrity*, in the Biba sense [Biba 77], into the definition of levels. This expands a level into a pair with the first component indicating sensitivity and the second component indicating integrity. This change will not complicate the policy rules given above in any way.

2.1.2.4.3 Labels and Sets of Levels Every MLS entity has a label consisting of a set of levels. In general, any set is permissible under this policy. However, there may be design features which make it desirable to rule out some possible sets of levels:

- It is possible to construct an entity which is restrictive, but which cannot meet the message passing policy unless every pair of levels in its label has a join (the least level dominating both) that is also in the label. It follows from this rule that there must be a level in the label of such an entity called the *join level*, which dominates every level in the label. Some SDOS entities require this.
- A design in which messages are broadcast to entities at many levels may require that the *meet level*, i.e., the greatest level dominated by all the others, exist in the label of the broadcast sender.
- A design which allows a flexibly large number of levels must allow an astronomical number of labels. Extra rules may be imposed on the labels just to reduce

their number. For example, every label might be required to be a range of levels, including just those levels greater than some minimum and less than some maximum.

2.1.2.4.4 Creation of New Entities SDOS operations that result in the creation of new entities are not, in a practical sense, constructed from more fundamental or primitive send-message operations. An entity that does not yet exist should not be able to receive messages. However, rules that govern the secure use of "create" operations can be obtained by fitting them into the generic send-message policy in the following way.

Let it be supposed that all possible entities exist already, if only potentially. Then the actual existence of an entity can be treated merely as a binary attribute: exists/does not exist. The creation of a new entity is a send-message operation received by a potentially existing entity, requesting that it actually become existing. From the rules given above, it is clear that a single-level entity may only create entities at its level and at greater levels.

A new process P may be created to run the code C only if the level of P dominates both the level of C and the level of its creator. In this way we avoid the need to give special policy rules for the "create" and "execute" operations.

2.1.2.4.5 Requests to Read As noted before, a "read" operation will often not be considered a fundamental operation in a distributed system. A "read" may be composed of a pair of messages: a request to read, followed possibly by a response. If reading data from a lower level is to be considered secure, as it is in Bell-LaPadula, it is because the request to read send-message causes no harm. However, the fact that a request to read has been sent is in general as sensitive as the reader itself, and in general it will downgrade information. There are two approaches to this problem.

If the reader can show that the request to read is overclassified, it can be sent at a lower level. This may be possible if the reader is MLS (in which case the read operation is not really a read-down). It will also be possible if specific reasons justify the downgrade. For example, a human user may intervene, and decide that the request to read need not be classified at the reader's level. The SDOS policy permits this kind of human intervention in some cases. (see section 2.1.3.4).

If assurance can be given that the receiver of the request to read will handle it securely, it can be sent at the reader's level. If the receiver is MLS at the reader's level, all is well. Otherwise, additional policy rules may be imposed on the receiver. For example, the receiver may be assured never to send out a copy of the request to read, and to expunge all traces of it immediately after processing. Formalizing such a policy for requests to read may be difficult. We have not pursued it further.

2.1.2.4.6 Secure Extensibility New software may be added to SDOS after it becomes a fielded system. Some of this software may have assurance that it meets a security policy. If a new component is restrictive, if its assurance is as great as the assurance of the other restrictive SDOS components, if the new component is not required to enforce any other aspect of the policy (such as the message passing rules), and if it labels messages in a way recognized by the rest of the system, then it may be included in SDOS as an MLS entity without destroying the overall security of the system. Because restrictiveness is a hook-up property, secure extensibility is possible. This fact can be exploited in the design of SDOS.

2.1.3 Instantiated Policy for SDOS

To arrive at a system-specific policy for SDOS, we performed the following instantiation of the generic policy. We

- identified SDOS entities that are instances of the entities of the generic policy;
- identified SDOS operations that are instances of the generic send-message operation between pairs of SDOS entities;
- included any additional system-specific restrictions on the use of the send-message operation that are necessary to define security. These additional restrictions will be both discretionary access controls and fixed rules governing the system configuration.

In the design of SDOS, there can be any number of layers of design, with higher layers showing greater abstraction and lower layers showing greater detail. The generic policy can be applied at many different layers. Part of the process of instantiation involves choosing the design layer at which the atomic components of the design are the entities of the policy. At each higher layer, the generic policy will apply. At lower layers, other methods of security analysis and various assumptions about the functions of the hardware will be relied on for security.

Two layers of SDOS entities will be discussed in the policy: the user interface, and the system layer. At the more abstract user interface layer, the SDOS is seen as an object-oriented system, defining sets of *abstract objects* of various types, and defining for each type a set of *abstract operations* that can be invoked on objects of the type. All objects are entities of the SDOS policy, and represent abstract resources of the system. Examples of objects are: files, directories, SDOS object manager processes, and so on. The policy for this higher, more abstract layer is independent of the hardware architecture supporting SDOS. This is also the layer at which discretionary access controls are defined.

At the second, more detailed system layer, the SDOS architecture and mechanisms become visible. SDOS is composed of interacting entities which need not be visible at the user interface. Applying the generic policy at this layer will produce security rules which govern the interactions of the architectural entities.

In section 2.1.3.1 we enumerate the set of entities that make up SDOS, differentiating between those entities that are visible at the user interface level and those that are not. The major kinds of system component are: user, process, object, message switch (for routing messages between entities), security database, object database, and the physical host and network entities. In section 2.1.3.2 we present the mandatory policy in this system, and describe how invocations of abstract operations can be achieved securely. In section 2.1.3.3 we present the discretionary access control policy. This section states the policy enforced by the various managers of types, and a policy for controlling the spread of discretionary access privileges. Section 2.1.3.4 contains the configuration policy. It consists of two parts: a set of policy parameters, used to control how SDOS is customized for a particular environment, and a set of rules which govern how these parameters can be changed under changes in the network connectivity.

2.1.3.1 SDOS Entities

Based on our understanding of secure systems and of distributed systems, we expect the list below to form a complete set of the kinds of entities in SDOS. Though this list is intended for an object-oriented system, we believe the roles and functions that the entities represent are common to any secure distributed operating system.

- *Users*: human users of the system, who issue commands and are given services in return. These are MLS entities, assured to label information only in secure ways, but the assurance that they meet an MLS policy is outside the scope of formal methods used on this project. Users are typically associated with *client identifiers* within the system. There are several users which have specific privileges that are fully described in section 2.1.3.4 on the configuration policy:
 - *System Manager*: responsible for managing the operational SDOS, including the setting of mandatory labels, and the creation and management of new client identities.
 - *System Auditor*: responsible for inspecting the audit record for suspicious events, noting possible penetration attempts, and blocking confirmed attempts.
 - *System Certifier*: responsible for approving any additions to SDOS of MLS entities other than users. New MLS software must be assured to meet the SDOS policy at least at the level of assurance of the original SDOS verification.
 - *System Controller*: responsible for controlling the creation and modification of new object types.
 - *System Architect*: responsible for pre-configuring each SDOS system. The System Architect will choose to have certain policy parameters permanently enabled and others permanently disabled. The System Architect performs his/her function before a system is fielded.

- *User Trusted Interface Process (TIP)*: an MLS entity associated with each user's terminal. In addition to being restrictive, this process is assured to correctly interpret security-relevant user commands, in particular those commands performing login, logout, and change of discretionary access rights (see section 2.1.3.3 for a description of the discretionary policy). A TIP is responsible for authenticating each user who logs in, and associating levels with the correctly authenticated user's commands.
- *Other User Processes*: single-level entities associated with a user, but not assured to meet any security properties. Examples are processes spawned by a user via a TIP acting on his behalf.
- *SDOS Objects*: instances of abstract data types. These objects are resources that can be addressed by users. They comprise a major portion of SDOS applications. They include the user processes mentioned above, user data, devices, etc. Some SDOS objects may be MLS.
- *Directories*: storage containers for symbolic, user-level names of objects. These names are aliases for global names given to entities by the system. Some directories may be MLS.

The following are SDOS entities not necessarily visible at the user interface, although their functions will still be common to any secure distributed operating system.

- *Hosts*: a collection of local resources, hardware and software, needed to support SDOS at one node of its underlying network. A host may be single-level or MLS. The security label of a host includes the security label of every entity on that host.
- *Network*: interconnections between hosts. A variety of hardware and software connections is possible. These interconnections may be either single-level or MLS entities.
- *Message Switch*: controls routing of messages, in particular the messages that result from abstract operation invocations. There is one message switch per host. It is the primary agent responsible for enforcing the message passing policy. The message switch has the same security label as the host on which it resides, and it is an MLS entity if and only if that host is MLS.
- *Security Database*: records the label of each system entity. If an entity is assured to satisfy some special policy, any necessary information about that policy (MLS, discretionary, etc.) is also recorded. The database has the same security label as the host on which it resides, and it is an MLS entity if and only if that host is MLS.
- *Object Database*: stores information about SDOS objects according to their type. The object database has the same security label as the host on which it resides, and it is an MLS entity if and only if that host is MLS.

- *Object Managers*: implement the abstract operations possible for the various SDOS types. Each type has at least one object manager which defines the operations for that type by manipulating the data in the object database. An object manager may be assured to enforce the discretionary policy given in a later section. An object manager may also be an assured MLS entity.
- *System Audit Record*: the collection of audit data. The Audit Record is an instance of a particular type of SDOS object. It is intended for use solely by the System Auditor.

2.1.3.2 Mandatory Policy

SDOS is an object-oriented system. Users of the system conceive of its basic function as implementing the invocation of abstract operations on abstract objects. These invocations are interactions among the higher, more abstract layer of SDOS entities.

Each invocation, however, is implemented as a chain of more primitive interactions among the SDOS entities at the more detailed, architectural layer of design. It is these interactions which we identify as the message passing events. SDOS will be mandatorily secure if each entity at this architectural design layer is described in terms of these events, meets the message passing policy, and satisfies restrictiveness if it is an MLS entity. The set of message passing events must be complete, in the sense that all interactions between entities at this design layer must be represented.

Which entities carry the responsibility for enforcing the policy? This is strictly a design question, rather than a policy one, but the answer will determine whether enforcement will be decentralized, involving the assurance of many different entities, or centralized, involving the assurance of a localized "reference monitor". In the remainder of this section, we describe a sequence of message passing events that can typically occur when a user invokes an abstract operation on an object and receives a result from that invocation. The sequence of events is the more general case, in which the user and the object of the invocation are on different hosts. While this example is framed as an object-oriented interaction between entities, the mandatory policy could be implemented in other ways (for example, using a remote procedure call model of entity interactions).

The following sequence of events is organized into the three phases which together support the completion of an abstract operation on an object: invocation, execution, and return.

1. Abstract Operation Invocation

- A user initiates an operation invocation by sending a message to his/her TIP. This TIP has previously invoked an authentication operation for this user, and so it knows the logged-in level of the user.
- The TIP sends a message, whose contents encode the invocation, to the message switch on the local host. The message is sent at the level of the logged-in

user. (Note that an invocation could instead begin with a process other than the TIP.)

- The message switch checks whether the level of the message is authorized. For this check, it must find out the security label and MLS attribute of the TIP or of the invoking process. It does this by an exchange of send-message operations with the security database, where the levels and attributes are stored.
- The message switch locates the destination of the message using a *locate* facility within the message switch. This facility either has cached the location of the object on which the operation was invoked, or searches for the object by communicating with other message switches. This location activity requires the host to exchange messages with others hosts. A remote message switch responds to a locate request by searching its security database for the named object and returning an acknowledgement accordingly.
- When the message switch determines the location of the object, it forwards the message to the message switch at that location, if the remote message switch is authorized at a level which dominates the message level. Again the authorization is determined by an exchange of messages with the local security database.
- The message switch on the remote host determines whether an appropriate object manager is authorized at a level which dominates the message level. It does this by an exchange of messages with the remote security database.
- If no manager has an authorized level at which to receive the message and no appropriate manager can be started, the invocation fails. Otherwise, the message switch forwards the message to the manager.

2. Abstract Operation Execution

- The object manager requests of the object database an exchange of information concerning the object of the invocation. The object database honors this request or not, depending on the level of the request and the label of the object.
- The object database exchanges messages with the security database to examine the label of the object.
- The object database replies to the object manager.

3. Abstract Operation Return of Results

- The manager replies to the invoking user or process via the message switches and hosts. This reply will be a sequence of send-message events at a level determined by the manager. The level will of course be within the manager's label, and it will also bear some relation to the level of the object and the level of the original invocation.

From the above example, one can see that each TIP, each message switch, each security database, and each object database residing on a host with users at multiple levels will need to be restrictive. Also, to avoid the need for creating equivalent object managers at multiple levels, one may prefer MLS managers to single-level ones for the sake of efficiency. Thus, the enforcement of the mandatory policy will be a joint responsibility of many separate design components.

In the above account, several simplifying assumptions were made about the design. For example, it was assumed that a object manager is local to the same host as every object it interacts with. This simplified the account, but is not necessary for the design. Entities on different hosts may always communicate if the proper intermediate entities exist and if the communication is an instance of the generic policy.

2.1.3.3 Discretionary Policy

Discretionary policy in SDOS controls a different set of operations from the mandatory policy. The mandatory policy defines security in terms of send-message events; the discretionary policy controls the use of abstract operations invoked on objects. As seen in the previous section, abstract operations can be built from repeated application of send-message. Since SDOS is based on an object-oriented paradigm, these higher-level operations are exactly those operations defined for the objects known to the system. As in any object-oriented system, the possible operations depend on the type of each object. Because it is always possible to define new types by building new object managers to implement them, the number and kind of operations to which discretionary policy is applied is not known in advance and is not fixed.

2.1.3.3.1 Operations on Objects Any SDOS object able to invoke operations on other objects is called a *client*. Each client process has a *client identity* established when the client is created. The client identity is used to authorize the client's access to objects. Any user process and object manager can be a client. Authorization to invoke an abstract operation on a particular object is granted to a client based on the client's identity, and the authorization is called a *discretionary access right* for that operation. The discretionary policy is: an abstract operation invoked on an object by a client will be performed only if the client has the corresponding discretionary access right.

Although the above policy applies to all invocations of abstract operations on objects, the assurance that this policy is correctly implemented will vary among abstract operations. Any user of the system may define new abstract operations, and in general there will be no assurance that an arbitrary object manager meets the discretionary policy. Therefore, the types known to the system are divided into two classes: *protected types* and *unprotected types*. The protection attribute of an object type is designated by the System Certifier (see section 2.1.3.4.4 on Policy Parameters). Assurance of the discretionary policy will be given only for the abstract operations defined for protected types. The types which are protected may increase over time if there is sufficient assurance

that their object managers meet the discretionary policy.

Operations on protected types are divided into two groups. *Direct operations* are abstract operations that can only be invoked by users, each represented to SDOS by a terminal interface process (TIP). The TIP must be assured to interpret user commands correctly. As a result, direct operations may only be invoked manually by people. This policy of manual invocation is useful for controlling the initiation of operations on objects of protected types that are of critical importance (e.g., some weapons systems). Operations which do not require manual invocation are called *nondirect operations*. The set of direct operations for a protected type is designated by the System Certifier (see section 2.1.3.4.4 on Policy Parameters).

In general, discretionary security policies do not strictly control information flow. Suppose *A* is an object of a protected type. If client *C* has no discretionary access rights to *A*, then *C* cannot directly gain information from *A*. However, indirect information flow is possible through an intermediate client *B*, which does have discretionary access rights to *A*, and which can relay the information back to *C*. (*B* could be a Trojan Horse constructed by *C*.)

Note also that this discretionary policy does not depend on the semantics of the abstract operations whose use is being controlled. It is consistent with the policy to deny a client abstract operation *X* but permit abstract operation *Y* for some object, even though *X* and *Y* have identical effects.

2.1.3.3.2 Modifying Access Rights The abstract operation of modifying discretionary access rights is of special importance. The goal of the discretionary policy is to reflect the intent of user authorizations via the set of discretionary access rights. If arbitrary (possibly Trojan Horse) processes are permitted to modify discretionary access rights, all rights could easily be granted to all possible clients. Of course, this would not reflect every user's intentions. Therefore, propagation of the discretionary access right that authorizes a client to modify discretionary access rights must be limited.

The right to modify access rights to a protected object is restricted to users (specifically, terminal interface processes) and managers of protected objects. These entities are collectively called *primary clients*. The policy states no rules for the use of operations on objects of unprotected types.

To prevent frequent conflicts between the intentions of different users over a particular assignment of discretionary access rights, the right to modify the discretionary access rights of an object is limited to a designated set of primary clients called the *controlling group* of that object. The controlling group for a protected object is designated by the System Controller (see section 2.1.3.4.4 on Policy Parameters). The controlling group of a newly created unprotected object is the client responsible for its creation. Membership of the controlling group of any object can be modified only by the System Controller.

2.1.3.4 Configuration Policy

There are system-specific rules of the security policy that are neither mandatory (based on security levels) nor discretionary (based on revocable authorizations that depend on client identities). These rules form the third part of the instantiated security policy, called the *configuration policy*. We adopted this name because many of the rules are related to the configuration of SDOS at a particular time and for a particular installation. These rules fall into two categories: network policy and policy parameters.

Network Policy

SDOS is assumed to be supported by a set of hosts, each having local memory but connected via a communication medium, called the *network*. Since SDOS will be a distributed operating system designed to make many of the details of hosts and the network invisible to users, most of the security policy is independent of these entities. However, we discuss network security here for two reasons:

1. The physical connections provided by the network may change over time. The policy is the set of rules describing how these changes may occur in a secure manner.
2. Some specific properties will be expected of the network that supports SDOS. These properties will be needed in implementing the high-level policy we have described so far.

Policy Parameters Certain aspects of how the mandatory and discretionary rules are applied within a particular system can be controlled by specific individuals, such as the System Architect and System Manager. The System Architect, for example, can specify whether the system is *tamperproof* or *experimental*. Making these choices allows the security policy rules to be tailored to fit the intended use of SDOS at a particular site. The features of the policy that can be controlled are called the *policy parameters* of the instantiated security policy. The set of values that these parameters can take defines a *family of security policies* that SDOS can implement.

In order to understand the configuration policy it is necessary to understand how SDOS is installed and configured. In the next section we introduce terms that relate to the configuration of SDOS and that allow us to describe more precisely the network policy and policy parameters. Section 2.1.3.4.2 describes the process of installation of SDOS. Sections 2.1.3.4.3 and 2.1.3.4.4 describe the network policy and policy parameters that make up the configuration policy.

2.1.3.4.1 Terms

Communication Path: an interconnection between hosts that allows information flow. A communication path refers primarily to a physical connection. For example, even

if all data sent over a physical connection is encrypted and cannot be decrypted at the destination, the two hosts involved are still considered to be linked by a communication path.

Network Configuration: the current set of communication paths linking hosts.

Network Reconfiguration: a change in the network configuration due to host failures, communication path failures, or intentional addition or deletion of hosts and communication paths from the network.

Security Preconfiguration: the set of policy parameters selected by the System Architect before a fielded version of SDOS becomes operational.

Security Configuration: a complete set of current policy parameters for SDOS including the security preconfiguration; selected by the System Architect, Manager, Auditor, and Certifier.

Preconfigured System: a set of hosts that can potentially be linked by communication paths and that are assured to have identical security preconfigurations.

Connected Subsystem: a subset of hosts of a preconfigured system that are currently linked by communication paths.

Non-system Host: with respect to a particular preconfigured system, a host that has a different security preconfiguration than the system or a host that will not be linked to any other host in the system.

Closed System: a system for which all communication paths to non-system hosts are known to the system and all such paths are selectively controllable by the system.

Open System: a system that is not closed. A system may be open, for instance, if its network is a LAN connected to arbitrary non-system hosts. This prevents all communication paths from being known to the system. A system may also be open, even if all paths are known, if wiretapping of a known communication path is possible, i.e., paths are not selectively controllable.

2.1.3.4.2 System Installation Each site where SDOS becomes operational will have individual security requirements. The System Architect is responsible for generating a version of SDOS appropriate to these local requirements. The System Architect generates and distributes a customized SDOS system along with any special hardware and/or firmware. This fielded implementation of SDOS incorporates values of the policy parameters which are the security preconfiguration of the system. These values are the choices for some of the policy parameters listed in this section. Once the System Architect has selected the security preconfiguration for a system and the system is fielded, the preconfiguration can only be altered by replacing one system with another. The fielded implementation is distributed and loaded into SDOS machines in the field. The system is then installed (booted) and configured by the System Manager, Auditor, and

Certifier. The process of security configuration involves selecting values for all policy parameters other than those already chosen in the security preconfiguration.

2.1.3.4.3 Network Policy The first part of the network policy is concerned with properties of the network. The network is an SDOS entity that transports messages from one host to another. Properties expected of this network include:

- *Data Confidentiality:* If a message is delivered, it must be delivered to the correct receiver.
- *Data Origin Authentication:* It must be possible for the receiver of a message to determine the message's sender reliably.
- *Data Integrity:* Messages must be delivered uncorrupted.

These properties, and methods for their implementation, are discussed in greater detail in section 3.4.

The second part of the network policy requires that there be a single security configuration for every connected subsystem. Several policy rules are needed to satisfy this requirement:

- If SDOS is installed on host *A*, and host *A* initializes the SDOS on host *B*, then *A* sets the policy parameters on host *B* that are equivalent to its own policy parameters.
- If a system is network-reconfigured so that a connected subsystem *S* is partitioned into two or more intraconnected subsystems, then each connected subsystem must have the security configuration of *S*.
- If a system is operating independently in two separate, connected subsystems, and those subsystems are then connected into a single connected subsystem, the two subsystems must reconcile differences in their security configurations. If a given policy parameter has different values in each separate subsystem, the value that takes precedence is the value that is more *strict*. The choice of values that are more strict is indicated in the next section by stating when one policy parameter value takes precedence over another. This rule prevents the bypassing of strict security on one host by connecting it to another on which a less strict policy parameter value has been chosen.
- Since choices made by the System Architect are not alterable, hosts within the same system must agree on these parameters. Therefore, within a system, hosts that are assured to meet the SDOS policy, but with a different security preconfiguration, are treated as single-level entities with no assurance.

In a closed system, it is possible for each host to assume that data received from the network is labeled correctly, and that data sent out over any communication path can be received only by cleared personnel. In such a system, data sent over communication paths need not be encrypted. For open networks, it is necessary to use a system of encryption of messages sent over the network to prevent both the unauthorized reception of legitimate classified data, and the malicious generation of fake data.

The most common example of non-system hosts are hosts that are not assured to enforce any security policy. Such a host cannot be a member of any system, since nothing about their security preconfiguration can be assured. Such a non-system host can be integrated in an SDOS, however, in one of two ways.

1. When the only communication path from the host to the SDOS system is via an access machine, and the access machine is assured to enforce the SDOS policy with the system's security configuration (i.e., is a member of the system). The access machine treats its host as a single, indivisible, single-level entity. In this case the combination of host-plus-access-machine is considered a single host of the system.
2. When all hosts recognize the non-system host as a single-level entity.

In both of these cases, the interaction of the non-system host with the rest of SDOS is restricted. Requests to read sent from higher-level hosts to the nonsystem host are prohibited (except when the read-down enabled policy parameter is set – see the next section) because the host cannot be assured to enforce an MLS policy. Since the non-system host is not assured to reliably authenticate its own clients, it cannot be trusted to reliably participate in the authorization of abstract operations requested by its clients on remote hosts. A non-system host can be given a limited capacity to use remote SDOS resources if it interacts with SDOS via an access machine. All clients on the non-system host can be associated with a single client identifier (reliably supported by the access machine), and can invoke abstract operations remotely if the host possesses the discretionary access rights for those operations. This is feasible for single-user workstations.

2.1.3.4.4 Policy Parameters Policy parameters are selected, at three different times, depending on the type of parameter:

1. As part of security preconfiguration, prior to the fielding of the system. These parameters are chosen exclusively by the System Architect.
2. At security configuration time.
3. During operation, e.g., when new object types are being defined.

The various parameters and rules listed in this section are organized into groups. For those parameters which are not part of the System Architect's security preconfiguration, we have indicated which values must take precedence when two network subsystems are connected together. A concise listing of all the parameters is provided in table 2.1.

Parameter Groups	Policy Parameter	Set By	Time of Setting
Policy Circumventability	Experimental/ Tamperproof	System Architect	Preconfiguration
Mandatory Security	Levels modifiable/ nonmodifiable	System Architect	Preconfiguration
	Security Label	System Manager	Configuration
Software Extensibility	Manual assured extensible/ Not assured extensible	System Architect	Preconfiguration
	Multi-level secure (MLS)/ single-level secure	System Certifier	Configuration
Policy Exceptions	read-downs enabled/ Read-downs disabled	System Manager	Configuration
	Manager create-abort enabled/ Manager create-abort disabled	System Manager	Configuration
Object Types	Protected/ Unprotected	System Certifier	During operation
	Controlling group membership	System Controller	During operation
	Direct use/ Nondirect use	System Controller	During operation
Auditing	Auditing on/ Auditing setable/ Auditing off	System Architect	Preconfiguration
	Auditing disabled/ Auditing enabled	System Manager	Configuration
Directories	Directories restricted/ Directories unrestricted	System Manager	Configuration

Table 2.1: Policy Parameters

Policy Circumventability

In order to effectively develop and test a secure system it is necessary to circumvent the security policy intended to be implemented by that system. The following policy parameter determines whether the SDOS security is circumventable:

- *Experimental/Tamperproof*: set by System Architect at system preconfiguration time. If tamperproof mode is set, then the implementation for an entity assured to enforce some property of the mandatory, discretionary, or configuration policy cannot be modified. The only exception to this rule is that the System Certifier may modify or add assured object managers to the system. In experimental mode these restrictions do not hold, and the implementation of assured entities may be changed. Experimental mode is intended for systems under development, while tamperproof mode is for fielded systems that need assured security.

Mandatory Security

Part of the security preconfiguration of a system is a choice of interpretations for the security levels used in mandatory security. For example, if SDOS supports N security categories, then the use of these categories must be determined and enforced system-wide. This interpretation is not under the control of SDOS, and so is not really a security property enforceable by the computer system. However, when two SDOS sub-groups of the same preconfiguration are joined, the representation of security levels (for example, the meaning of the security categories described above) in the two sub-groups must be the same. The original interpretation of the levels for a fielded system is made by the System Architect.

The following policy parameters pertain to the flexibility of the system of levels used in mandatory security:

- *Levels modifiable/Levels nonmodifiable*: set by System Architect at system preconfiguration time. If the levels nonmodifiable parameter is chosen, levels for newly created entities must be selected, but levels for existing entities may not be changed. If the levels modifiable parameter is chosen, the System Manager may explicitly change the labels of entities. Levels nonmodifiable is more strict, and therefore takes precedence.
- *Entity labels*: set by the System Manager. These parameters are the labels for every entity in the system. Entity labels are stored in the security database. If the label of an entity is replicated, the more recent value of the label is more strict, and takes precedence.

Assured Software Extensibility

The following policy parameters pertain to the extensibility of the assured system software.

- *Manual assured extensible/Not assured extensible*: set by System Architect at system preconfiguration time. If the not assured extensible parameter is chosen, the set of entities containing the executable code for assured object managers cannot be changed. These managers are ones assured to meet the mandatory policy for MLS entities and/or the discretionary policy. The MLS assurance attributes in the security database will be unmodifiable, and the set of protected types cannot be expanded. On the other hand, if the manual assured extensible parameter is chosen, the set of assured object managers may be extended by the System Certifier. New entities containing the assured executable code of the managers can be created.
- *Multi-level secure (MLS)/single-level secure*: set by the System Certifier. This attribute of entities may be modified, but only by the System Certifier. Some entities will be marked as MLS in the initial system configuration. If information about the MLS/single-level secure attribute of an entity is replicated, the more recent value of the attribute takes precedence.

The following policy rules pertain to the extensibility of the assured system software.

- MLS process entities, including MLS object managers, may only be created from executable code marked as MLS.
- Any MLS object manager will also be assured to meet the discretionary policy. The converse need not be true.

Exceptions to Mandatory Policy

The following policy parameters define limited exceptions to the mandatory security policy:

- *Read-downs enabled/Read-downs disabled*: set by System Manager at system configuration time. When disabled, every send-message event that requests to read information from an entity at a lower level must be secure as defined in the mandatory policy. When enabled, users may override the strict message passing policy to downgrade a request to read. Each such request must be explicitly approved by the user. This requirement for user approval limits the rate at which information could possibly be compromised. Enabling this parameter makes single-level hosts with no assurance more useable: a user on one host may approve read-down requests to be sent to entities on a single level host. The disabled parameter is more strict, and takes precedence.
- *Manager create-abort enabled/Manager create-abort disabled*: set by System Manager at system configuration time. When disabled, users may only create and abort processes at levels which dominate their own. When enabled, users may override the strict message passing policy to create/abort object managers at a lower security level. Explicit approval by the user of each create/abort is required

in order to limit the rate at which information could possibly be compromised. The disabled parameter is more strict, and takes precedence.

Object Types

The following policy parameters pertain to the definition of new object types:

- *Protected/unprotected*: set by the System Certifier at the time a new object type is created, or during ordinary operation. When unprotected, object managers for the type are not assured to enforce the discretionary access control correctly. The unprotected parameter is more strict, and takes precedence.
- *Controlling group membership*: set by System Controller (or automatically on his behalf) at the time a new object is created, or during ordinary operation. This list determines the clients that may change the discretionary access rights of the object. The intersection of controlling groups is more strict, and takes precedence.
- *Direct use/Nondirect use*: set by System Controller at the time a new object type is created, or during ordinary operation. This parameter controls the direct use attribute of each operation defined by an object type. Direct use requires that the operation only be invoked by a user's terminal interface process (TIP). The direct use value is more strict, and takes precedence.

Auditing

The SDOS system is able to perform auditing. The following policy parameters pertain to auditing:

- *Auditing off/Auditing settable/Auditing on*: set by the System Architect at system preconfiguration time. If auditing on or auditing off are chosen, no other user may alter the choice. If auditing settable is chosen, the System Manager is able to select between the following two parameters.
- *Auditing disabled/Auditing enabled*: set by System Manager at system configuration time. No other user may alter this parameter. Auditing settable, chosen in combination with auditing enabled, turns on auditing in SDOS, and is equivalent to a preconfiguration of auditing on. Auditing settable, chosen in combination with auditing disabled, turns off auditing in SDOS, and is equivalent to a preconfiguration of auditing off. Auditing enabled is more strict, and takes precedence over, auditing disabled.

The following policy rules pertain to auditing:

- The audit record is visible only to the System Auditor.
- The audit record may only be modified by an authorized audit database manager.

Directories

Directories contain user-level symbolic names for abstract objects. The following policy parameters pertain to directories:

- *Directories restricted/Directories unrestricted*: set by System Manager at system configuration time. If the restricted parameter is set, each directory is restricted to containing names of objects whose level is exactly the directory's own level. This rule need not hold if the unrestricted parameter is set. The unrestricted parameter is more strict, and takes precedence.

2.2 Assigning Values to the Security Policy Parameters

The security provided by SDOS may be tailored to the needs of a particular site through the choice of values for the security policy parameters. A particular choice of values is called the *security configuration*. A set of these parameters has already been defined in the SDOS Security Policy. (See section 2.1 for details). It includes the security label associated with each SDOS entity, parameters which may modify the mandatory policy based on those labels, as well as other parameters which constrain the role of some of the system security administrators. Some constraints on the values of these parameters were stated as rules of the Configuration Policy. In this chapter, we will consider in greater detail how their values might be assigned in practice.

Sections 2.2.1 and 2.2.3 consider general questions about choosing a security configuration. Section 2.2.2 deals with the special important case of the choice of security labels for SDOS entities.

2.2.1 Life-Cycle Phases

Some aspects of the security policy, such as the decisions made by the System Architect and the choices for policy parameters made by system officers, depend on the phase of the SDOS software life-cycle. We will now take an overview of that life-cycle to see how policy parameters get set at each phase, and to list certain procedural rules that must be observed for SDOS security.

In some cases, the procedural security rules are addressed by DoD regulations or regulations of the individual services. It is not our aim to reproduce those rules, but rather to point out what kinds of security rules must exist.

2.2.1.1 Development

All security in SDOS environments will obviously depend on the correct implementation of security-relevant parts of the message-switch and of assured managers. This implementation should therefore be carried out in an environment in which only authorized

personnel can code or make changes to the system. This restriction may be enforced both by software and procedural controls, but ultimately there must be some physical controls, e.g., secured areas, that prevent unauthorized access to the code under development. Procedural rules must govern not only who may modify the software of assured parts of the system, but also how modifications are distributed to SDOS sites.

An SDOS system itself may act as the site for development of new assured managers. This development may be speeded by setting the "policy circumventability" parameter to "experimental". Assured managers developed in such an experimental environment should not then be exported to other SDOS sites running in "tamperproof" mode, unless the development site is restricted to personnel authorized to modify the assured managers.

2.2.1.2 Installation

The System Architect makes choices for certain of the policy parameters, as indicated in Table 2.1. This "security preconfiguration" is unalterable without repeating the installation procedure.

Particular users must be chosen for the administrative roles of Manager, Auditor, Certifier, and Controller. For fielded systems with many users, these will probably be different people. For smaller systems, some roles may be filled by the same user. For example, the Certifier and Controller may be the same person. Combining roles in this way spreads authority among fewer users and therefore increases the amount of damage to security that a single person can cause. Because the Auditor acts as a check on suspicious activity by every other user, designating the same user to fill both the Auditor and other roles should be prohibited.

The system administrators are then responsible for setting the remaining parameters of the security configuration to appropriate values. Some set of these values may already exist on stable storage from a previous installation of the system.

2.2.1.2.1 Installation of Multi-level Devices Some of the peripheral devices connected to SDOS will be used at more than one security level. This poses the risk that such a device contains Trojan Horse hardware or software, and could store information at one level to be released at another level. A user's display terminal is one example. If the terminal can be used for login at multiple levels, there is a possibility that the terminal can store information from a higher level session, and encode it later into operations at a lower level without the knowledge of either of its users.

Before connection to the system, each such hardware device must be certified (probably by the System Certifier) either to be an MLS entity satisfying a multi-level security policy, or to be purgeable on command from SDOS. In the latter case, SDOS must purge any such device before changing its security level of operation.

2.2.1.3 Normal Operation

During normal operation, policy parameters may be changed. Changes are intended to be effective for the entire system; however, this will be impossible if some hosts are disabled, or if parts of the network are running separately without communication. The security policy rules govern automatic changes to policy parameters that must take place when hosts with different security configurations are joined by a communication path. These rules either choose the more "conservative", or strict, value of a parameter when a choice must be made, or choose the later one if information on relative timing is available.

Perhaps the most common change to the security configuration will be changes to discretionary access rights. These changes reflect authorizations for new data, or changes to rights which reflect changes in users' responsibilities.

The security labels will be much more static. Their assignment is discussed further in the section on labels.

2.2.1.4 Modification

It will be possible to modify the system with new software to meet new conditions. There are basically two ways this may happen.

1. The old version of the system will be halted, and an entirely new one installed. This will follow the procedure outlined in the section "Installation".
2. New software may be added in the form of type managers. If the software is to operate at a single level, the extension will have no impact on multi-level security. However, installing a new MLS manager will require that the "manual assured extensible" parameter be enabled. There must also be a system user (the System Certifier) who will guarantee that the new software is indeed MLS. He does this either by confirming that the software is delivered from a source authorized to verify that it is MLS, or by verifying it himself. In the latter case, the System Certifier must have verification tools and security methodologies at least equal to those used in the original certification of the SDOS kernel, and he must be trained in their use.

2.2.1.5 De-installation

The system de-installation involves clearing stable storage of sensitive information. This procedure carries the risk that highly reliable data may be lost. Procedural controls may be used to prevent this.

2.2.2 Assigning Security Labels

Assigning an appropriate value to the security label of each entity is one of the more complicated aspects of the security configuration process. This is especially so since the SDOS labels may carry a great deal of internal structure. Each label, in full generality, may consist of a set of levels at which the entity is authorized to send messages. Each level, in turn, has the standard Bell-LaPadula-Biba structure of security and integrity classifications and category sets. We will consider these various components separately.

2.2.2.1 Security Levels

The prototypical part of the security level is the security classification. This classification is assigned both to users and to stable data objects by various government agencies. The classification of a data object represents an upper bound on its sensitivity, while the classification of a user represents his clearance for sensitive information.

Security categories are assigned to users and data objects in a similar fashion, although they are often used to represent distinctions among different kinds of information, and to represent a user's "need-to-know" for each kind.

2.2.2.1.1 Creating Security Categories Some security categories may be created specifically for use at a particular SDOS site. For example, an SDOS fielded at BBN may have a security category "BBN proprietary", which is used to mark local proprietary information and users who are authorized to see such information. Any categories defined in this way must be identified by the System Architect at the time of security preconfiguration; the meanings of the various categories cannot be dynamically changed.

On what basis are data objects assigned these new security categories? One straightforward answer is: an object must be assigned a category if it contains information "about" the subject of that category. If it does not contain such information it may still be assigned the category; this means that it may contain information of that category sometime in the future. In the example above, a file will be "BBN proprietary" if it contains (or may in the future contain) BBN proprietary information. Typically, the subject-matter of a data object is determined through inspection by the System Manager. It will usually be based on syntactic evidence, i.e., the appearance of keywords such as "BBN patent #", etc., will probably lead to assignment of a data object to the "BBN proprietary" category. The inspection will most likely be based on some standard interpretation of the data, rather than non-standard ones (such as encoding BBN proprietary information into the placement of punctuation marks).

Instead of relying on the syntactic form of an object's data, why not assign categories based on an object's semantic content (under some standard interpretation) instead? Following this approach may lead to difficulty. Suppose there is a mapping, *catset*, which maps from content to category sets, and suppose this mapping consistently obeys the following intuitive rules:

1. Two data objects with logically equivalent contents are assigned the same categories.

$$A = B \rightarrow \text{catset}(A) = \text{catset}(B)$$

2. The data object containing both A and B has a category set which contains both A 's category set and B 's category set.

$$\text{catset}(A \text{ and } B) \supseteq \text{catset}(A)$$

$$\text{catset}(A \text{ and } B) \supseteq \text{catset}(B)$$

3. A and not- A have the same category set (since they each answer questions "about" the same subject).

$$\text{catset}(A) = \text{catset}(\text{not } A)$$

We can show as a consequence of these three rules that no nontrivial assignment of categories is possible. Using rule 2, note that $\text{catset}(A \text{ and } \text{false})$ contains $\text{catset}(A)$, for any contents A . Using rule 1, $\text{catset}(\text{false})$ must then contain $\text{catset}(A)$. Also note that $\text{catset}(A \text{ and } \text{true})$ contains $\text{catset}(A)$, so $\text{catset}(A)$ contains $\text{catset}(\text{true})$, for any contents A . But rule 3 shows that $\text{catset}(\text{false}) = \text{catset}(\text{true})$, so $\text{catset}(A) = \text{catset}(\text{true})$, for any A . It follows that some, single, category set must be assigned to every object. This approach is therefore not useful.

In practice, syntactic evidence is used to determine subject-matter, and this violates rule 1 above. For example, a file stating " A implies A " may be given category " A ", (since it is syntactically "about" subject A), even though it is logically equivalent to a file containing "true", which need not be assigned any categories.

2.2.2.2 Integrity Levels

In what follows, we will begin by considering only complete integrity levels, undifferentiated into classifications and categories.

Data integrity has two goals:

1. Trustworthiness of data
2. Protection of data

These two goals are related, but are not the same. It is possible to have no restrictions on the modification of data, and yet maintain trustworthiness of data by recording the history of the modifications which were made. Data would then be trustworthy if the history showed that they had only been created and modified in appropriate ways.

Similarly, there could be two goals in assigning integrity levels to users:

1. To indicate that the user has a certain competence for dealing with a specific type of data;

2. To indicate that the user is trusted to make decisions about the modification of data.

As an example of the first goal, consider a system in which there is a data type "C-program". Some object of this type may be an important program for the system and could be protected by being assigned an integrity level indicating that it should only be modified by "C" programmers. Thus a user would need a "C-program" integrity level in order to modify such a protected C program.

As an example of the second goal, there may be a data object called "System Policy" which indicates what actions are allowed on the system. There would then be only a few users on the system, perhaps only the "System Manager", who would be trusted to set system policy, and so this could be enforced by making an integrity level for system policy and having only the system manager possess that level.

2.2.2.2.1 Integrity Levels for Data Objects An integrity level could be used for each type of data object in which either

1. Special skill or knowledge is needed for the correct modification of the data object.
2. Modification of the data object has important consequences for the functioning of the system, so that only users with authority can be allowed to modify the object.

2.2.2.2.2 Integrity Levels for Users Before a user is assigned an integrity level, the assigning authority must consider

1. Is the user trained or knowledgeable in the type of data protected at this integrity level?
2. Is the user trusted with the modification of data at this integrity level?
3. Is the user trained in the use of all programs which are allowed to modify data at this integrity level?

2.2.2.2.3 Integrity Levels for Programs A program must serve in two roles, first as a data object written into by a programmer, and secondly as instructions read into some newly initiated process entity. The program's integrity level should therefore be appropriate to both roles. As a data object, the program is assigned a level according to the guidelines listed above. As instructions for a process, the program may be read into a process at any lower integrity level, and must be verified (or other assurance must be given) that it will correctly handle data at any of these levels.

It is possible that no assignment of integrity level will satisfy both requirements. For example, suppose a program P has been given a "C-program" integrity level. This

ensures that only "C" programmers may modify P. It is now possible to start a process at the "C-program" integrity level which loads program P and operates on data at the "C-program" level (possibly even on P itself). This may not be appropriate if P is not assured to operate on data at that level.

In spite of this problem, integrity levels can still be usefully assigned to programs if it is recognized that the Bell-LaPadula-Biba scheme cannot be used to prevent the loss of "integrity" in all possible cases. Situations such as the one described in the previous paragraph, in which the integrity level cannot be made to serve double-duty, can be handled in either of two ways:

1. Program P may be assigned a level not including integrity category "C-program", and instead discretionary access controls will be relied on to limit the set of users who may modify P.
2. Program P may be assigned integrity category "C-program", but it will be assumed that P will typically be used for reading and writing data not marked as "C-program". Then P will typically not be invoked at level "C-program", and hence cannot be used to modify C programs improperly.

2.2.2.2.4 Creating Integrity Levels Because of the constraints in the Bell-LaPadula-Biba model of security and integrity levels, the integrity levels will form a lattice. However it may be preferable not to use all levels which are theoretically possible. If the different integrity restrictions, "C-program", "System Policy", etc., are each associated with a single integrity category bit, then it may be preferable that not all integrity category bits be independently settable. For example, there may be a bit indicating that a data object is a program of some kind, and a second bit indicating that it is in fact a "C" program. It is impossible for an object to be a "C" program without being a program, so it will never be the case that an object has the category "C-program" without also having the category "program". In this situation we may say that one category represents a subtype of another category.

Thus, there needs to be a partial ordering of the integrity category bits (not to be confused with the partial ordering on levels themselves) indicating that one integrity category is a subtype of a second integrity category. If a data object is given an integrity category corresponding to a data type, then it should also be given the categories corresponding to all of its supertypes.

The hierarchical integrity classifications are then a special case of the integrity categories. The highest integrity classification is a subtype of the next highest, and so on, with the lowest integrity classification being a supertype of all the others.

The hierarchical integrity classifications should, in the case of users, mean the degree of trust in the judgement and the reliability of the user. In the case of programs, the integrity classifications should indicate the reliability of each program, and no program should be given the highest integrity classification unless it has been verified to work

correctly (with the meaning of "working correctly" being dependent on the type of the program, as indicated by the program's integrity categories.) In the case of data objects, the integrity classification should indicate the degree of protection the data should have, which should be determined by considering the harm that would result if the integrity of the data were to be compromised.

2.2.2.3 Level Sets for MLS Entities

Each MLS entity is authorized to operate within a given set of security levels. The fact that an entity is MLS means that there is some reason, or special security policy for that entity, which guarantees that it will handle information of many levels in a trusted manner. Given this reason, the simplest way to assign a set of security levels to each MLS entity is to authorize every level. If software assurance were purely a yes/no decision, this would be a natural approach.

However, assurance is not purely binary, and some entities will not be authorized for all levels. In particular, the level sets of SDOS users are restricted. Typically, each user may operate over a range of security classifications, from UNCLASSIFIED to a maximum classification. He is also authorized to log-in with various need-to-know categories. His integrity level will also typically be chosen either from a range of integrity levels, from the lowest on the system to a maximum integrity, or as a function of his security level.

It may also be desirable to restrict the level sets of various software entities. Even though each such entity must have passed some rigorous form of certification to have been labeled as MLS, the certification should not be given complete trust.

MLS hosts may be labeled with different level sets. There are two separate reasons for this:

1. The degree of assurance of the MLS host software may be linked to the maximum level of sensitivity of data kept on the host. An example of this sort of linkage can be found in the NCSC document [DoD Guidance 85], which requires, for example, a B3 level of assurance if SECRET data is to be kept on a system used by uncleared users. Although MLS entities in SDOS should all meet the A1 level of assurance, which requires use of formal methods, it may be that some formal methods engender more confidence than others. Hosts which run managers which have a greater degree of assurance may perhaps contain more highly classified data.
2. System administrators may wish to keep data of different levels physically separate. For example, an MLS SDOS host may be certified to handle all levels of data, but if its level set has SECRET as the highest level, then it can be removed from the system with assurance that its stable storage contains no TOP SECRET data. (There may be exceptions to this, such as user-authorized requests to read-down, or other downgrading; a principal goal of certification is to identify these exceptions).

Objects within a host, such as MLS processes, may also be have limited level sets. The first of the reasons given above for hosts applies here as well. Of course, it won't make sense if an MLS process is authorized for levels which are denied to the host it's running on.

Storing an arbitrary set of levels in every label is absurdly expensive. With 2^{32} levels, for example, (a reasonable lower bound adequate for 4 classifications and 30 categories), recording an arbitrary set of levels requires up to $2^{(2^{32})}$ bits. Therefore no SDOS design will use this full generality. One simple and useful restriction would be to require each level set to be a range of levels. A range is specified by giving two levels, and the corresponding level set is then the lattice of levels greater than one and less than the other. This reduces the storage requirement to 64 bits per label.

2.2.3 Strict vs. Flexible Assignments

For the most part, each policy parameter of the configuration policy may be varied independently of the others. For example, "levels non-modifiable" may be set according to the needs of the site, regardless of the value of "auditing enabled/disabled". However, it will probably happen that certain configurations will occur more often than others. We distinguish three common site configurations, though others are possible:

1. The site may be used for research and development, either of SDOS itself or of application software. In this case, policy circumventability will be set to "experimental", and "manual assured extensible" will be enabled.
2. The site may be used primarily for document preparation and controlled dissemination. In this case, policy circumventability will be set to "tamperproof". "Levels modifiable" will most likely be enabled. "Read-downs" will be enabled, in order to allow the maximum secure access to information throughout the network, however, "manager create/abort" will probably be disabled, since the typical user will have little need to initiate new managers rapidly, or at levels which he cannot access. Auditing will be fully enabled.
3. The site may exist primarily to support embedded applications. In this case, the system will be largely autonomous, and in normal operation will require little interaction with system officers such as the System Manager. (Many of these officers may in fact be designated to be the same person). The policy parameters will be set to reflect this fact. Policy circumventability will again be set to "tamperproof". Since the system will probably be used for a static collection of tasks, operating on a fairly well-defined set of data, "levels modifiable" may be disabled. "Assured extensibility" will be disabled. "Manager create/abort" will be enabled, in order that any designated individual will have maximum flexibility to initiate the system functions. Most types will be protected, and many operations will be designated "direct use". Auditing may be "settable", so that the System Manager will have the option to disable it if its processing costs interfere with real-time processing.

2.3 The Formal Model

A primary goal of the SDOS effort is verification that the design for SDOS satisfies its security policy. It is not possible to demonstrate this correspondence directly and formally because the security policy is stated only in informal terms. To make such a demonstration possible, we must first formalize both the rules stated in the security policy, and the algorithms of the design. The model presented in this chapter is the formal statement of the security policy. By formalizing, we clarify and make precise the constraints of that policy.

In this chapter the design, whether expressed formally or informally, will sometimes be referred to as the *implementation* of the security policy and formal model.

2.3.1 Approach

Wherever possible, the model expresses policy constraints as extrinsic properties rather than intrinsic ones. An extrinsic property constrains the external behavior of the system; an intrinsic one puts relations on internal states, possibly for different system components or at different times. The emphasis on extrinsic properties aligns with the object-oriented philosophy on which the system is based: objects are defined on the basis of their behavior rather than their implementation. This distinction between internal state and external behavior makes it possible to change an object's implementation (for example, its internal representation) without changing its definition. Stating properties extrinsically retains the maximum freedom for the design. Flexibility in the design phase is important because of the need for the design to balance the system's performance against a set of system constraints, of which security is one.

Constraining behavior naturally requires first describing that behavior. To describe behavior, we concentrate on the history of a system's interaction with its environment. A history of interaction is composed of *events*, which are actions the system may take. The model will describe acceptable (secure) relations among events in a given system history, and (secure) relations among different possible system histories. The meaning of the events will not be defined in the model; if the policy that the model describes is to be a meaningful one, then most or all of its events should have an intuitive meaning for outside observers. For example, the event *login user U on terminal T* should ideally have meaning in terms of particular keystrokes and display at terminal T. Perhaps this meaning would be: type the name and password for user U and receive a login acknowledgement. If there is also an event *logout user U on terminal T*, then a simple example of a policy on an actual history of events of this system might be: two *login* events are always separated in time by a *logout* event.

This general approach to specification is often called the *trace* approach, where a trace is defined as a totally ordered sequence of events. This approach is developed in great detail in [Brookes et al. 84]. The trace approach is especially attractive for distributed systems, since the notion of an abstract global *state* of such a system may

be hard to define. It is also attractive because a policy constraining histories of external events for the entire system may be decomposed into policies on the histories of nodes processing in parallel. The behavior of one node naturally involves the communication events between that node and its neighbors. A policy for that node would be a constraint on histories of events of two kinds: communication events with neighbors, plus any of the system's external events that are produced by the node.

We have made some slight modifications of the trace approach. The events in a history will be only partially ordered, rather than totally. In the next section we describe the form used to express the constraints on events and event histories.

2.3.2 The Language of the Formal Model

The framework of the model is a list of definitions of the various logical components mentioned in the security policy, along with statements which show relationships among these components. Generally, the abstract entities out of which SDOS is built are encapsulated as *types*. Elements of the various types have attributes that are represented by *functions*. The policy rules are then formalized as first-order logic sentences, presented under the heading *policy*, which show how the functions are related.

Some types and functions are defined in terms of other types and functions appearing in the model, and some are left unspecified. During the process of verification, the unspecified types and functions will be given meaning by being put in correspondence with concrete features of the design.

2.3.2.1 Types

Types are defined first in the model and are presented in section 2.3.5.1. They are organized in a top-down fashion, with the most encompassing types listed first. The types of the model include all of the system components mentioned in the security policy (e.g. hosts, users, entities), plus the type *event*.

The entities explicitly identified in the instantiated message-passing policy are listed here as subtypes of the type *entity*. A subtype, *object*, refers to all abstract objects of the object model. A subtype of type *object*, *code_object*, refers to certain passive containers of data which hold the code executed by SDOS processes. These objects are singled out so that policies controlling the modifiability of assured processes can be stated.

The type *label* is constructed to be a set of *levels*. These in turn are constructed from the unspecified types of security and integrity classifications and categories.

As in the security policy, the type *event* has been defined to be equivalent to the type of *send_message* operations. Every event is the sending of a message, and vice-versa. Complex system activities are constructed from sets of *send_message* events. In particular, the abstract operation invocations of the object model involve a collection of related *send_message* events. In order that an abstract operation invocation be re-

lated in time to other events, each *abstract_operation_event* is identified with the final *send_message* event which relays successful completion back to the requesting client. If the abstract operation invocation does not complete successfully, then there is no *abstract_operation_event*, although there will be other *send_message* events resulting from the invocation.

The type *command* is a subtype of the abstract operation events. These commands include all changes to the system's security configuration.

2.3.2.2 Functions

Functions are defined next in the section 2.3.5.2. Unlike types, the functions are organized in a bottom-up fashion, with unspecified functions listed first. Functions defined in terms of other functions then always refer to previously defined functions.

Every event type has parameters that are unique to that type and which distinguish between different events of that type. For example, different login events can be distinguished if they log in different users. Section 2.3.5.2.1 gives a list of unspecified functions which return these parameters.

Section 2.3.5.2.2 consists mostly of functions which distinguish events according to their order in time. SDOS histories will in general be unordered sets of events. However, an important part of most constraints on event histories is event sequencing. A distributed system consists of a set of computers executing in parallel, and it may not be desirable to assume that there exists an absolute, global time which gives an order to every pair of events. In the model we begin by assuming that there are concurrently executing *nodes*, some or all of which are the SDOS hosts. For each node there is a total ordering of events. The function *sequenced* indicates the order of two events that occur on a particular host. From these orderings for each node, plus the existence of *send_message* events which pass between nodes, we define a single partial order for all events, *causal*. *Causal* is used repeatedly in the model to decide which information about the values of security policy parameters can be known at a particular event.

The configuration of SDOS may change with time. Functions are defined which return the system's configuration as it can be known at each event *X*. These functions are always defined in terms of previous events that can be causally connected to the event *X*. The rules of the policy are then enforced at each event in a manner appropriate to the configuration known at that event.

The partial order defined by *causal* is extended into a total order in the function *before*. Events which are not ordered by *causal* are assumed to be ordered by some mechanism in SDOS, perhaps by physical clocks on each node. This ordering *before* serves only one purpose in the model: when disconnected subnetworks are reconnected, they must make their security configurations consistent. In some cases, discussed in the security policy, the choice between different configurations is made based on which configuration was established later.

This method of ordering events using actual `send_messages` between nodes, was inspired by [Lamport 78]. Note that each `send_message` event represents actual, successful communication. If a communication path between nodes is not working, then no `send_message` between them may occur, even though the nodes may attempt to communicate.

The functions in section 2.3.5.2.3 define a partial order of levels in the standard fashion.

The functions in section 2.3.5.2.4 define the system's security configuration at any event in a history. They include definitions of all policy parameters found in the security policy. For example, the value of an entity's security label which should be used in mandatory access control at event *A* is defined in terms of the latest event which should have changed the level at *A* and which could have causally affected event *A*.

The final section presents several special-purpose functions. Included are constant functions representing the special SDOS users. Also included is an unspecified function `security_relevant`, which identifies some SDOS commands as more important than others for enforcement of the policy. Every instance of any command type mentioned in this model is security relevant, while other abstract operation invocations need not be.

2.3.2.3 Policy

The principal rules of the policy are stated in the sentences marked as **policy**. These are divided into sections of configuration, discretionary, and mandatory rules, which correspond directly to the three parts of the security policy.

Implicit in each policy statement is a universal quantification over sets of events which can be produced by the system. Each statement can be thought of as prefaced by: "for all histories *h*, if *h* is produced by the system, then ...". Each statement then asserts a particular property about *h*. Unless indicated otherwise, events referred to in a policy statement are elements of this implicit history *h*.

Most **policy** statements are also universal quantifications over the set of events in the history. This corresponds to an invariant for the execution of the system. When other events are mentioned in a **policy** statement, these other events are usually related to the event in the outermost quantification by the *causal* relation. This is equivalent to saying that the event *now* under consideration bears some relation to events in the past. If the policy rules did not take this form, they could be stating *liveness* properties of the system, i.e., that given condition *A*, the system will produce event *B* eventually. As in the security policy, we have avoided such liveness assertions.

The only case not covered by the previous paragraph is the policy statement for MLS entities. This policy statement includes an explicit quantifier over histories, but also a relation between two separate but possible system histories.

There are other, ancillary rules of the policy stated as **restrict** statements. These

restrictions apply both to types and to functions. For types, they usually restrict one type to be a subset of another. For functions, they usually place some constraint on a single function, rather than on a combination of several. In both cases of types and of functions, the restrictions were needed because we either could not or did not choose to define a type or function in terms of more primitive ones. For example, the type *command* is restricted to be a subset of *abstract_operation_event*. Certainly in the SDOS implementation, the type *abstract_operation_event* will be some union of various events, including those of type *command*. We chose not to define it as such a union here, however, because the *abstract_operation_events* that are not commands (as defined in the model) cannot be known in advance: they are defined during system operation by adding new type managers. These unspecified events are not mentioned elsewhere in the model, except as elements of the type *abstract_operation_event*.

As another example, the function *label* is not given directly as a definition. Instead, it is restricted by a rather lengthy relation among those previous events that could possibly have affected the label in question. Without further information, such a restriction may lead to more than one interpretation of the function *label* in the SDOS implementation. If so, the model considers them equally acceptable. (If such a restriction, or any policy rule, leads to no possible interpretations, then there can be no implementation which satisfies the policy.)

2.3.3 Implementation and Verification

The goal of the verification of the system is to show compliance of the system design with the policy as expressed in the formal model. The model defines acceptable histories of events. The events of these histories should have some intuitive meaning to users of the system, since it is their relation that defines the acceptable, secure behavior of the system. Therefore, the first step in the verification process is relating unspecified elements of the model to concrete features of the design.

Verification may be accomplished through the steps outlined below.

1. Give a concrete interpretation in the design to all unspecified types, including all unspecified event types.
2. Show that all relations between types, (e.g. $\subset$), hold in this interpretation.
3. Give a concrete interpretation in the design to all unspecified functions.
4. Demonstrate that all restrictions on functions hold in this interpretation.
5. Demonstrate that all policy statements hold in this interpretation.

Making this correspondence properly, i.e., giving unspecified elements of the model an intuitive interpretation in the design, is a crucial part of the verification process. If this correspondence were made arbitrarily (for example, if the interpretation of *login*

events were exchanged for the interpretation of *logout* events) the policy rules (for *login* and *logout*) would have little meaning for the system's users. The correspondence must also be complete. If the interpretation of *login* events in the design covered only some of the cases for which users thought they correctly followed a login sequence, then again the policy rules would have little significance for the real system. Types and functions which are not unspecified are defined in terms of other types and functions. Usually these elements of the model are meant to correspond to features of the design which have intuitive meaning. This is not necessary, though. The entire formal model could be rewritten by replacing each occurrence of a defined type or function with its definition. In this expanded form, verification may begin after just the correspondence with unspecified elements is given.

In the formal model, we have not biased the choice of language(s) in which the design is to be expressed. However, we expect that a formal description of the design will be given in Gypsy. In that case, "concrete features of the design" will mean particular Gypsy constructs. For example, a correspondence will be given between types of the formal model and Gypsy types. The entities of the model may correspond to Gypsy procedures, and events to Gypsy buffer communication, etc.

2.3.4 Notation

Before any mechanical specification and verification system can be used to prove that some design correctly implements this model, both the design and the model must be expressed in the language of that system. Each specification and verification system language has its own idiosyncrasies and limitations which prevent one from saying clearly and directly what needs to be said. Therefore, we have avoided expressing the model in the language of any particular system. Even though we expect the model will need to be restated in Gypsy notation, using that notation now would cause us to make rather arbitrary, possibly inappropriate, decisions about the Gypsy constructs to be used, and in particular, about those constructs used to model events and histories.

The notation we have used is not complicated, and this fact should allow it to be translated easily into most verification system languages. The following is a description of the notation used in the model:

- **Type Construction Operators**

A new type is constructed from existing types using one of three set construction operators.

- **Union**

$type1 \cup type2$ is the type of all instances either of *type1* or of *type2*.

- **Direct product**

$type1 \times type2$ describes a new type whose instances are ordered pairs of instances of the component types. If i is of *type1* and j is of *type2*, then (i, j) is the instance of $type1 \times type2$ with these components. If i is of $type1 \times$

type2, then *i.type1* describes an instance of *type1* that is the first component of *i*. (In cases for which *type1* and *type2* are the same type, they will be given distinguishing names.)

- **Powerset**

powerset(*type*) describes a new type whose instances are sets with elements of type *type*.

- **In**

$i \in \text{type}$ holds if *i* is an instance of type *type*. $i \in \text{set}$ holds if *i* is an element of set *set*.

- **Subset**

$\text{type1} \subset \text{type2}$ holds if every instance of *type1* is an instance of *type2*. $\text{set1} \subset \text{set2}$ holds if every element of *set1* is an element of *set2*.

- **Quantification**

$\forall i:\text{type}$ means universal quantification over type *type*. $\exists i:\text{type}$ means existential quantification over type *type*.

- **Propositional connectives**

and, *or*, *xor*, *not*, $\rightarrow$, and $\leftrightarrow$ are the usual propositional connectives.

- **Function declarations**

funct(*arg1:type1*, ..., *argn:typen*) : *type* declares a function named *funct* with arguments of various declared types, and which returns a value of type *type*. If *argi* is of type *typei* for *i* in [1..n] then *funct*(*arg1*, ..., *argn*) is the value of the function applied to those arguments.

- **Conditionals**

if A then B else C is an abbreviation for a function which returns the value B if A is true, and returns the value C otherwise. A must be a boolean value, and B and C must be of the same type.

- **Comments**

Lines of comments describing the features of the model are preceded by bullets (•).

We have avoided abbreviating the names of types and functions except in extreme cases. This is intended to preserve as much of the intuitive content as is possible. Many names will be overloaded. For example, the name *entity* stands for a type, and it also stands for several different functions. These ambiguities can always be resolved by context. In particular, the different functions with the name *entity* may be distinguished by the type of their argument.

2.3.5 Model

2.3.5.1 Types and Restrictions on Types

2.3.5.1.1 Entities The *entities* which are instances of these types are the abstract entities out of which SDOS is built.

type entity

type host

restrict host $\subset$ entity

type link

restrict link $\subset$ entity

type location = host $\cup$ link

- hosts are any entities performing processing in parallel.
- links are entities which hosts use for communication.
- a location is either a host or a link; it is an entity which is spatially separated from other locations.

type user

restrict user $\subset$ entity

type group = *powerset*(user)

- users are the human users of SDOS.

type object

restrict object $\subset$ entity

type abstract_type = *powerset*(object)

- objects are the abstract objects of the object-oriented paradigm.

type client

restrict client $\subset$ object

type terminal_interface

restrict terminal_interface $\subset$ client

type type_manager

restrict type_manager $\subset$ client

type primary_client = user $\cup$ type_manager

- clients invoke abstract operations on objects.
- every client acts on behalf of some primary_client.

type binding = name $\times$ object_uid

restrict directory $\subset$ object

type name

type object_uid

type directory = *powerset*(binding)

- a name is a local identifier which is bound to a global identifier for some object.
- the binding is stored in a directory.

type code_object

restrict code_object $\subset$ object

type code_objects = *powerset*(code_object)

- some objects are designated to hold the code for SDOS system objects.
- SDOS processes load their executable code from objects of type *code_object*.

type message_content

2.3.5.1.2 Labels The following label types will be used in the mandatory security policy. They are constructed in the standard fashion from component types.

type label = level_set

type level = security_level $\times$ integrity_level

type level_set = *powerset*(level)

type security_level = security_classification $\times$ security_categories

type integrity_level = integrity_classification $\times$ integrity_categories

type security_category

type security_categories = *powerset*(security_category)

type integrity_category

type integrity_categories = *powerset*(integrity_category)

type security_classification

type integrity_classification

2.3.5.1.3 Events SDOS execution histories will be sets of events of the types listed below. Several points should be noted about the intended interpretation.

- Every event is a *send_message* event. Most *send_message* events are internal communication, occurring in the model only as intermediates for enforcing global information flow rules.
- Some *send_message* events are of subtype *reply_message*. Each *reply_message* event will be associated with some *send_message*.
- *Send_message* events that are responses to clients are always of subtype *reply_message*, and of subsubtype *abstract_operation_event*. These are the SDOS commands of various kinds. They are interpreted as happening when a command successfully completes and notifies the requesting client of its completion. Related events, such as the issuing of a command, internal changes and messages, and final, external notification of failure are not *abstract_operation_events*; they are other, undistinguished, events of the parent type *send_message*.
- Different events of the same type will differ in some of their attributes, such as the time and place at which they occurred. Attributes will be listed in the *functions* section.

type event = *send_message*

type history = *powerset*(event)

type *send_message*

type *reply_message*

restrict *reply_message* $\subset$ *send_message*

type *request_to_read*

restrict *request-to-read* $\subset$ *send_message*

type *create*

restrict *create* $\subset$ *send_message*

type destroy

restrict destroy $\subset$ send_message

type abstract_operation_event

restrict abstract_operation_event $\subset$ reply_message

type command = DAC_command $\cup$

set_experimental $\cup$

set_levels_modifiable $\cup$

label_command $\cup$

set_extensible $\cup$

enable_read_downs $\cup$

disable_read_downs $\cup$

enable_manager_create/abort $\cup$

disable_manager_create/abort $\cup$

protect $\cup$

modify_controlling_group $\cup$

set_direct $\cup$

set_auditing_off $\cup$

set_auditing_on $\cup$

set_auditing_settable $\cup$

enable_auditing $\cup$

disable_auditing $\cup$

restrict_directories $\cup$

unrestrict_directories $\cup$

log_command $\cup$

directory_command

restrict command $\subset$ abstract_operation_event

type DAC_command = grant $\cup$ rescind

type label_command = set_label $\cup$ set_MLS $\cup$ set_single_level

type log_command = login $\cup$ logout

type directory_command = add_binding $\cup$ delete_binding

type grant, rescind

type set_label

type set_MLS, set_single_level

type protect

```

type modify_controlling_group
type set_direct
type set_experimental
type set_levels_modifiable
type set_extensible
type enable_read_downs, disable_read_downs
type enable_manager_create/abort, disable_manager_create/abort
type set_auditing_off, set_auditing_on, set_auditing_settable
type enable_auditing, disable_auditing
type restrict_directories, unrestrict_directories
type login, logout
type add_binding, delete_binding

type audit_value = {auditing_off, auditing_settable, auditing_on}

type abstract_operation

```

2.3.5.2 Functions and Restrictions on Functions

2.3.5.2.1 Parameters of Events Every event has parameters which depend on the type of the event. The following functions return those parameters.

```

function sender(send_message) : entity
function receiver(send_message) : entity
function message_level(send_message) : message_level
function message(send_message) : message_content

function client(abstract_operation_event) : client
restrict  $\forall$  aoe: abstract_operation_event( receiver(aoe) = client(aoe) )
function abstract_operation(abstract_operation_event) : abstract_operation
function object(abstract_operation_event) : object

```


function target(DAC_command) : primary_client

function target_operation(DAC_command) : abstract_operation

- for granting and rescinding discretionary access rights, these
- functions return the client for which the right is granted, and
- the operation being controlled.

function entity(label_command) : entity

function label(set_label) : label

function label(create) : label

function abstract_type(protect) : abstract_type

function abstract_operation(set_direct) : abstract_operation

function group(modify_controlling_group) : group

function binding(directory_command) : binding

function user(log_command) : user

2.3.5.2.2 Distributed Phenomenology For a distributed system, there need not be an unambiguous way to order the system's events in time. Therefore, the values of the security policy parameters will be defined unambiguously as functions of events, rather than as functions of time. Because SDOS is distributed, its underlying network may become partitioned into subnetworks which cannot communicate. Therefore, the value of any policy parameter at event A must be defined only in terms of events that can causally affect A. The following functions define *can causally affect* phenomenologically, i.e., in terms of other events which could provide a causal link.

function location(event) : location

restrict

$\forall sm: \text{send\_message}(\text{sender}(sm) \in \text{location} \rightarrow \text{sender}(sm) = \text{location}(sm))$

- every send_message will be associated with some location.
- every inter-location send will be associated with the location from which it was sent.

function location_ordered(loc:location, ev:event) : boolean =
 location(ev) = loc or (ev $\in$ send_message and receiver(ev) = loc)

- a location *loc* will impose a total ordering on every event *ev* satisfying this predicate.
- the ordering includes all events directly associated with *loc*, plus all messages received from other locations.

function sequenced(event, event) : boolean

restrict

$\forall$ ev1: event(not sequenced(ev1, ev1))

$\forall$ ev1, ev2: event(
 if location_ordered(location(ev1), ev2) or
 location_ordered(location(ev2), ev1)
 then not ev1=ev2 $\rightarrow$ sequenced(ev1, ev2) xor sequenced(ev2, ev1)
 else not sequenced(ev1, ev2))

$\forall$ ev1, ev2, ev3: event(
 sequenced(ev1, ev2) and sequenced(ev2, ev3) $\rightarrow$ sequenced(ev1, ev3))

- *sequenced* is a total ordering of events for each location.
- pairs of events which cannot be ordered at a single location are not related by *sequenced*.
- pairs of events which are ordered at more than one location must be sequenced in the same order for each.

function causal(ev1:event, ev2:event) : boolean =
 (sequenced(ev1, ev2) and location_ordered(location(ev2), ev1)) or
 $\exists$ sm: send_message(sequenced(sm, ev2) and causal(ev1, sm))

restrict

$\forall$ ev1: event(not causal(ev1, ev1))

$\forall$ ev1, ev2: event(causal(ev1, ev2) and causal(ev2, ev1) $\rightarrow$ ev1=ev2)

$\forall$ ev1, ev2, ev3: event(
 causal(ev1, ev2) and causal(ev2, ev3) $\rightarrow$ causal(ev1, ev3))

- *causal* is a partial ordering of all events.
- if two events are *causal*, then there is a chain of send_messages which could have propagated information from one to the other. it must therefore be possible to causally connect one to the other.

function before(ev1, ev2) : boolean

restrict

$\forall ev1: \text{event}(\text{not before}(ev1, ev1))$

$\forall ev1, ev2: \text{event}(\text{not } ev1=ev2 \rightarrow \text{before}(ev1, ev2) \text{ xor } \text{before}(ev2, ev1))$

$\forall ev1, ev2, ev3: \text{event}(\text{before}(ev1, ev2) \text{ and } \text{before}(ev2, ev3) \rightarrow \text{before}(ev1, ev3))$

$\forall ev1, ev2: \text{event}(\text{causal}(ev1, ev2) \rightarrow \text{before}(ev1, ev2))$

- *before* is a total ordering of all events which extends the partial order defined by *causal*.

function *originating*(*reply_message*) : *send_message*

restrict

$\forall rm: \text{reply\_message}(\text{sender}(rm)=\text{receiver}(\text{originating}(rm)) \text{ and } \text{receiver}(rm)=\text{sender}(\text{originating}(rm)))$

restrict

$\forall rm: \text{reply\_message}(\text{sequenced}(\text{originating}(rm), rm))$

- every reply was evoked by the send returned by *originating*.

function *initial_send*(*send_message*) : *send_message*

restrict

$\forall sm: \text{send\_message}(\text{initial\_send}(sm) = sm \text{ or } \exists prev: \text{send\_message}(\text{sequenced}(prev, sm) \text{ and } \text{sender}(sm) = \text{receiver}(prev) \text{ and } \text{message}(sm) = \text{message}(prev) \text{ and } \text{initial\_send}(sm) = \text{initial\_send}(prev)))$

function *initial_sender*(*sm: send_message*) : *entity* = *sender*(*initial_send*(*sm*))

- a message may be passed along via a chain of entities.
- *initial_send* returns the first send in the chain.

function *exist*(*ent:entity*, *ev:event*) : *boolean* = $\exists cr: \text{create}(\text{receiver}(cr)=ent \text{ and } \text{causal}(cr, ev) \text{ and } \forall ds: \text{destroy}(\text{receiver}(ds)=ent \text{ and } \text{causal}(ds, ev) \rightarrow \text{before}(ds, cr)))$

- defines whether an entity exists.
- all possible entities are assumed to exist potentially;
- actual existence is brought about by sending a *create* message.

2.3.5.2.3 Levels and Labels These functions are needed to define relations among levels.

function lteq(security_classification, security_classification) : boolean

restrict

$\forall$ sc1: security_classification(lteq(sc1, sc1))

$\forall$ sc1,sc2: security_classification(
not sc1=sc2 $\rightarrow$ lteq(sc1, sc2) xor lteq(sc2, sc1))

$\forall$ sc1,sc2,sc3: security_classification(
lteq(sc1, sc2) and lteq(sc2, sc3) $\rightarrow$ lteq(sc1, sc3))

- security_classifications are totally ordered.

function lteq(integrity_classification, integrity_classification) : boolean

restrict

$\forall$ ic1: integrity_classification(lteq(ic1, ic1))

$\forall$ ic1,ic2: integrity_classification(
not ic1=ic2 $\rightarrow$ lteq(ic1, ic2) xor lteq(ic2, ic1))

$\forall$ ic1,ic2,ic3: integrity_classification(
lteq(ic1, ic2) and lteq(ic2, ic3) $\rightarrow$ lteq(ic1, ic3))

- integrity classifications are totally ordered.

function lteq(sl1:security_level, sl2:security_level) : boolean =
lteq(sl1.security_classification, sl2.security_classification) and
sl1.security_categories $\in$ sl2.security_categories

function lteq(il1:integrity_level, il2:integrity_level) : boolean =
lteq(il1.integrity_classification, il2.integrity_classification) and
il1.integrity_categories $\in$ il2.integrity_categories

function dominates(l1:level, l2:level) : boolean =
lteq(l2.security_level, l1.security_level) and
lteq(l1.integrity_level, l2.integrity_level)

2.3.5.2.4 Security Configuration These functions return the current configuration of the policy parameters at a given event.

```
function experimental(ev:event) : boolean =
     $\exists$  on: set_experimental( causal(on,ev) )
```

```
function tamperproof(ev:event) : boolean = not experimental(ev)
```

```
function levels_modifiable(ev:event) : boolean =
     $\exists$  on: set_levels_modifiable( causal(on,ev) )
```

```
function levels_nonmodifiable(ev:event) : boolean =
    not levels_modifiable(ev)
```

```
function change_label(ent:entity, ev:event) : boolean =
    ((ev  $\in$  set_label or ev  $\in$  create) and entity(ev)=ent)
```

```
function label(ent:entity, ev:event) : label
```

restrict

```
 $\forall$  ent, ev: entity(  $\exists$  newest: event(
    change_label(ent,newest) and
    causal(newest,ev) and
     $\forall$  new: event(
        change_label(ent,new) and causal(new,ev)  $\rightarrow$  before(new,newest))
    and label(ent,ev) = label(newest)
```

- this function defines the current label of an entity.
- the label is determined by the latest change_label event that can be causally connected to the current event.

```
function manual_assured_extensible(ev:event) : boolean =
     $\exists$  on: set_extensible( causal(on,ev) )
```

```
function MLS(ent:entity, ev:event) : boolean =
     $\exists$  on: set_MLS(
        entity(on) = ent and causal(on,ev) and
         $\forall$  off: set_single_level(
            entity(off) = ent and
            causal(off,ev)  $\rightarrow$  before(off,on)))
```

```
function single_level(ent:entity, ev:event) : boolean = not MLS(ent,ev)
```

```
function read_downs_enabled(ev:event) : boolean =
     $\exists$  on: enable_read_downs( causal(on,ev) and
     $\forall$  off: disable_read_downs( causal(off,ev)  $\rightarrow$  causal(off,on)))
```

```
function read_downs_disabled(ev:event) : boolean = not read_downs_enabled(ev)
```

```
function manager_create_abort_enabled(ev:event) : boolean =
  ∃ on: enable_manager_create_abort( causal(on,ev) and
  ∀ off: disable_manager_create_abort(
    causal(off,ev) → causal(off,on)))
```

```
function manager_create_abort_disabled(ev:event) : boolean =
  not manager_create_abort_enabled(ev)
```

```
function abstract_type(object) : abstract_type
```

- this function defines a unique association between each object
- and its abstract type.

```
function protected(at:abstract_type, ev:event) : boolean =
  ∃ p: protect( abstract_type(p)=at and causal(p, ev))
```

```
function controlling_group(o:object,ev:event) : group
```

```
restrict
```

```
∀ usr: user( usr ∈ controlling_group(o:object,ev:event) ↔
  ∀ newest: modify_controlling_group(
    object(newest) = o and causal(newest,ev) and
    not ∃ newer: modify_controlling_group(
      object(newer) = o and causal(newer,ev) and
      causal(newest,newer) ) →
    usr ∈ group(newest) ))
```

- the current controlling group is the intersection of the groups named in all modify commands which have not been superceded for this object.

```
function direct_use(ao:abstract_operation, ev:event): boolean =
  ∃ on: set_direct( abstract_operation(on)=ao and causal(on,ev) )
```

```
function auditing(ev:event) : audit_value =
  if ∃ off: set_auditing_off( causal(off, ev) )
  then auditing_off
  else if ∃ set: set_auditing_settable( causal(set, ev) )
  then auditing_settable
  else auditing_on
```

```

function auditing_disabled(ev:event) : boolean =
   $\exists$  off: disable_auditing( causal(off,ev) and
     $\forall$  on: enable_auditing( causal(on,ev)  $\rightarrow$  causal(on,off)))

```

```

function auditing_enabled(ev:event) : boolean =
  not auditing_disabled(ev)

```

```

function directories_unrestricted(ev:event) : boolean =
   $\exists$  off: unrestrict_directories( causal(off,ev) and
     $\forall$  on: restrict_directories( causal(on,ev)  $\rightarrow$  causal(on,off)))

```

```

function directories_restricted(ev:event) : boolean = not directories_unrestricted(ev)

```

```

function in_directory(d:directory, b:binding, ev:event) : boolean =
   $\exists$  add: add_binding(
    object(add) = d and
    binding(add) = b and
    causal(add,ev) and
     $\forall$  del: delete_binding(
      object(del) = d and
      binding(del) = b and
      causal(del,ev)  $\rightarrow$  before(del,add) ))

```

```

function primary(client) : primary_client

```

```

function primary_client(aoe:abstract_operation_event) : primary_client =
  primary( client(aoe) )

```

- every client which is not primary is acting on behalf of some primary client.

```

function logged(u:user, ti:terminal_interface, ev:event) : boolean =
   $\exists$  in: login(
    user(in) = u and
    client(in) = ti and
    causal(in,ev) and
     $\forall$  out: logout(
      user(out) = u and
      client(out) = ti and
      causal(out,ev)  $\rightarrow$  before(out,in) ) )

```

```

restrict

```

```

 $\forall$  u: user, ti: terminal_interface, ev: event(
  logged(u,ti,ev)  $\rightarrow$  primary(ti)=u )

```

- this function returns true if this terminal interface is associated with this user.

- a terminal associated with a user is acting on behalf of that user.

```

function discretionary_right(aoe:abstract_operation_event, ev:event) : boolean =
   $\exists$  g: grant(
    object(g) = object(aoe) and
    target(g) = primary_client(aoe) and
    target_operation(g) = abstract_operation(aoe) and
    causal(initial_send(g),ev) and
     $\forall$  r: rescind(
      object(r) = object(aoe) and
      target(r) = primary_client(aoe) and
      target_operation(r) = abstract_operation(aoe) and
      causal(initial_send(r),ev)  $\rightarrow$ 
      before(initial_send(r),initial_send(g))))

```

- defines whether a discretionary right for event *aoe* exists at event *ev*.

2.3.5.2.5 Other Useful Functions

```

function system_architect() : user

```

```

function system_manager() : user

```

```

function system_auditor() : user

```

```

function system_certifier() : user

```

```

function system_controller() : user

```

- these functions are constants which denote users who act in special system roles.

```

function object(object_uid) : object

```

- a mapping from uids to objects that they represent.

```

function assured(ent:entity, ev:event) : boolean =
  MLS(ent,ev) or ent  $\in$  terminal_interface

```

```

function code(entity) : code_objects

```

- a fixed mapping from entities to executable code which they may read. This mapping is unimportant for entities which are not assured.

```

function security_relevant(send_message) : boolean

```


restrict

$\forall \text{ com: command(security\_relevant(originating(com)))}$

- all command events are originated by security-relevant request messages.

function internal_form(message_content) : message_content

- a mapping from a user-recognizable command form, to an internal form.

2.3.5.3 Policy

The policy rules are grouped here into the same categories as can be found in the SDOS security policy.

2.3.5.3.1 Configuration Policy**policy**

$\forall \text{ config, ev: abstract\_operation\_event(}$
 $\text{primary\_client(config) = system\_architect() and}$
 $\text{not primary\_client(ev) = system\_architect() } \rightarrow$
 $\text{causal(config, ev))}$

- all preconfiguration of the system must be done by the system architect.
- this preconfiguration includes, conceptually at least, creation and initialization of some entities and their attributes, such as their labels.

policy

$\forall \text{ ev: set\_experimental(primary\_client(ev) = system\_architect())}$

policy

$\forall \text{ sm: send\_message(tamperproof(sm) } \rightarrow$
 $\text{(initial\_sender(sm) } \in \text{ code\_object and}$
 $\text{assured(receiver(sm), sm) } \rightarrow$
 $\text{initial\_sender(sm) } \in \text{ code(receiver(sm))))}$

- for a tamperproof system, code that is loaded into assured processes
- must be taken from some unmodifiable set of code objects.

policy

$\forall sm: \text{send\_message(tamperproof(sm) } \rightarrow$
 $\quad (\text{receiver(sm)} \in \text{code\_object and}$
 $\quad \text{not sm} \in \text{request\_to\_read and}$
 $\quad \exists \text{ent: entity(}$
 $\quad \quad \text{receiver(sm)} \in \text{code(ent) and}$
 $\quad \quad \text{assured(ent,sm)) } \rightarrow$
 $\quad \forall \text{ent: entity(}$
 $\quad \quad \text{receiver(sm)} \in \text{code(ent) } \rightarrow$
 $\quad \quad \text{ent} \in \text{type\_manager) and}$
 $\quad \quad \text{initial\_sender(sm)} \in \text{client and}$
 $\quad \quad \text{primary(initial\_sender(sm)) = system\_certifier()))}$

- for a tamperproof system, the code loaded into an assured process may be modified only if that process is a type manager, and only if the modification is made by the system certifier.

policy

$\forall ev: \text{set\_levels\_modifiable(primary\_client(ev) = system\_architect())}$

policy

$\forall ev: \text{event(levels\_nonmodifiable(ev) } \rightarrow \text{not ev} \in \text{set\_label)}$

- if levels are not modifiable, then labels may not be explicitly reset once an object has been created.

policy

$\forall ev: \text{set\_label(primary\_client(ev) = system\_manager())}$

policy

$\forall ev: \text{set\_extensible(primary\_client(ev) = system\_architect())}$

policy

$\forall ev: \text{event(ev} \in \text{set\_MLS) } \rightarrow \text{manual\_assured\_extensible(ev))}$

- the set of MLS entities cannot be expanded unless the system is manual_assured_extensible.

policy

$\forall com: \text{command(}$
 $\quad com \in \text{set\_MLS or}$
 $\quad com \in \text{set\_single\_level } \rightarrow$
 $\quad \text{primary\_client(com) = system\_certifier())}$

policy

$\forall$ com: command(
 com $\in$ enable_manager_create/abort or
 com $\in$ disable_manager_create/abort or
 com $\in$ enable_read_downs or
 com $\in$ disable_read_downs $\rightarrow$
 primary_client(com) = system_manager())

policy

$\forall$ com: command(
 com $\in$ set_auditing_off or
 com $\in$ set_auditing_settable or
 com $\in$ set_auditing_on $\rightarrow$
 primary_client(com) = system_architect())

policy

$\forall$ com: command(
 com $\in$ enable_auditing or
 com $\in$ disable_auditing $\rightarrow$
 primary_client(com) = system_manager() and auditing_settable(com))

policy

$\forall$ com: command(
 com $\in$ restrict_directories or
 com $\in$ unrestrict_directories $\rightarrow$
 primary_client(com) = system_manager())

policy

$\forall$ ev: event, d: directory, b: binding(
 in_directory(d,b,ev) $\rightarrow$
 dominates(label(object(b.object_uid)).current_level, label(d).current_level)
 and
 (directories_restricted(ev) $\rightarrow$
 label(object(b.object_uid)).current_level = label(d).current_level))

- a directory's level is dominated by the level of any object whose name it holds. If directories are restricted, each object named in a directory has the level of that directory.

2.3.5.3.2 Discretionary Policy

policy

$\forall$ aoe: abstract_operation_event(
 $\exists$ ti: terminal_interface(logged(primary_client(aoe), ti, aoe)))

- a user must be logged in for requests under his authorization to be honored.

policy

$\forall$ ev: protect(primary_client(ev) = system_certifier())

policy

$\forall$ ev: modify_controlling_group(
primary_client(ev) = system_controller())

policy

$\forall$ ev: command(ev $\in$ grant or ev $\in$ rescind $\rightarrow$
primary_client(ev) $\in$ controlling_group(object(ev), ev))

policy

$\forall$ ev: set_direct(primary_client(ev) = system_controller())

policy

$\forall$ aoe: abstract_operation_event(
direct_use(abstract_operation(aoe), aoe) $\rightarrow$
primary_client(aoe) $\in$ terminal_interface)

- direct-use operations must come from user's terminal.

policy

$\forall$ aoe: abstract_operation_event(
protected(abstract_type(object(aoe)), initial_send(aoe)) $\rightarrow$
initial_sender(aoe) $\in$ type_manager and
discretionary_right(aoe, initial_send(aoe)))

- if an operation governed by discretionary access control is completed, then a type manager is responsible, and the discretionary rights for the operation must be known by the type manager at the time of the operation. This is a constraint on type managers, rather than on the behavior of the entire system.

2.3.5.3.3 Mandatory Policy

policy

$\forall sm: \text{send\_message}(\text{exist}(\text{sender}(sm), sm))$

- entities which do not exist do not send messages.

policy

$\forall ent: \text{entity}, ev: \text{event}(\text{single\_level}(ent, ev) \rightarrow$
 $\forall l1, l2: \text{level}(\text{$
 $l1 \in \text{label}(ent, ev) \text{ and}$
 $l2 \in \text{label}(ent, ev) \rightarrow l1 = l2))$

- single-level entities must be labeled with unique levels.

policy

$\forall sm: \text{send\_message}(\text{$
 $\text{level}(sm) \in \text{label}(\text{sender}(sm), sm))$

- message label must be one for which the sender is authorized.

policy

$\forall sm: \text{send\_message}(\text{$
 $(sm \in \text{request\_to\_read} \text{ and } \text{read\_downs\_enabled}(sm)) \text{ or}$
 $(sm \in \text{create} \text{ and } \text{receiver}(sm) \in \text{type\_manager} \text{ and}$
 $\text{manager\_create/abort\_enabled}(sm)) \text{ or}$
 $sm \in \text{leak} \text{ or}$
 $\exists lev: \text{level}(\text{$
 $lev \in \text{label}(\text{receiver}(sm), sm) \text{ and}$
 $\text{dominates}(lev, \text{level}(sm))))$

- receiver must be authorized to receive a message with this label.

policy

Every MLS entity must be restrictive with respect to every level. (This policy rule cannot be stated in the language we have used for the rest of the Formal Model. See sections 2.1.2.3 and 4.2 for formal statements of restrictiveness.)

Chapter 3

Design

3.1 The Functional Description

3.1.1 Introduction

The purpose of the functional description is to describe the decomposition of the system into logical components, the assignment of functions to those components, and the description of approaches likely to be taken to implementing those functions. In this section we begin by considering the hardware and software that makes up the distributed environment to identify the context for the system. We then consider our approach to developing the design, of which this description is the first step. Section 3.1.2 describes the functions that make up the system and the principles that underlie their definition.

3.1.1.1 SDOS Implementation Strategy

Implementation strategy refers to the set of hardware and software technologies used in implementing the host and network facilities making up the distributed system. Constraints on the usable implementation strategies will often limit the achievable functionality of a system, since certain items of functionality can only be achieved with certain technology. Conversely, constraints on the functionality (absolute requirements) will constrain the implementation strategies to those that are able to support the required functionality. This section discusses a number of implementation strategies and points out the advantages and disadvantages of each, including the functionality that can and cannot be achieved using each strategy.

3.1.1.1.1 Features And Assurance The attributes of a secure system include, among other things, security features and assurance. Security features include such things as user authentication, discretionary access controls, mandatory controls, and auditing.

Assurance is a measure of an evaluator's confidence that the security features are correctly implemented and protected from tampering. Assurance is achieved by means of some combination of hardware protection features, advanced software design and implementation technology, project management and source control procedures, code inspection, penetration testing, and formal verification.

Functionality is usually thought to be synonymous with features; but in a secure system, assurance is at least as important a part of functionality as features are. The Criteria specify, for each evaluation class, a detailed set of assurance requirements, as well as a set of required security features. It is convenient to name assurance levels after their corresponding evaluation classes - thus, "B2 assurance" means that which is achieved by following the assurance requirements in the B2 section of the Criteria. In a distributed system, whose components are implemented on many different computers, it is useful to talk about the assurance level of each security feature, or the assurance of the individual components that implement each security feature, rather than the assurance of the system as a whole.

It is a fundamental property of assurance that the assurance of a component, C, can be no greater than the assurance of the components "below" C. Below is loosely defined, such that a component, B, is below C if 1) C depends on B for proper operation (e.g., C calls B), or 2) B is able to access C's databases (e.g., B is in a lower, more privileged ring than C).

In a traditional, single-host, layered operating system, the relationship between components is clear, straightforward, and obvious. The components in lower layers or rings are below those in higher layers or rings. In a distributed operating system, the relationships between components are much more complex. Different components can be physically located in equally-privileged, mutually-suspicious protection domains (e.g., separate computers). Further, the logical relationship between components can be the reverse of their physical relationship. For example, in a layered communications protocol, a higher layer may do encryption and then pass the encrypted data to a lower layer that need not be trusted since it has no security function. Of course, the lower, insecure layer must not be able to access the higher, secure layer's databases.

Each security feature specified in the Criteria has a minimum required assurance level. For example, it would be pointless and unacceptable to implement mandatory controls in a component having less than B level assurance. However, it is acceptable to implement discretionary controls in a component with C level assurance, and mandatory controls in a separate component having B or A level assurance.

Given the above considerations, an implementation strategy must be chosen that allows each of the required security features to be implemented in a component having the appropriate level of assurance.

3.1.1.1.2 Alternative Strategies In addition to security features and assurance, we must consider time and cost of implementation of the SDOS, preservation of in-

vestment in existing application and operating system software, and any specialized requirements of the applications to be run on the SDOS.

There are two implementation strategies for distributed operating systems: native and layered. A native DOS implements all distributed functions (such as interprocess communication, distributed authentication, and distributed file system functions) as well as conventional operating system functions (such as process scheduling, memory management, and I/O control). In contrast, a layered DOS is executed as a layer on top of the OS on each host. The native approach is adopted by the Mach DOS [Young et al. 87], while the layered approach is adopted by the Cronus DOS [Schantz et al. 86].

Native DOS's have the benefit of providing a uniform set of OS services, such as emulation of UNIX by Mach. However, they have the disadvantage of being unable to integrate machines having different operating systems. The layered approach adopted by Cronus allows the DOS to run on heterogeneous hosts, in a layer above existing operating systems such as UNIX and VMS. The existing operating system is called the constituent operating system (COS).

There are four implementation strategies for secure layered DOSs. Strategy 1 consists of adding security features to an existing DOS. This has the obvious disadvantage that, if, for example, the underlying operating systems have only C2 assurance, the resulting SDOS could have no more than C2 assurance. This is unacceptable because multilevel secure (MLS) implies at least B2 assurance, and our objective is a MLS SDOS with an A1 rating. This alternative is mentioned only for completeness.

Strategy 2 consists of porting a DOS to a MLS COS (having assurance of at least B2 and preferably A1), and then adding the appropriate security features. This would involve partitioning the DOS into security-relevant and non-security-relevant parts, and making the former part of the TCB.

This strategy has two variants: Strategy 2A, the native approach, involves writing a MLS COS that provides exactly those features needed to support SDOS; and Strategy 2B, the layered approach, involves using an existing MLS COS and building SDOS on top of it.

Strategy 2 has the advantage of being MLS, but it has the disadvantages of losing heterogeneity and losing investment in much of the existing application software. Strategy 2A has the additional disadvantages of long development time and high cost for the COS, while 2B avoids these, with zero COS development time, and COS development cost shared by all the customers who purchase rights to the existing MLS COS. In addition, the existing COS would have at least some software development tools that run on it, and possibly also some useful application software. There are several variations on strategy 2A that depend on the degree to which the COS functions are integrated with the DOS functions. However, the integration issue has a greater impact on performance and adaptability than cost.

Strategy 2B is clearly preferable on cost grounds, since the MLS COS does not have to be developed. Strategy 2A would only be chosen if no suitable existing MLS COS

could be found. Strategy 2A may be adopted as a longer-term strategy for enhanced performance if existing application software can be preserved.

Strategy 3 involves the use of secure front-ends or access machines to connect existing C2-rated DOS hosts to an insecure network. Mandatory controls would be implemented in the MLS access machines, while discretionary controls would be implemented in the C2 hosts. This strategy has the advantages of preserving heterogeneity and preserving the investments in the existing DOS, the applications that run on it, and all the usable software development tools and application software that runs on the existing COSs. It has the additional advantage of lower SDOS development cost than either alternative in strategy 2. However it has the disadvantage of not providing for any MLS hosts. This makes some classes of SDOS application systems very awkward, if not impossible, to implement.

Strategy 4 is a hybrid of strategies 2 and 3. It involves use of a MLS COS, on which SDOS software would be built, which could serve either as a secure front-end to connect existing single-level systems, or as an MLS SDOS host, running SDOS application software. This combines the advantages of strategies 2 and 3: preservation of heterogeneity and investment in existing application software, and provision for MLS SDOS hosts. Implementation cost would be somewhat higher than that of strategy 2 alone, but not significantly so, if the hybrid nature of this strategy were in the design from the beginning. This strategy has one additional advantage over either of strategies 2 or 3: existing, non-distributed applications, that need to be interfaced to a distributed system, but which cannot be changed for reasons such as extreme age, would require not only security in the interface machine, but also some non-security-related processing, to convert old data formats and communication protocols into those used by modern networks and hosts. The hybrid MLS SDOS machine provides both capabilities in one machine.

Strategy 4 has two variants, corresponding to 2A and 2B. The hybrid software could be built either on an existing MLS COS (4B), or on an MLS COS written as part of SDOS development (4A). As with strategy 2B, 4B is preferable, provided that a suitable existing MLS COS can be found.

3.1.1.1.3 Discussion of Strategies Strategies 3 and 4 share an important advantage: the ability to connect insecure DOS hosts to the SDOS. There will always be some specialized hosts, such as very-high-speed CPUs, or LISP machines, whose use is essential to the success of the application, but which are unlikely to ever have a MLS operating system. The access machine approach allows for connection of such hosts to the SDOS.

Strategy 2 has the serious disadvantage of loss of heterogeneity. This means that the entire SDOS must be composed of one type of CPU, running one MLS COS. All investment in existing applications is lost, unless they can be ported to the new COS. No provision is made for connecting specialized, insecure hosts to the SDOS. (Porting SDOS to several MLS COSs, and possibly writing MLS COSs for several types of CPU,

would be prohibitively expensive.)

Strategy 4 provides for heterogeneity without incurring the cost of SDOS (and possibly COS) development for several different CPUs.

Ranking the strategies by cost and functionality: Strategy 4 provides the most complete functionality and has the highest cost. Strategy 2 is a close second in cost but has the serious disadvantage of loss of heterogeneity. Strategy 3 is lowest in cost but has the disadvantage of not providing for MLS hosts. Strategy 1 was eliminated because it is not MLS.

3.1.1.2 System Environment

The architectural characteristics of the computer system is an important consideration in choosing an approach for the SDOS (and any other) development effort. The distributed environment being considered consists of a set of hosts connected by a communication subsystem. Initially, the targeted communication subsystem is a local area network. Expansion to a more diverse environment is dependent on low-level communication protocols, encryption capabilities, and proper labeling of messages on each host. These considerations are orthogonal to the model adopted for interprocess communication in SDOS, and other communication media can be considered at a later point.

A constituent operating system (COS) refers to the native operating system on a host. SDOS (or any other distributed operating system) can either be implemented as the native operating system on a host or on top of the COS. There are several classes of host/COS system pairs that are candidates for integration into the SDOS environment. Each host class has a different impact on the implementation of SDOS on hosts of the class. Viewing SDOS as a technology for interconnecting systems, the following list contains the major classifications of machines that could be usefully connected:

- **Native SDOS Hosts**

Hosts that have no constituent operating system can run SDOS as the native OS. While the difficulty of reconciling the SDOS security policy with COS security is not an issue on native SDOS hosts, native implementation of a full-scale operating system is a major undertaking that is not merited in the early phases of this effort.

- **Evaluated Multi-level Secure Hosts**

This class contains systems that have been evaluated by the DoD Computer Security Center using the Trusted Computer System Evaluation Criteria and have received an A or B rating. They support mandatory security, which would have to be reconciled with the SDOS definition of multi-level security, as well as discretionary access controls.

- **Evaluated Hosts Providing Access Controls**

This class contains systems that have been evaluated by the DoD Computer Security Center using the Trusted Computer System Evaluation Criteria and have

received a C rating. As with A and B systems, the SDOS discretionary access controls are likely to provide finer grained access control than these systems. These hosts will execute at a single security level, and the difficulty is providing secure mechanisms in the kernel that enforce the SDOS information flow rules.

- **Minimal Protection Time-Shared Systems**

These systems provide unreliable or inadequate security features, and cannot be trusted to distinguish between individual users accurately. Minimal protection hosts will execute at a single security level and access control will be supported at the level of the entire host.

- **Workstations**

Workstations are powerful, sophisticated single-user machines in which the owner acts as both user and administrator for the host. Unless administrative control can be removed from the user, the software executing on these machines is subject to corruption. Workstations can be used in SDOS only if they access the communication subsystem through a trusted interface that reliably labels messages with its source and security level, and mediates all interaction between the workstation and the network.

- **Access Machines**

Access machines provide secure interfaces for collections of terminals and nonsecure hosts to the distributed system. The services they provide can range from simply labeling messages with the proper source and security level to participating in user authentication and client identification, and implementing layers of the interprocess communication protocol.

The envisioned use of SDOS is as a facility to connect both single-level and multi-level hosts into a single secure distributed system. We believe such an integrated system is the most feasible given our perception of the evolution of computer systems over the next decade and beyond. Specifically, single-level systems are necessary because such nonsecure systems (connected to a secure network) will continue to be 5 to 10 years more advanced and considerably more widely accepted than their multi-level secure counterparts. Furthermore, they will be considerably more specialized, providing capabilities customized for supporting (for example) artificial intelligence and computationally intensive applications. As a result, their inclusion in a system will be crucial.

MLS hosts are also an important part of SDOS. Multi-level objects (e.g., databases) and tightly coupled objects (symbolic name directories) require configuration together on a single host for performance reasons. MLS hosts also simplify information flow between security levels when the flow does not violate the security policy.

The application base intended for SDOS are large, complex distributed applications that are adequately robust to survive failures and are well-structured to facilitate their evolution. Our experience in distributed systems has convinced us of the importance of supporting heterogeneity in the hardware and software used to construct distributed sys-

tems. Single level and MLS hosts are but one example of this type of heterogeneity likely to be encountered in computer installations needing secure distributed applications.

3.1.1.3 The Approach

The work to date on the SDOS project has concentrated on adapting an object-based architectural model for structuring the system to include multi-level security features, and on developing a global security policy that is applied to each host. Software in the SDOS is organized by decomposing the system into a set of basic system concepts and treating all of these concepts in a uniform way. The basic system concepts are framed as objects, or instances of abstract datatypes. Such basic objects include users, hosts, processes, and files. A type is a set of objects that are all accessed in the same way, with a set of operations defined by the type. New types can be created, and types serve as the basic constructors for building software. Any application can be expressed as a set of new types. The object model used in SDOS was adapted from the Cronus DOS under development at BBN Laboratories with funding from Rome Air Development Center, where the effectiveness of the underlying model has undergone extensive evaluation.

Since every host in a heterogeneous system will have its own definition and implementation of security, a global security policy is necessary as a foundation for allowing different hosts to interact securely. The task of connecting a host into SDOS involves reconciling the local security policy with the global security policy, determining the assuredness of software running on the host, and selecting the appropriate security attributes of the host.

Another aspect of host integration is the implementation of SDOS services necessary to allow the host and the distributed system to be interconnected. These services must be implemented with sufficient assurance to provide multilevel security. This means they must have a Computer Security Center B2 assurance at a minimum. The assurance of these services determines the assurance of the SDOS. Thus, B3 or A1 are desirable goals.

If the SDOS services are implemented within a host, on top of the COS, they can have no greater assurance than that of the COS. The resulting SDOS can have no greater assurance than that of the lowest-rated host.

One approach is to implement the SDOS services in an access machine which connects the host with the network. The access machines can be homogeneous (the same hardware and operating system being used for all access machines). The access machine software (operating system and SDOS services) can have high assurance at relatively low cost, since the volume of software will be small. This approach allows an SDOS with A1 assurance to be built out of hosts having much lower assurance (C2 or even D). This approach has been identified as the one that holds greatest promise for short term results in connecting single level machines and demonstrating distributed system security. However, as discussed in Section 3.1.1.1.3, this approach results in functional limitations because of its inability to support MLS hosts.

The construction of multi-level secure hosts to support SDOS is considerably more difficult than connecting existing single level hosts to a secure network. For this reason, we concentrated our effort on the design of a multi-level secure host. This approach offers the opportunity to investigate difficult verification problems (e.g., how to ensure that MLS services maintain separation of information in different access classes) and examine difficult design decisions during this effort.

Our approach to the SDOS development effort consisted of the following overlapping phases:

- **Host Selection**

Started in parallel with the prototype design, this stage involved examining host and corresponding secure constituent operating system candidates that could form the implementation base for SDOS. The lack of availability of supported multi-level secure systems on the market greatly limits the choices available.

- **Kernel Prototype Design**

It is first necessary to experiment with the SDOS kernel (described in section 3.6.2) and evaluate its adequacy and efficacy in providing interprocess communication and multi-level security services based on the underlying services provided by the implementation base selected in the first phase. We developed a top level specification describing the programmer interface to the SDOS services, a detailed design specification describing the SDOS internals (including layering, definition of objects and operations, and the security relevant operations), and a lower-layer specification describing the relationship between SDOS and the underlying system.

- **Kernel Prototype Development**

A prototype kernel should be quickly developed to experiment with many of the design issues. Depending on the maturity of the development facilities on the secure host, the prototype kernel could be developed on a *development system* host providing a good programming environment. The kernel could then be implemented as an application on the COS (e.g., UNIX, VMS). Such an implementation would provide useful performance comparisons with the Cronus DOS. As the development environment for the selected secure machine matures to the point of making implementation on it feasible, the implementation could be ported onto the secure machine.

- **Test and Evaluation**

Although evaluation is needed within every phase, an evaluation of the prototype will determine if the object model is an appropriate structuring concept in a secure system and if the coexistence of SDOS with constituent systems is feasible. Evaluation involves analysis of performance characteristics, security violations (such as denial of services, covert channels, and compatibility between security policies) that cannot be corrected, and ease of use. Future steps depend on the results of the evaluation.

3.1.2 Principles Underlying SDOS Functions

3.1.2.1 Introduction

SDOS is a distributed operating system intended for supporting the secure execution of application software running on a distributed computer system consisting of a diverse set of hardware and software resources. Support for distributed applications requires provision for the following activities:

1. System administration and operation,
2. System development and maintenance, and
3. Application development and maintenance.

SDOS is modeled after other distributed operating systems that support the concept of data abstraction as a fundamental and pervasive architectural principle for software design. The SDOS orientation toward security will necessarily limit flexibility and have a significant impact on performance, based on the experiences of other secure operating system development efforts. Despite these concerns, we view the additional security requirements of SDOS as an evolutionary extension of general-purpose distributed operating systems. In order to support the development, maintenance, and operation activities listed above, the following services will need to be provided in an SDOS implementation:

- **Object Management**

Objects are the major components used to build the operating system and applications. An object is an instance of a type of resource; the type determines the set of operations that can be invoked to access the object. Examples of objects include files, large data structures, devices, and hosts. The operations invoked on objects are performed by object managers. The use of the object paradigm improves the system's modularity, leads to well-specified interfaces between system components, and provides a uniform approach to structuring applications and system services.

- **Process Management**

The process is one of the object types in the SDOS system; a process is an object used to perform a task. Processes may act as object managers by responding to operation invocations, act as clients by invoking operations on objects, and issue system calls. Large applications are decomposed into asynchronously executing processes in much the same way as large data structures are decomposed into a set of objects.

- **Interprocess Communication (IPC)**

Processes communicate by sending messages through a message switch on each host. IPC is based on a client-server relationship between correspondents established by a protocol for invoking operations on objects. Messages are dispatched in a host transparent fashion, and the message switch on each host is responsible for delivering messages to their destination.

- **Secure Information Transfer**

SDOS transfers information between system components in correspondence with the security policy, which models all transfer as message transfers between entities. The major points of transfer that are relevant to the functional description are: one process sending a message to another; an object manager accessing the state of an object; communication between a host and the communications subsystem; and communication between a process and a device.

- **Access Control**

Access to information is always modeled in SDOS as access to an object. Controlling access to objects involves two parts: authenticating the identity of a client, and authorizing the client's access to an object based on the client's identity and the object's type.

- **Symbolic Naming**

SDOS maintains two name spaces: system-level names (unique identifiers, or UID's) used to create, locate, and destroy objects; and user-level symbolic names chosen by the SDOS users and used to identify objects in applications. The symbolic names are maintained in a hierarchical structure managed by the Catalog Manager, whose primary responsibility is to map symbolic names into their corresponding UID's.

- **User Interfaces**

There are two aspects of humans using the SDOS system: obtaining access through an access point, and interacting with the system through a user interface. Certain actions can only be safely and securely initiated by human users. It is essential that secure access points are provided to allow system access through a user interface which is assured to correctly translate user requests and forward them properly for processing. Other users interfaces can also be developed for specific applications or general-purpose system interaction.

- **Controlled Software Development**

Much of the software that executes in a secure system must meet a set of criteria that establish its reliability and security. One means of ensuring that these criteria are satisfied is to establish a controlled environment for software development whereby it is impossible to execute software without first demonstrating its trustworthiness. SDOS will provide a software development facility that simplifies the task of developing secure software and helps to prevent execution of insecure software.

- **Configuration, Monitoring, and Control**

SDOS will provide the basic facilities necessary to manually operate the system.

The remainder of this section describes each of these functional areas in more detail.

3.1.2.2 Object Management

The purpose of object management is to provide a means of manipulating the state of an object in response to client's requests to access the object. There are several goals of organizing a system in terms of objects maintained by object managers:

- Hiding the details of how an object is implemented from the clients (users, applications, and system services) that use it allows an object's implementation to change without affecting the clients using it.
- The clean interface between clients and objects reduces the likelihood of interface errors.
- A single structuring paradigm allows mechanisms to be applied uniformly across all object types (e.g., access control and interprocess communication).

After initialization, the manager of an object commonly repeats the following tasks:

1. Receive an operation invocation request from a client;
2. Check the privilege of the client to access the object;
3. Execute the operation, which can cause the object to be modified;
4. Returns the results of the operation to the client;
5. Waits until the next operation invocation request is received.

All objects have a state saved on stable storage in a repository called the object database. When an object manager receives an operation invocation request, it obtains the object's state from stable storage and begins access authorization.

Object management in SDOS is based on the following principles:

- Each object has a system-level, unique identifier (UID) used to locate the object and direct invocation requests to the object's manager from anywhere in the system. The UID of an object is not based on the object's host location.
- Each object has a single type. An object's type determines the operations that can be used to access the object and the set of manager processes that can service an invocation. The operations defined by an object's type are the protocols used to access that object.
- An object can be accessed only through its manager. A manager implements the interface to objects; objects may actually be implemented in a variety of ways. For example, an object may be a device, a remote host, or a resource defined by software developed independently of SDOS (e.g., a commercial database product).

- New objects are created by a manager associated with the object's type.
- An object may be used to represent any instance of a resource, including a data structure, a component of a database, a device, and a host. A file is an object used as the building block for all applications in file-based operating systems. Although the file type of object is supported in SDOS, it is only one way of storing data. The object is a structuring concept that is used for building both applications and SDOS services.
- There is flexibility in the mapping of objects to managers. A manager process may service requests from a single client or from multiple clients; and may manage one object of a single type, many objects of a single type, or objects of many different types.
- New object types can be added to the system by creating an object manager's image and configuring the manager in the system. SDOS is designed to be extensible, where new types and instances of those types can be introduced into the system by users with the privilege to do so.
- Any object can be given a symbolic name as a handle for users referencing the object. All complex distributed applications in SDOS will be structured as a set of clients and objects. However, the user interface of an application may not make this structure visible to the user.

3.1.2.3 Process Management

A process is an object used to complete a task, and collections of processes serve as the active elements used to build applications. The three defining characteristics of the processes in SDOS are:

1. A process is the unit of sharing operating system resources, including memory, I/O devices and the processor on the local host. Thus, there is a low-level connection between a process and the host on which it executes.
2. The internal activity of a process, namely the access of its memory, is distinct from its external activity, namely interprocess communication and system calls, in terms of performance, security, and operating system involvement. The primary role of the operating system is with the external actions of a process by providing a facility to create processes and allow them to communicate securely using an object-based protocol.
3. Processes serve a dual role as both clients for and managers of resources. A client process includes a user interface process that allows a user to interact with the system, or an object manager that invokes operations on objects different from the ones it manages. A process is a manager of an object if it is able to service invocations requests addressed to the object.

Process management in SDOS involves a mechanism to create processes, modify their attributes as they execute, and destroy them. Process management is based on the following attributes:

- Processes are modeled in the same way as other types of objects. All processes have UIDs used to address them, and all the processes on a host have a process manager that services requests invoked on them. This yields a uniform approach to active and passive components of an application. Conformity to the object model also provides extensibility, the ability to create new types from existing ones. As a result, a simple file type may be used to build more specialized type implementations, such as fast or replicated files.
- Process creation is flexible to support a variety of different uses. A process can be automatically initiated by SDOS or selectively in response to specific requests. Manager processes are automatically initiated when a host is booted or to service an unhandled invocation request. Any type of process may be selectively initiated by a process with appropriate privilege requesting to create one.
- Access authorization is based on the privileges of processes. Processes have identities bound to them which are used to authorize their actions as clients to access other objects (see section 3.6.4.1).
- Processes may be transient or perpetual. A process has access to stable storage to save its state and allow its perpetual execution despite failures.
- Processes may be initiated when the system boots, when services are initiated, when users log into the system, or when processes are spawned to accomplish a specific task in an application.

3.1.2.4 Interprocess Communication (IPC)

The SDOS interprocess communication facility is responsible for reliably and securely relaying messages between processes located anywhere in the system. The message switch located on each host is responsible for message routing and delivery.

IPC in SDOS is based on the following principles:

- A client/server protocol is used based on addressing objects for operation invocations.
- At the highest layer, message delivery is directed on the basis of a host-independent object identifier. Object identifiers are mapped to processes on specific hosts using a facility to locate objects by the message switch. Addressing is useful for operations on a collections of objects of a type or for operations on "unknown" objects (e.g., to create an object).

- At the highest layer of communication, interhost and intrahost communication are treated uniformly by basing addressing on objects and types rather than hosts.
- Replicated objects are supported. By allowing the optional designation of the destination host in addition to the object identifier when addressing a message, a client can route a message either to any copy of an object or to a specific copy.
- An initial SDOS implementation should be based on the DoD standard (nonsecure) internet (IP) and terminal transmission (TCP) protocols to facilitate speedy prototyping.

3.1.2.5 Secure Information Transfer

The SDOS security policy models all information flow as instances of the send-message operation issued by entities. Determining whether information flow is secure in the system will involve establishing a mapping between entities in the policy and system components in an implementation, and identifying all instances of component interactions that are send-message operations. There is not a one to one correspondence between the policy entities and objects: each object is an entity, but many entities are not implemented as objects (e.g., the IPC facility).

The asymmetry of objects and entities derives from the difference in how objects and entities are defined. Objects are used to structure a system, and are defined at a level in which they are manipulated by manager processes and addressed by client processes. The decision to define an object is a pragmatic one based on how to organize a collection of software, and involves consideration of software performance and complexity.

In contrast to an object, an entity is defined as the components of a system which information flows into and out of, regardless at the level of their implementation. Entities are used to model an implementation. As a result, although entities all have the same set of attributes, in practice they may be implemented quite differently. For example, all entities have security labels. All entities that are objects have security labels that are explicitly represented in the system. Other entities, particularly components of the operating system, may have labels with no explicit representation in the implementation.

Ensuring secure communication flow in SDOS is based on a multi-level security policy. The SDOS implementation will involve the use of several different mechanisms that together are modeled in the security policy as the secure send-message operation. Examples of these mechanisms are the interprocess communication facility, the object database facility, and the communications subsystem.

Secure information flow in SDOS is based on the following properties:

- Every object has a security label. The security label can include levels for sensitivity and integrity, need-to-know categories, and a security attribute (single-level or multilevel secure). Labels are set by the System Manager and System Certifier.

- Every IPC message has a security label, set by the message sender, that is validated by the IPC mechanism prior to delivery. A message's label is invalid if it is incompatible with the label of its creator (as defined by the security policy). Messages with invalid labels are rejected. The labels of messages originating from a process are validated by the IPC mechanism; the labels of messages originating from the IPC mechanism are assumed to be accurate.
- Every IPC message is forwarded toward its destination by the IPC mechanism only if the forwarder can ascertain that the message can be forwarded securely to the next component in the message's path. A message is forwarded securely if the message's label is consistent with the receiving component's label. Examples of message forwarding is between hosts and from the message switch to a process.
- Object labels are separated from objects and their managers. A security database maintained on each host contains the label of every object located on the host and every host in the system. The message switch uses the security database to validate message labels and deliver them securely. The concentration of labels in a single repository on each host facilitates high performance message delivery. Labels are referenced by addressing the host they are located on rather than the object associated with the label.

3.1.2.6 Discretionary Access Control

The purpose of a discretionary access control mechanism is to restrict accesses to objects based on the identity of the accessors and the way they are attempting the access. This section examines the general properties and functions proposed for the SDOS discretionary access control mechanism.

The components involved in discretionary access control are:

- *Clients*: active user or system components that have specific identities and that request access to resources. A client is commonly implemented as a process.
- *Resources*: objects, which are instances of object types; an object's type defines the operations through which object's of the type may be accessed.
- *Object managers*: perform operations on objects they manage that are requested by clients, and authorize client's privilege to request the access.

Discretionary access controls prevent unauthorized clients from accessing objects. These controls are provided through two mechanisms: *client authentication*, where a client is given an identity for access control purposes; and *access authorization*, where a client's request to access an object is granted or denied. Access is granted if the client has the *privilege* to access the object.

3.1.2.6.1 Properties of SDOS Access Control The discretionary mechanisms supported by SDOS will have the following properties:

- **Unique client identities:** Each user and each system service has a unique identity that is known throughout the system.
- **Single client log-ins:** Users log into any host in the system once per session and provide their user name and password for authentication. Passwords are protected in the system by a one-way transformation and are stored in encrypted form. The transformation is applied to the password supplied by a user logging into the system. If the transformed value matches the stored value then authentication is successful; otherwise, the user is not permitted to continue.
- **Specificity in client identification:** Client identities will allow clients to be distinguished based on:
 - Unique identifiers at the granularity of user or service. This will allow privileges to be given to individual users or services.
 - Association of the client with an organization of clients. This will allow privileges to be given to groups of clients based on their participation in a group or organization (e.g., a department).
 - The activity of the client. This will allow privileges to be given to clients based on a role or responsibility they have in a large, complex application (e.g., data entry, monitoring, and operational control).
- **Host independence:** Access control is performed in a host-independent fashion. The location of the client is not considered in the decision to grant access to an object.
- **Type-specific privileges:** Access control privileges may be defined independently for each type. Access to an object is limited to the set of operations defined by the object's type. Privileges are based on the abstract operations defined by each type.
- **Uniformity across types:** Although privileges are defined in a type-independent fashion, the activities performed by users span many different types. By defining the *roles* of clients, there is a uniform basis for identifying the activities of clients across different types.
- **Authorized nested invocations:** Clients may act as *proxies* for other clients. This will allow one client to act on another's behalf. Any client may transfer a proxy identity as part of an invocation. This permits the manager handling the invocation to act on the client's behalf in invoking other operations.
- **Intermodule connection control:** A manager receiving an invocation is always able to distinguish the client's actual identity from the proxy identity it passes. An object type can be designed to only allow specific software (e.g., another manager)

to invoke certain critical operations on them. However, the client invoking the critical operation is permitted to invoke it on behalf of other users.

- **Direct operations:** Certain critical operations should only be invocable by a human user through a trusted interface. Such direct operations will be protected by a mechanism which is able to distinguish clients that are users from those that are not.
- **Nondiscretionary controls:** Access privileges for each type will be divided into discretionary and nondiscretionary parts. The nondiscretionary privileges may only be assigned by the System Controller. The discretionary privileges may be assigned by any client in the *controlling group* of the type, which is set by the System Controller.
- **Automated access control list initialization:** An access control list for a newly created object is automatically initialized based on properties of the object's creator and the creator of the type.

3.1.2.6.2 Discretionary Access Control Functions Access authentication and access authorization are the two functions that comprise discretionary access control. Access authentication is the interaction between a terminal interface process (TIP), the Authentication Manager (AM), and a Process Manager for the TIP.

- **Terminal interface process:** responsible for (iteratively) accepting the user name and password of a human user, relaying this information to the Authentication Manager, returning an indication of the success of the authentication (and possibly reprompting for name and password if the login fails), and executing the appropriate user interface software for the particular user.
- **Authentication Manager:** The Authentication Manager accepts or rejects the authentication attempt by searching for the user name and password supplied by the TIP in its password database. If this check succeeds then the Authentication Manager sets the bindings, or process identity, of the TIP to correspond to the identity associated with the user name. This binding is set by interacting with the Process Manager for the TIP.
- **Process Manager:** The Process Manager is responsible for maintaining the process bindings of the processes it manages. It accepts requests exclusively from Authentication Managers to assign the bindings of processes.

Access authorization occurs when a process invokes an operation on an object. The purpose of the authorization is to determine if the process has the privilege to access the object with the operation. Authorization involves the comparison of the identity, or process bindings, of the invoking process against its privileges, as designated by an access control list, to use the object. The following describes the role of each component involved in access authorization:

- **Client process:** requests access to an object by invoking an operation on the object.
- **Object manager:** receives the access request, obtains the process bindings for the client process, and performs the access authorization. Client process bindings are obtained from the Process Manager of the client process. The success of authentication depends upon the privileges defined by the type of the object, the particular privileges granted to the client process for the object, and the identity of the client.
- **Process Manager:** responds to requests by object managers to obtain process bindings for processes it manages.

3.1.2.7 Symbolic Naming

Symbolic naming allows resources in the system to be easily identified by users. The Catalog Manager is the service responsible for providing the symbolic naming capability. This service maintains a set of user-defined symbolic names for objects that it can translate into system-level unique object identifiers (UID), which are in turn used to locate and access objects. Objects having symbolic names are said to be cataloged in the directory. Since the sole purpose of symbolic naming is to make a system easier to use for the human user, the catalog service is a user interface facility. The SDOS symbolic naming facility is based on the following principles:

- The symbolic name space is global and host-transparent. Any object in the system can be cataloged by the service, regardless of its location or any other property of the object. The catalog service is accessible from anywhere in the system as a by-product of SDOS object-based IPC.
- The symbolic name space is type independent. The name of any type of object may be cataloged by the catalog service.
- Symbolic names all have the same syntactic form. This means that symbolic names can be manipulated without concern for the type or location of objects they name.
- The selection of every symbolic name is under the control of its creator. No constraints are placed on naming because of the concern for a clear, flexible user interface. User-controlled naming allows symbolic names to be a conduit for communicating sensitive information. Information leakage is avoided by placing security labels on cataloged information, and forcing access to the catalog to conform to the security policy information flow rules.
- Symbolic names are loosely tied to UIDs. One object may be cataloged under several different symbolic names to allow different users to select their own names independent. Objects are not required to be cataloged; some objects are never directly named by users or are placed in private name spaces using separate services. In contrast to symbolic naming, each object only has a single UID.

- The symbolic name space is structured. The enormous number of objects that can populate a distributed system necessitate the organization of symbolic names into a structure that allows users to quickly find names. The two most important requirements are to be able to group names of related objects, and represent simple relationships between different groups. A hierarchical name space has been adopted in SDOS, forming a large tree. Inner nodes of the tree are directories, objects managed by the Catalog Manager that contain a list of symbolic names. Leaf nodes are the catalog objects.
- The catalog is dispersed among many Catalog Managers on different hosts. Dispersion allows the catalog to scale in a way that avoids performance and resource bottlenecks, but can cause catalog lookups to span several hosts. A multiple host lookup facility will be supported by the Catalog Manager.
- The symbolic name space is replicated to ensure availability and improve performance. Catalog copies are kept consistent through a replication protocol implemented within each Catalog Manager that distributes catalog updates.

3.1.2.8 User Interfaces

The purpose of the user interface is to provide human users with an easy to understand, reliable method of interacting with the SDOS system. The user interface is responsible for interpreting user commands, initiating the activities that perform the requested tasks, and returning the results accurately to the user.

User interfaces have increased significantly in sophistication with the proliferation of graphics capabilities on workstations and personal computers. This growth has naturally been accompanied by an enormous increase in the complexity of user interface implementations. A key property of a user interface in a secure system is that it interpret user commands reliably and securely, particularly for commands having a critical impact on the security of the system, and present the corresponding results accurately. Demonstrating that these properties are satisfied by a complex user interface is extremely difficult.

The approach adopted with SDOS is to provide three types of user interface:

- **Basic Interface**

Presents the actual low-level model of objects that make up the system to the user. This interface is intended to be very simple and is used for security critical commands where reliability is essential. Commands to this interface correspond to (sets of) operation invocations on objects. The Basic Interface constitutes a trusted path to system components.

- **Extended Interfaces**

A more standard set of user interfaces, implemented graphically or textually, that allows users to issue more complex commands than provided in the Basic Interface.

These interface do not map commands into operations on objects, but may present the object-oriented flavor of the system to the user.

- **Special Purpose Interfaces**

Examples of special purpose interfaces are application interfaces and subsystem interfaces, such as to the Configuration, Monitoring, and Control System. These interfaces are developed independently with no attempt at uniformity. Diverse special purpose interfaces are useful for allowing access to applications in the way best-suited for their use.

The SDOS approach to user interfaces is based on the following principles:

- Validation of the correct performance of the Basic Interface is essential for the control of security-critical aspects of the system, such as the assignment of security labels, modification of access privileges, and selection of the system parameters designated in the security policy. Validation is made feasible by the Basic Interface acting as a simple interpreter of object-based operations.
- Security-critical commands can only be initiated from within the Basic Interface. It is not only necessary to ensure that such important commands are interpreted correctly by the Basic Interface, but also that these commands cannot be issued through another interface that may incorrectly interpret them. This is achieved in conjunction with access control by specially designating processes that execute the Basic Interface.
- The Basic and Extended Interfaces are extensible to allow the growth of the interface to parallel the growth of the system. As new types are created, the interface should allow users to interact with those types in an appropriate way. The Basic Interface can be table driven by the set of types in the system. The Extended Interfaces will be modularized by command and extended on a command by command basis.
- The Basic and Extended Interfaces allow the execution of programs that provide Special Purpose Interfaces to the user, and permits the user to interact with these interfaces. This feature is necessary to allow Special Purpose Interfaces to temporarily replace one of the standard system interfaces.
- A single command to the Basic Interface can initiate a sequence of operations on objects. This makes it possible to build more complex commands from simple ones and thereby make the interface easier to use.
- The Extended Interfaces allow the user to execute several commands simultaneously, including commands that initiate the execution of other programs. The user may switch between different execution threads to allow concurrent activities.

3.1.2.9 Controlled Software Development

The development of trustworthy software that can execute in a secure system and, ideally, safely maintain the separation of information having different sensitivity levels, is a difficult task involving many stages. These stages include specification, design, implementation, integration, and validation. Software development is further complicated in SDOS by its execution in a distributed environment. The purpose of the Controlled Software Development System is to provide tools to make the task of developing software simpler while establishing high levels of assurance that software conforms to requirements for reliability and security.

Controlled software development in SDOS is based on the following principles:

- Modules of software are typed in accordance with their stage of development and information contained in the modules. Examples of modules include specification modules, source code, object code, library routines, executable modules, building commands. Typing modules defines strict interfaces to each module in accordance with how the module is used.
- Software modules can only be manipulated by tools specific to each type. It is impossible for a developer to arbitrarily modify a particular module. Rather, use of a module must correspond to the module's type. For example, object code modules can be only created, linked, or destroyed.
- Software can only progress into a new phase when the component modules have been certified to pass a criteria specifically established for the new phase. For example, a typical application is built from many different modules. The application cannot be fielded until each module has been certified to have been successfully validated. Validation may include inspection, testing, and formal validation. Certification guarantees that each module must meet specific conditions and that the definition of an application (e.g., the modules that comprise it) is consistent across each phase of development.
- All modifications to modules are audited, and version control is provided to ensure consistency between related modules in different stages. Auditing provides a means of reviewing the use of modules, while version control is useful for archiving and reconstructing previous versions and detecting incompatibilities between related modules. For example, a change to a specification can invalidate an executable module.
- Specifications that can be interpreted by development tools enhance software validation. Specification can provide a means of simply stating the properties of software. When the specifications are machine-readable, they can be used in two ways: to automatically generate code that guarantees the implementation is consistent with the specification; and to validate the consistency of the specification with the specification of other modules, thereby enforcing a system-wide policy for software.

- Software developers should be shielded from the complexity of the distributed system. One of our goals is to concentrate the software developer's energies on the development of the application, minimizing the impact of its execution in a more complex environment. A second goal is to provide support for those activities commonly performed in virtually all applications. Examples of methods of reducing the complexity of distributed application development in SDOS include:
 - Automated data type translation. Different hosts support different representations of data; for example, the representations of an integer on a Vax, C/70, and Sun are all different. Interprocess communication in such a heterogeneous environment in a host transparent fashion becomes extremely difficult when an application developer must be concerned with the destination of every message for data translation purposes. SDOS avoids this complexity by introducing canonical data type representations for data transferred between processes and automatically generating code to perform the translations.
 - Simplified model of concurrent execution. Processes in a distributed system execute in parallel, and parallel activities are difficult to understand. Routines are provided to handle the receipt of multiple messages and schedule their servicing, and to provide a simple subroutine-style interface to independently scheduled application routines.

3.1.2.10 Configuration, Monitoring and Control (CMC)

System operation entails many functions:

- **Configuration:** initialization of system parameters, movement of software between hosts, installation of software updates;
- **Monitoring:** inspection of the status of executing services and applications, including message flow, performance, and internal resource usage;
- **Control:** the initiation and termination of system services and the adjustment of the system as it executes in response to changes in load, resource usage, and administrative requirements.

System operation is made easier in SDOS than in other systems because of the use of objects as a structuring method. Configuration involves the placement of objects and object managers and the assignment of initial system parameters in the basic set of services (process management, interprocess communication, and authentication). The system is monitoring by viewing the status of object managers and their interactions. System operation involves interacting with individual managers. The CMC system is driven by an SDOS system console, used to display information to the user and service as an access point for system control.

The configuration, monitoring, and control facility in SDOS is based on the following principles:

- The facility can be applied uniformly to system services and application components. The organization of software based on objects makes it possible to interact with all object managers regardless of the function or level in the system. This uniformity greatly simplifies the CMC system.
- The status of events are collected both actively and passively in correspondence with the properties of the events. Active collection involves the CMC system initiating an action to collect information, and may be manually directed, periodic (e.g., polling), or irregular (e.g., in response to some other event). Passive collection corresponds to collection initiated by a component, such as the occurrence of an exceptional event. Two modes of collection allow the CMC system to initiate the collection of status information as well as be alerted to unexpected or emergency events.
- Logging is supported at various levels of detail. Logging can be performed within software components through the use of logging mechanisms available to software developers. Intracomponent logging is useful for highly detailed records of activity. External logging is provided by the system to monitor high-level events, such as the operations invoked on a manager. All logging may be activated or suppressed under CMC control.
- Monitoring is supported at various levels of detail. Component-based monitoring is concerned with how an individual manager performs. Service-based monitoring is concerned with how a service, possibly implemented by many different managers, performs. Host-based monitoring is concerned with monitoring the collection of services on a host. Monitoring at different levels of detail corresponds to the different levels of control used to operate the system.

3.2 Assured COS Support

The basic functions needed to support SDOS operations on a particular host, such as file system services and memory management, will be provided by various constituent operating systems (COSs). As in Cronus, each COS will be local to a particular host, and there may be several heterogeneous kinds of COS supporting SDOS.

Certain aspects of COS operation must be known to be correct in order to have assurance that SDOS enforces its security policy. The COS must provide:

- assured process separation — the ability to prevent direct inter-process communication that does not involve COS access control;
- non-interference with process operation — SDOS processes responsible for security must not be tampered with;
- stable storage — data needed for enforcing security, such as user authentication data, must be stored in a fault-tolerant manner.

Without these basic features of each COS, it may always be possible to destroy or bypass any security features of SDOS.

The COS may additionally assure a more complicated policy. For example, the COS may be a multi-level secure OS. We have favored MLS OSs which meet the DoD security evaluation criteria [DoD Criteria 85], since their security policies are more likely to support SDOS', and since they possess a high degree of assurance that their policies are met.

Given an MLS COS, the most important design question is: how can the SDOS design best use the COS functionality? This question of COS integration is addressed in the following sections.

3.2.1 Alternative Approaches to Integration

There are two basic approaches to building SDOS on top of an arbitrary MLS COS.

1. Treat SDOS as an application supported by the COS.
2. Treat SDOS as an extension of the COS kernel.

3.2.1.1 Approach I

In the first approach, SDOS is entirely responsible for enforcing multi-level security. Any MLS features the COS might possess are ignored. This approach has conceptual simplicity as an advantage, since there is no possibility that the SDOS security policy and the COS security policy can conflict.

Each SDOS process, both those that are assured to meet some policy and those that are not, will be run as a separate COS process. In particular, the multi-level SDOS kernel will execute as a single COS process. The COS will control IPC in the following manner: IPC will be possible in both directions between two SDOS processes if and only if one of those processes is the SDOS kernel. Thus, all SDOS communication will be forced to pass through the kernel, and hence through the SDOS message switch. The message switch and MLS managers will then enforce the SDOS policy as described in section 2.1.

If the COS is a multi-level secure OS, there will be two sets of security labels active: the COS labels and the SDOS labels. The MLS capability of the COS will be effectively disabled by assigning the COS label of every SDOS process to be some fixed level *X*. Then MLS access control within the COS will never fail due to mandatory security, since all COS levels are equal.

This approach has the effect of decoupling the SDOS design from the design of the COS. Not only will problems due to incompatibilities of security policies be minimized, but so will the effort needed to port the SDOS design from one COS to another. On

the other hand, with only slight coupling between SDOS and each COS, it is unlikely that efficiencies incorporated into the design of the COS can be exploited. Some extra inefficiency may even be added. For example, whatever processing power the COS expends on checking mandatory security labels will be wasted. All checks will continue to be performed, even though they must always succeed (since all levels are *X*).

It is essential to this approach to be able to set up the access controls on IPC as described above. The most likely method is to set up these controls using the COS discretionary access control mechanism and then make the COS DAC unalterable.

Note that under this approach, MLS software designed to run under the COS cannot interact directly with SDOS at levels other than *X*, unless it is first converted to the SDOS system of labels. This will be a significant drawback, especially if the COS has pre-existing MLS networking software.

3.2.1.2 Approach II

The alternative approach offers tighter coupling between SDOS and its supporting COS. In this approach, the COS is assumed to be an MLS OS. SDOS uses the MLS features of the COS to enforce mandatory security.

As in Approach I, each SDOS process is implemented as a COS process. The COS security labels are taken to be equivalent to the SDOS security labels. Direct IPC between processes is constrained by COS access controls based on the COS security labels. Thus, it is possible for processes to communicate directly without the intervention of the SDOS message switch (although they sacrifice the location transparency provided by SDOS in this case). Also, non-SDOS processes may communicate directly with SDOS processes, allowing the possibility that MLS applications which were written specifically for the COS can be integrated into the SDOS environment. This is the heterogeneous integration which is one of the prime goals of the Cronus DOS, transferred here to SDOS.

Implementing Approach II will be more complicated than Approach I, as there are a several problems which must be overcome.

- SDOS must use the COS system of security labels. This will be a problem, for example, if there are not enough bits in each COS level to record the information in an SDOS level. The number of bits in SDOS levels has not been specified in this report, but it must be sufficient to include both security and integrity classifications and categories. The COS labels must also include the possibility that an entity is multi-level, and possibly to allow some sets of levels to be authorized for multi-level entities.

By not specifying the structure of its labels in complete detail, the SDOS scheme remains very general. It can probably be restricted to conform to whatever labeling scheme is used on most MLS COSs. There is the risk, however, that in an SDOS supported by heterogeneous COSs, that a common denominator would need to be

found for the labeling on all the different COSs, and that it would be unnecessarily constraining.

- Not only must the COS labels distinguish between multi-level and single-level entities, but the mechanism used to create new MLS entities must be flexible and available for use by the SDOS kernel. SDOS must have the capability to start new MLS managers, which will require that multi-level COS processes be created interactively. An inflexible COS design, allowing only a fixed set of multi-level processes, known in advance, would not be able to meet this requirement. Nor would a COS design in which MLS processes could be started only by system administrators.
- This approach uses the COS system of labels, and it therefore uses a COS "security database". In order that the SDOS kernel be able to make access control decisions based on labels, it must be able to read this COS "security database". Therefore, COS operations to permit this must exist.

The SDOS security database contains information other than labels, replication counts, for instance. For this approach, the SDOS design must partition the SDB into a COS part and an SDOS part.

- Other aspects of the COS and SDOS security policies may be incompatible. In particular, the SDOS configuration policy, which defines the unchangeable system security preconfiguration (see 2.1.3.4), may bear no relation to the policy of the COS. For example, the SDOS security preconfiguration may require that labels be non-modifiable, but this cannot be enforced in Approach II unless it is enforced by the COS.

The configuration policy also requires SDOS to keep the security configuration consistent across the network. The COS, acting purely locally, will meet no such policy. Therefore, COS operations must exist to support SDOS in implementing this consistency.

3.2.2 Examples of SDOS Integration with an MLS COS

As examples of how the above integration approaches might be implemented given the current COS technology, the following sections describe the considerations to be taken into account when hosting SDOS on GEMSOS and KeyKOS.

The GEMSOS secure operating system is a product of Gemini Computers of Carmel, California. It is an existing product which is designed to meet the NCSC class B3 evaluation criteria. It runs on the 80286 processor, and exploits that architecture's division of privilege into layers, called *rings*.

The KeyKOS operating system is an existing product of Key Logic, Inc., of Santa Clara, CA. It defines an object-oriented structure of resources in which access control is capability-based. It currently runs on IBM 370 architecture computers.

3.2.2.1 Hosting SDOS on GEMSOS

Processes in GEMSOS will ordinarily be able to communicate with each other by using shared memory structures called *segments*, *event counts*, and *sequencers*. Segments are blocks of memory which can be read and written once visible in a process' address space. Associated with each segment is an event count and a sequencer. These are registers which allow secure synchronization using the scheme of Reed and Kanodia [Reed and Kanodia 79]. Each association of a segment, event count, and sequencer is assigned a security level. A process may use the association for communication once the GEMSOS security kernel has successfully mediated its request for access. Two processes with access to the same association can communicate directly without further intervention by the kernel.

The GEMSOS ring 0 (the ring with greatest privilege) contains all software which is directly involved in enforcing a mandatory policy based on levels. This includes:

- memory management, including mediation to decide whether particular segments, event counters, and sequencers will be made accessible to particular processes;
- any multi-level device-drivers. Such drivers include disk drivers which ensure multi-level access to stable storage on disk, as well as software to run transport-layer protocols on multi-level network links.

The GEMSOS ring 1 contains ancillary software for MLS, such as authentication routines, and also software to enforce discretionary access controls.

We expect that either Approach I or Approach II can be made to work with GEMSOS. Bidirectional IPC is possible for a pair of processes at the same GEMSOS level by creating a pair of segments, with associated event counts. For Approach I, the ability of a process other than the SDOS kernel to create new shared segments must be eliminated. For Approach II, the SDOS kernel and other MLS managers must be declared to be multi-level GEMSOS processes. GEMSOS uses security labels which include a range of levels for multi-level entities. This will be sufficient if SDOS level sets are taken to be ranges also. In either case, the SDOS kernel, MLS managers, and trusted interface processes can be made to execute in ring 1.

3.2.2.2 Hosting SDOS on KeyKOS

The KeyKOS kernel defines a set of *objects*. These objects may contain both programs and data. Communication between objects is via invocation of abstract operations, and this communication is uniform whether the objects are processes or data, and whether kept in volatile memory or on stable storage. The distinction between objects on stable storage and objects in volatile memory is hidden, with the system providing periodic checkpointing services which aid restart/recovery.

Use of an abstract operation requires possession of a capability for that operation. The capabilities, called *keys*, are maintained in software by the KeyKOS kernel. Keys can be used by objects but cannot be manipulated directly, and hence cannot be forged.

KeyKOS presents a very minimal functional interface, emphasizing efficiency. It is claimed that services which are missing from this minimal structure, such as multi-level security and auditing, can easily be added. The approach recommended in the KeyKOS documentation is to divide the keys into disjoint sets, each set possessed only by objects at a single level and authorizing operations only on objects at that level. This completely isolates each security level. To support vital operations, such as "read down", the documentation recommends creating multi-level "filters", which allow information to flow between levels, but only securely. Such filters have apparently not been built for the KeyKOS environment.

The collection of the SDOS kernel and MLS managers is, in fact, such a filter. Therefore, it is possible to use Approach II above, in which each single-level application written for KeyKOS is given only keys for operations on objects at some particular level. Any multi-level application will additionally require keys for communication with the SDOS kernel, which possesses capabilities for all levels. Thus, indirect communication across levels is possible.

It is also possible to use Approach I above, in which an initial allocation of keys permits each KeyKOS object which is not the SDOS kernel to communicate only with the object representing the SDOS kernel. Then multi-level security can be enforced by the SDOS design. This allocation of keys is simpler, but more restrictive, since it prevents integration of KeyKOS applications into SDOS. Approach II should be favored instead.

3.3 Communications Layering

Even though the DoD protocols are used to support SDOS networking needs, general discussions of telecommunications networking are centered around the architecture of the *Reference Model for Open Systems Interconnection* (or OSI Reference Model). This model was developed for distributed information systems and the functionality needed to communicate between heterogeneous hosts. After being approved by the International Organization for Standardization (ISO) and by the International Telephone and Telegraph Consultative Committee (CCITT), OSI became a standard in 1983. The next few sections will examine OSI and then relate it to the protocols used in SDOS.

3.3.1 OSI Reference Model

A succession of functions is involved in a communication. The technique of layering employed by OSI is to organize those functions in a logical order. OSI provides the framework for the definition of standardized procedures for information exchange. These

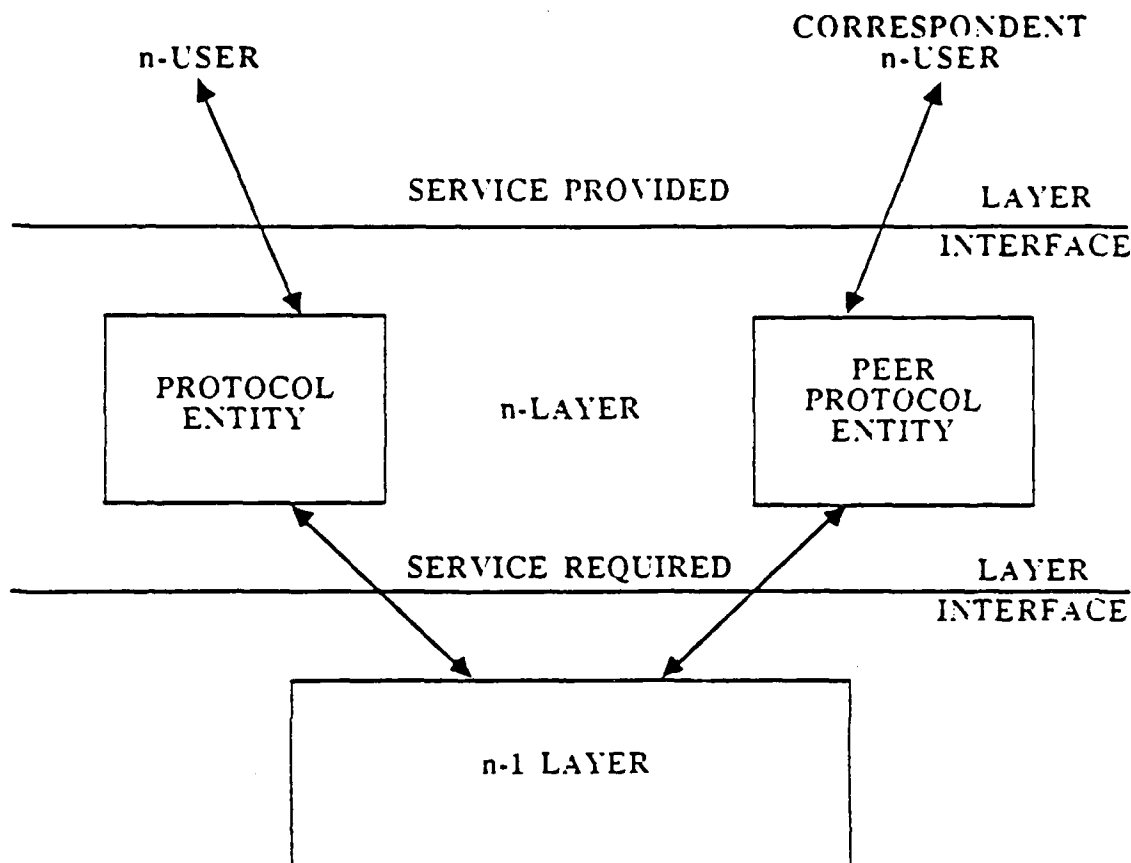


Figure 3.1: Layer Service Hierarchy

are the protocols developed at each layer. OSI is also technology independent—the standardized procedures are independent of the hardware and software chosen to implement them. Each protocol layer (layer "n") provides a set of *services* (functions) to its user in the next higher layer (layer "n+1") by building upon the services provided to it by the next lower layer (layer "n-1"). Layers are defined with similar services grouped together and interactions across layer boundaries minimized. The grouping of services within a layer is called an *entity*. In addition, layers are defined so that different protocols may be used within a layer without affecting the layer service definition. This is sometimes called *layer independence*. Layer independence means that the implementation details of a layer are considered hidden from its users. This makes it possible, for example, for the same transport protocol to be used on packet-switched as well as circuit-switched networks.

Entities in the same layer communicate with each other using peer protocols. Figure 3.1 shows the abstraction of the idea of one layer building on another to provide services to a user. More specifically, the figure illustrates the idea of service hierarchy and shows the relationship of two correspondent n-layer users and their associated n-layer peer protocol entities.

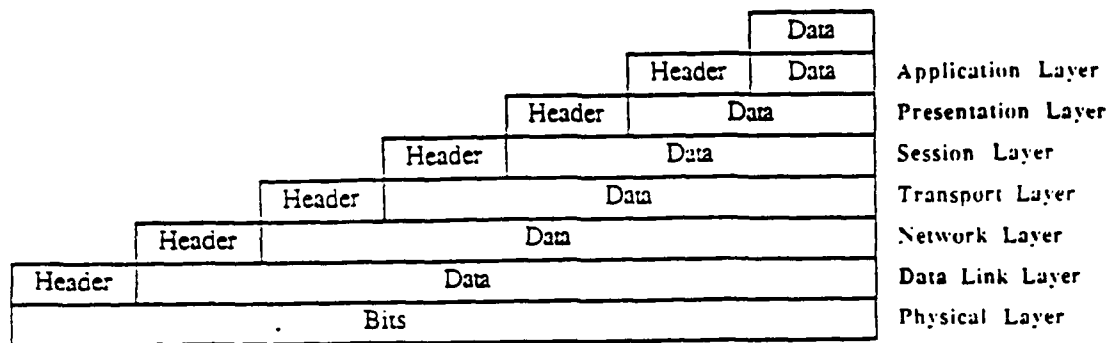


Figure 3.2: PDU Construction

Information transfer between peer entities takes place with protocol data units (PDUs) composed of control information and data; and information transfer between adjacent protocol layers takes place with service data units (SDUs). At each layer a protocol header conveying the peer protocol information for that layer is added to the SDU from the " $n+1$ " layer. This in turn, becomes the SDU to the " $n-1$ " layer. The appending of protocol control information onto the data units follows layer by layer as the outgoing PDU is constructed. At the recipient end system, each layer " n " in turn looks at its header, performs the required function and passes the data unit minus header " n " to the " $n-1$ " layer. Figure 3.2 shows the construction of the outgoing PDU.

Abstract *service primitives* are the interactions that take place at a layer boundary between the provider of a layer's services (*service provider*) and the user of a layer's services (*service user*). There are four types of service primitives: requests, indications, responses, and confirmations. A service user invokes a service by issuing a *request*. A service provider gives an event or condition to the service user by issuing an *indication*. A service user responds to the event or condition by issuing a *response*. Finally, the service provider issues a *confirmation* to the service user to give the result of the service request. See Figure 3.3.

Figure 3.4 shows the OSI Reference Model. Seven protocol layers are defined—grouping logically related services together. The upper three layers provide services in direct support of the application process, while the lower three layers are concerned with the transmission of the information between the end users of the communication. The Transport Layer is the essential link between these two groups of services; it provides end-to-end integrity of the communication, ensuring that the appropriate quality of service from the lower three layers meets the requirements of the upper three layers.

1. **Physical Layer.** The Physical Layer standardizes the physical medium and the signalling techniques used across that medium.
2. **Data Link Layer.** This layer synchronizes the transfer of information over the physical link.

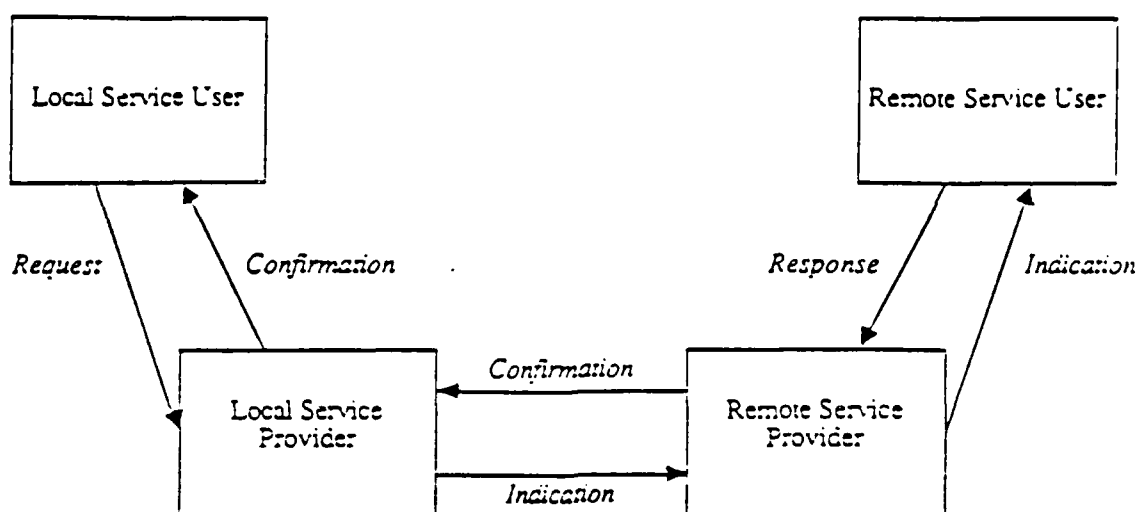


Figure 3.3: Confirmed OSI Service

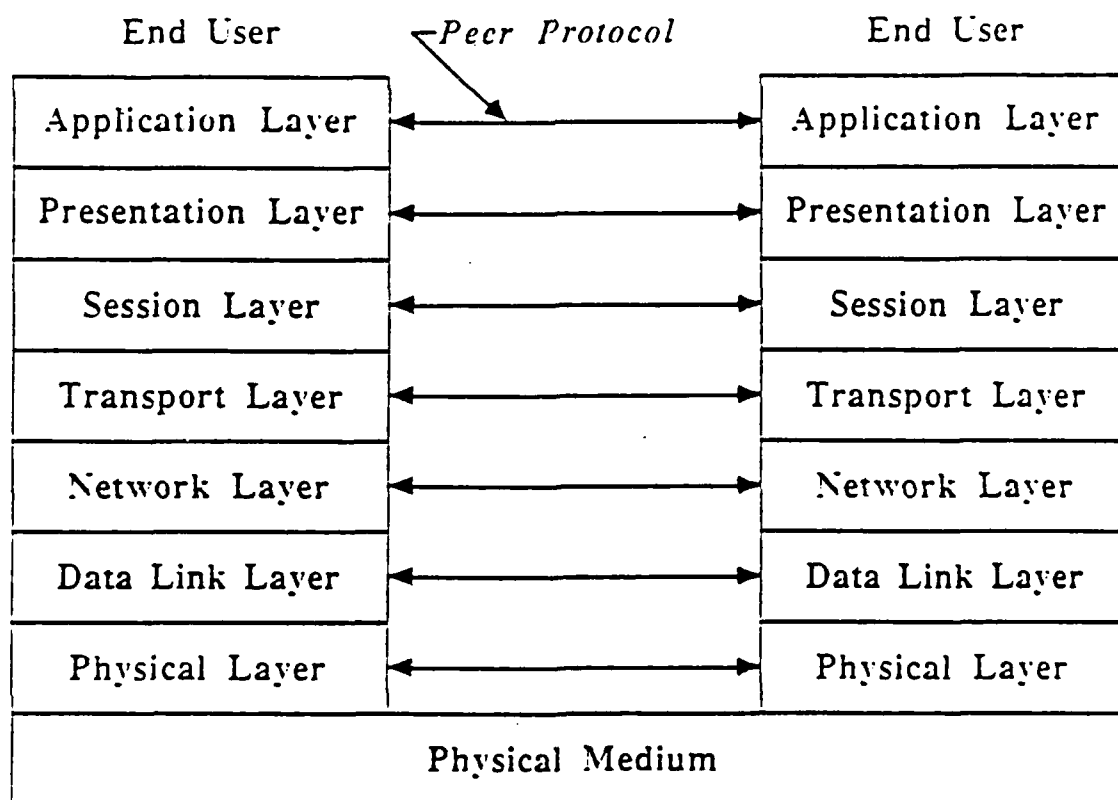


Figure 3.4: OSI Reference Model

3. Network Layer. This layer provides the switching and routing functions needed to establish, maintain, and terminate switched connections and to transfer data between the communicating end-users. It is the lowest layer providing a host to host service.
4. Transport Layer. This layer provides transmission control of the information interchange.
5. Session Layer. This layer maintains the conversation between cooperating users.
6. Presentation Layer. The representation of the data is managed at this level.
7. Application Layer. The Application Layer provides the user direct access to the OSI environment and to the distributed information services to support the user and to manage the communication (e.g., distributed transaction management, terminal emulation, file transfer, and mail transfer).

3.3.2 SDOS Protocol Hierarchy

Figure (3.5) relates SDOS to OSI. The upper three layers, Application Layer, Presentation Layer, and Session Layer services are provided by the *SDOS-specific Layers* of SDOS. For SDOS the Application Layer comprises client software and object managers. The Presentation Layer in SDOS corresponds to the canonical datatype representation layer of Cronus, where Application Layer messages are encoded into machine independent representation. Session Layer services are provided by the Inter-process Communication (IPC) Layer of SDOS where the Message Switch, Locator, and Process Manager manage the organized and synchronized exchange of messages (see Sections 3.5 and 3.6 for a complete discussion of these components of the SDOS kernel).

The lower four layers, the *communications substrate*, correspond to the U.S. Department of Defense (DoD) Internet Protocol Suite (specifically TCP, UDP and IP) over Ethernets and the ARPANET. The Transport Layer services are provided by the Transmission Control Protocol (TCP) when reliable message transport is required or by the User Datagram Protocol (UDP) when short, unreliable message transport is appropriate. The Network Layer services are provided by the Internet protocol (IP) along with protocols specific to the communications subnet employed. And the Data Link Layer and Physical Layer services within the host interface to an Ethernet or the ARPANET. The Internet Protocol Suite was originally developed for the ARPANET and then adopted as standards by the DoD.

3.4 Network Security

Network security complements the security mechanisms implemented above the communications substrate within individual SDOS hosts (SDOS-specific security) by protecting

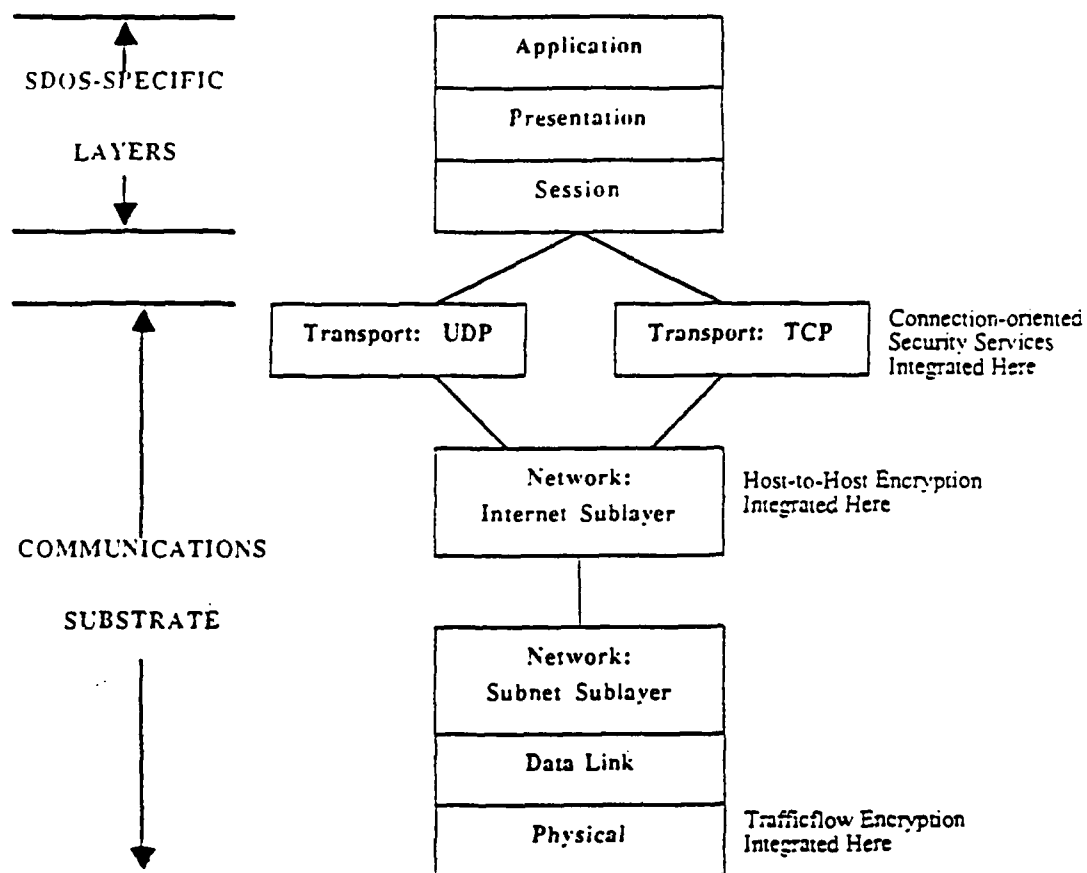


Figure 3.5: SDOS and the OSI Reference Model

the communication channels connecting the hosts of the distributed operating system. Adding security functions to the communication channels guards against passive and active wire-tapping and (optionally) against the unauthorized disclosure of message traffic patterns. SDOS-specific security is also enhanced because the data transfers within the distributed operating system are protected. Adding security functions to the data transfers protects message content from unauthorized disclosure, ensures that messages arrive intact, and verifies the identities of the communicating systems.

Applying security functions to the communications substrate serves to segregate hosts of different trust or of different accreditation levels. Connections and messages between heterogeneous hosts can be audited and interaction limited.

For this work we looked at:

- ISO 7498, The Basic Reference Model for Open Systems Interconnection,
- ISO 7498 Part 2, Security Architecture, and
- Trusted Network Interpretation of the Trusted Computer Systems Evaluation Criteria [NCSC TNI 87].

There are two main parts to this discussion. The first two sections are tutorial; they detail security services and the security mechanisms to apply those services to the network model. The third section applies the knowledge of the tutorials to SDOS.

3.4.1 Security Services and Mechanisms

This section considers security services and the security mechanisms with which to provide those services. The problem in SDOS is which of these services are needed for SDOS and where to apply them (at what layer of OSI). Security services are functional characteristics (e.g. authentication), and are distinct from the mechanisms which implement the services.

This discussion emphasizes services which can be provided between peer distributed operating system hosts on an end-to-end basis. These services are independent of any security-related services which may be provided within the intervening communications facilities. The Network Layer (layer 3) is the lowest layer with end-to-end significance between communicating hosts, therefore all services considered in this discussion will be provided at the network layer or higher. It is possible that the set of security requirements needed by a distributed operating system (e.g., traffic flow confidentiality) may necessitate link-oriented mechanisms in addition to end-to-end mechanisms.

Security services can be provided at a range of OSI protocol layers. In general, as services are placed at lower layers, protection is host-oriented or link-oriented (e.g., *coarse service granularity*). There is also the potential for integrating the security services as a separate hardware component on the communication path between the host and

the network. When security services are placed at higher layers, protection is provided on a per-user (or per-client) basis (e.g., *fine service granularity*). On a per-user basis additional security services become appropriate. However, this requires more invasive integration into the communicating hosts and their software, and raises the computer security concerns that comes with that integration.

Not all security services are appropriate or feasible in all contexts, and it is necessary to distinguish those services that can be encapsulated within the communications substrate from those that must be provided (wholly or in part) within SDOS-specific host TCB components. The selection of appropriate security services depends on several parameters, including the protocol layer positioning of peer entities between which traffic is to be protected. For example, a host processing electronic mail incorporates protocol peer entities spanning the OSI seven-layer range. But the set of security services appropriate to protection of a host at the network layer is significantly different from the set of security services appropriate to protection of an electronic mail user agent at the application layer.

Placing security functions at lower layers or on the transport medium limits the amount of data accessible to a wiretapper, because placing security service functions higher in the protocol hierarchy passes more unencrypted header information. Though security functions at a given layer rely on the unencrypted header information (i.e., control information) that is passed down from higher layers, the security functions need not interpret the data passed with the headers. See Figure 3.2: PDU Construction.

Placed at lower layers, security functions can also limit the covert channel bandwidth from the higher layers into the transport medium, but only if all paths from the higher layers into the transport medium pass through the security mechanisms.

Independent of the layer at which security service functions are placed, they cannot guarantee that higher layers function correctly. The primary purpose of most security functions is to assure the security characteristics required by their clients, though certain services, such as access control, act to restrict or filter the actions of their clients.

The five OSI security services are discussed in the following section. *Authentication* and *access control* are relevant security issues whether or not an environment is distributed and have already been examined in their non-networking role. Provided in the networking substrate, they further enhance this role. *Data confidentiality* and *data integrity* are integral services to the message passing involved in a distributed operating system. The fifth OSI security service, non-repudiation, is offered on a per-user rather than a per-host basis. A possible sixth security service, *communication availability*, is also discussed.

3.4.1.1 Security Services

3.4.1.1.1 Data Confidentiality When considering the concept of network security, *data confidentiality* is traditionally the first service which comes to mind. This is

especially true in the DoD classified environment, where protection of classified information from unauthorized disclosure is of overriding importance relative to other security services. Data confidentiality is important in the unclassified environment as well, although in certain commercial contexts (e.g., EFT transfers) its importance is secondary to authentication and integrity services.

The primary data confidentiality requirement in all environments is protection of user data from disclosure. In the unclassified environment, the relevant concern (discretionary confidentiality) can be defined as disclosure to an entity other than the data's intended recipient. The DoD classified environment adds another concern (mandatory confidentiality), that assumes primary importance: disclosure of classified data to an entity which lacks sufficient clearance to receive it. Both types of confidentiality service can be provided usefully at a range of OSI protocol layers.

In the DoD classified environment, traffic flow security is generally required in addition to user data confidentiality. The preferred approach to traffic flow security requires physical layer link encryption, independent of any end-to-end encryption. Traffic flow security measures at higher protocol layers offer limited protection and can be excessively costly with respect to network bandwidth.

3.4.1.1.2 Authentication

3.4.1.1.2.1 Data Origin Authentication The *data origin authentication* service provides a recipient OSI peer entity with assurance that associated data originated at the claimed source peer entity. The service is useful at a range of OSI protocol layers, affording finer authentication granularity (e.g., per-user instead of per-host) when applied at higher layers.

In general, the data origin authentication service offers significant value only if a data integrity service which ensures that the received contents have not been modified in transit is also provided. There is limited utility in knowing that a message arrived from a particular source unless the recipient can also be assured that the contents are the same as those which were provided by the authenticated source entity. Fortunately, the mechanisms used to provide data origin authentication are closely related to those used to provide per-message data integrity.

3.4.1.1.2.2 Peer Entity Authentication The *peer entity authentication* service confirms the identities of the connected peers, either at the connection establishment phase or during the data transfer phase of the connection. Bidirectional assurance of timeliness is provided. The peer entity authentication service can be provided only in the context of a connection oriented communication service.

The peer entity authentication service is not applicable to connectionless communications or to contexts in which connections are terminated and regenerated at sites intermediate to endpoint systems. For example, while relevant to a virtual circuit between

a pair of X.25 peer entities, peer entity authentication is inapplicable to an electronic mail transfer in which a piece of mail is staged or relayed at intermediate points between the application layer entities which process mail on behalf of users.

The peer entity authentication service is closely associated with host-based connection oriented protocols. And while the data origin authentication and per-message data integrity services are not necessarily host-based, they can aid a host in offering support of this service.

3.4.1.1.3 Data Integrity ISO 7498 Part 2, Security Architecture subdivides *data integrity* services into several categories. Protection of an individual message's contents from undetected modification, applied either to an entire message or to selected fields within a message, is a basic level of data integrity service. In a connection oriented environment, it is meaningful to offer additional data integrity services dealing with the stream of messages transferred on a connection. These additional services are dependent on the per-message data integrity services, and apply distinct mechanisms in order to protect against undetected message reordering, loss, replay, and spurious insertion of messages into a protected connection.

Data integrity services at the per-message level (*per-message data integrity*) are appropriate at a wide range of OSI protocol layers. The per-message data integrity service is closely associated with the data origin authentication service; both services are likely to be based on shared cryptographic mechanisms. These services can be provided wholly within a communications substrate, or the cryptography provided by the substrate supporting a confidentiality service can aid SDOS-specific components in performing authentication and message integrity checks.

Message stream integrity services are based commonly on host-resident protocol sequencing mechanisms. Provision of these mechanisms as communications substrate security services that are not necessarily host-based raises significant integration and performance issues. If a non-host-based communications substrate supplies host software with a stream of authenticated messages, individually verified for integrity, it may be reasonable for message stream integrity measures to be provided within existing host-based protocols.

3.4.1.1.4 Access Control *Access control* is needed to enforce a distributed system's security policy and to maintain the overall distributed system integrity.

In the OSI model, the access control service provides protection against unauthorized use of OSI-accessible resources. This service can be provided on a group basis (e.g., X.25 closed user groups) or on an individual entity basis. The access control service controls resource usage based on the authenticated identity of a would-be accessor, and hence is closely associated with (although distinct from) the authentication service which provides it with necessary identification data. Like the authentication service, the granularity of available access control service becomes finer if the service is provided at

higher OSI protocol layers.

In the DoD classified environment, the primary access control requirement is mandatory, based on a defined lattice representing information sensitivity as a combination of hierarchic classification level and associated category designation. Although discretionary access control is also required in the DoD classified environment, it assumes importance secondary to that of mandatory access control. In the unclassified environment, in contrast, no analogous information sensitivity lattice exists. As a result, a strictly discretionary access control service is likely to be appropriate in this environment.

In a distributed environment where all SDOS TCB components are mutually trusting and operate at the same access class, their internal access control mechanisms may render communications substrate access control functionality superfluous. Intra-substrate access control could, however, be useful in establishing "firewalls" in front of any less trusted hosts or hosts which operate at different access class ranges.

3.4.1.1.5 Non-Repudiation The *non-repudiation* services are subdivided into two subcategories: non-repudiation, origin (prevention of a sender's falsely denying that a message was sent) and non-repudiation, receipt (prevention of a recipient's falsely denying that a message was received). In general, these services are associated with individual human users, rather than hosts, and hence are inappropriate at protocol layers below those at which individual users are distinguished. For example, all non-repudiation services may be provided at the Application Layer. Since this layer is implemented within SDOS, it may not be appropriate to offer a non-repudiation service within the enhanced communications substrate.

3.4.1.1.6 Communications Availability *Communications availability* assurance (protection against denial of service) may be an important issue, but is not addressed in ISO 7498 Part 2, Security Architecture. Should communications availability be viewed as a security requirement, its assurance must be addressed by mechanisms within the interconnecting communications facilities themselves. It is possible for endpoint modules to detect certain types of communications service loss, particularly in a connection oriented environment, but there is no way for endpoint modules to enhance the level of communications availability provided by an intervening network. (It is assumed that the endpoints are not provided with an alternate communications path over which messages can be redirected in the event of a detected denial of service condition on a primary path.)

3.4.1.2 Security Mechanisms

Figure 3.6 shows the classes of security mechanisms that are provided by the security services (with the exception of communications availability) discussed in the previous section.

MECHANISM SERVICE	Encipher- ment	Digital Signa- ture	Access Control	Data Integrity	Authenti- cation Exchange	Traffic Padding	Routing Control	Notari- zation
Data Confidentiality								
User Data	X						X	
Traffic Flow	X					X	X	
Data Integrity								
Per-message	X	X		X				
Message Stream	X			X				
Authentication								
Data Origin	X	X						
Peer Entity	X	X			X			
Access Control			X					
Non-Repudiation		X		X				X

Figure 3.6: Security Services versus Security Mechanisms

3.4.1.2.1 Encipherment Encipherment, or encryption, can be used to encrypt the entire message stream, individual messages, the data within a message, or data fields within a message. It complements or plays a role in many of the other mechanisms discussed.

3.4.1.2.2 Digital Signature Mechanisms The digital signature mechanism is an authentication technique, and as such can be used to provide the authentication security service and the non-repudiation security service. It assures per-message integrity as well. The signature is the result of a signer's private encipherment key applied to the data unit.

3.4.1.2.3 Access Control Mechanisms Access control mechanisms may be based on the distribution of cryptographic keys, on the contents of access control information bases, or on an attempted accessor's possession of authentication information (e.g., passwords) or capabilities. In some cases, access control mechanisms also depend on the contents of security labels associated with entities or with data units.

3.4.1.2.4 Data Integrity Mechanisms Enciphered, or non-enciphered block check codes, cryptographic checkvalues, time stamping or cryptographic chaining can be used to protect data integrity. The choice of mechanisms used depends on which form of data integrity, message-stream integrity or per-message integrity, is needed. It also depends on the strength of the data integrity service that is required .

3.4.1.2.5 Authentication Mechanisms Passwords, cryptography, time stamping and synchronized clocks, two- and three-way handshakes, and digital signatures used in various combinations with one another are available choices to provide the authentication service.

3.4.1.2.6 Traffic Padding Mechanisms Physical layer encipherment is the preferred mechanism to support a traffic flow confidentiality service, and is sufficient in itself to provide that service, but traffic padding mechanisms (if complemented with a confidentiality service at or below the layer where the padding mechanisms are implemented) can be used to provide various levels of traffic flow confidentiality.

3.4.1.2.7 Routing Control Mechanisms Data routes through the network can be chosen dynamically or statically based on the security labels of the data or knowledge of the security of the individual networks.

3.4.2 SDOS Network Security Approach

3.4.2.1 SDOS Network Security

For SDOS, host-to-host network security can be provided by integrating encryption-based security in or near the Internet Protocol (IP) sublayer of the Network Layer (see figure 3.5). This layer represents the lowest place in the OSI Reference Model where end-to-end encryption can be done across the internet, and also limits the invasiveness of communications substrate security integration into SDOS hosts. Encryption here serves both the TCP and UDP traffic in a uniform fashion. Integration of security mechanisms at higher layers passes more unencrypted control information in the message headers (see figure 3.5), and would require individualized mechanisms for each of the higher layer protocols. This would increase the security integration effort.

Encrypting messages at the IP sublayer provides:

- protection of information from unauthorized disclosure (data confidentiality),
- a recipient assurance that associated data originated at the claimed source (data origin authentication),
- protection of an individual message's contents from undetected modification (per-message integrity), and
- protection against unauthorized use of those resources accessible through SDOS (access control).

For the case of TCP's sequenced traffic, placement of encryption and associated security functions at the IP sublayer cannot provide stream oriented assurance. Instead, the IP sublayer's security features provide the per-message secure basis on which trusted TCP modules can build a secure stream-oriented communication service. Note that no secure stream-oriented service is offered for UDP traffic.

Service granularity is limited with the IP sublayer placement, e.g., the access control function cannot distinguish one application service from another (TCP from UDP); therefore inter-service or inter-user segregation is handled above the communications substrate by the trusted functions of the SDOS-specific layers.

3.4.2.2 Separate Versus Embedded Security Module

Traditionally, communication security modules are devices separate from the hosts they are protecting, but the traditional host is untrusted. Given that SDOS is composed of a set of trusted hosts, use of embedded communication security modules is suggested as a flexible and cost-effective approach.

3.4.2.3 Traffic Flow Confidentiality

The traffic flow confidentiality security service cannot be offered at the IP sublayer, though it can be integrated lower within the communications substrate. Traffic flow confidentiality can protect against passive wiretapping by non-SDOS hosts, though this protection should be provided in a way independent of any SDOS security mechanisms, i.e. in a network specific way. As deployment of a heterogeneous distributed system is more easily achieved if the host components can be connected to an arbitrary communications network rather than requiring special characteristics in the underlying medium, it becomes important to keep the use of physical layer traffic flow confidentiality mechanisms separate and decoupled from the remainder of the network security architecture.

3.4.2.4 TCB Components

For SDOS the Application Layer down through all or part of the Internet sublayer are in the TCB; layers below the Internet sublayer are not in the TCB. As encryption is integrated into the Internet sublayer, the layers below the encryption-providing layer have no relevant security function (except with regard to traffic flow confidentiality, as discussed in Section 3.4.2.3). Some hosts may have these layers within the TCB while others may not; the security or the functioning of SDOS is not affected.

The lower layers (Network: Subnet, Data Link, and Physical) should have the following properties:

1. It is important that the lower three layers be the only path by which messages can reach network resources.
2. The lower layers should be isolated from other modules within the system preventing unauthorized data flows across the encryption boundary.

3.4.2.5 Relationship With The Trusted Network Interpretation Document

According to the draft *Trusted Network Interpretation* (TNI), it is acceptable to decompose the implementation of Discretionary Access Control, Identification/Authentication, Audit, and Mandatory Access Control mechanisms among multiple mutually trusting components. It is not reasonable to decompose implementation of functions among components which are not mutually trusting. In the case of SDOS, the distributed system functions are decomposed among a cooperating set of SDOS computers.

The lowest assurance level in any component supporting a decomposed mechanism sets an upper bound on the assurance level which can be claimed for that mechanism. While it is not necessary from the TNI viewpoint that each component meet the full complement of Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) (DOD- 5200.28STD) standards at a particular level, it is necessary for each

component to meet the TCSEC assurance requirements in addition to correctly performing its role in the TNI- evaluable network-level distributed functions which are allocated to the component.

3.5 Descriptive Top-Level Specification

This section describes the portions of the programmer interface to SDOS that relate to the security of the system. Since SDOS is based on the Cronus DOS, operations which do not relate to security are unchanged from Cronus and can be found in the Cronus User's Manual [Cronus 88].

The SDOS services visible to a developer are the kernel and a set of managers. The SDOS programmer interface is divided into two parts: a set of system calls supported by the *Message Switch* component of the SDOS kernel, and a set of operations defined by various types. The types that are described in this section include:

- **host:** corresponds to the kernel on the local host;
- **process:** the object that is associated with an executing program;
- **principal:** used to associate an identity and label with each process; the principal is used for discretionary access control.
- **project:**, associated with principals, projects are also used for discretionary access control;
- **object:** this type is the supertype of all other types; operations defined on it are inheritable by all other types.

Note that the target object of an invocation is always an implicit parameter to all operations.

3.5.1 Message Switch System Calls

SDOS supports the multi-level security policy based on message passing, as described in Section 2.1.2. The purpose of multi-level security is to prevent the unauthorized access of information by users of the system. This is achieved by classifying information in accordance with its sensitivity; tagging each user and device with a security level clearance; and denying users having inadequate clearance to access sensitive information.

The SDOS kernel provides three systems calls. All other services provided by the SDOS kernel and other system services are provided through the use of these systems calls. The system calls provided by the SDOS kernel are:

- **Invoke:** used to invoke an operation on an object. Causes a message to be transmitted to the process managing the object. If the location of the object is not known, the Message Switch delivers the invocation request to the *Locator* kernel component, which is responsible for determining the location of the object. **Invoke** may be called by any process. The parameters to **Invoke** are:
 - *host* (input parameter; hereafter abbreviated *in*): location of the object (optional);
 - *object* (in): object on which operation is invoked;
 - *label* (in): label of the message being invoked (optional; if not included, the message is labeled with the minimum label of the process);
 - *message* (in): buffer containing the invocation request, including the operation;
 - *messagelength* (in): length of the buffer;
 - *sendflags* (in): flags indicating message delivery requirements;
 - *writeup* (in): a flag indicating whether the operation is a write up;
 - *messagehandle* (output parameter; hereafter abbreviated *out*): unique identifier of the message transaction.

An **Invoke** can fail for security reasons if the invoker sets an inappropriate security label, or if the Message Switch is unable to deliver the message without violating the security policy.

- **Reply:** used to reply to an invocation request; it is callable only by a manager. Causes a reply message to be transmitted to the client process that previously invoked an operation on an object managed by the calling process. The parameters to **Reply** are:
 - *host* (in): location of the client process;
 - *process* (in): client process object;
 - *label* (in): label of the reply message (optional; if not included, the message is labeled with the minimum label of the process);
 - *message* (in): buffer containing the reply;
 - *messagelength* (in): length of the buffer;
 - *sendflags* (in): flags indicating message delivery requirements;
 - *messagehandle* (out): unique identifier of the message transaction.

Reply will fail for security reasons if the Message Switch is unable to deliver the message without violating the security policy.

- **Receive:** used to receive the manager reply to an invocation request; it is callable by any process. **Receives** returns a message if one is awaiting delivery; otherwise, the calling process is either blocked or the call fails (depending on the flags select by the caller). The parameters to **Receive** are:

- *descriptor* (in): a descriptor used by the kernel to match the reply with the original invoke;
- *message* (in): address of buffer in which to place the reply;
- *messagelength* (in): length of the buffer;
- *timeout* (in): length of time to wait if not message is queued for delivery;
- *Blockflag* (in): indicates whether the process should block if no message is queued for delivery;
- *messagehandle* (out): unique identifier of the message transaction.

3.5.2 Type Host

An object of type host represents the SDOS kernel on a particular host. Operations on a host object are grouped into three categories: operations on the Security Database on the host, operations on the Object Database on the host, and operations to monitor and control the execution of SDOS on the host. Each of these is considered separately.

3.5.2.1 Security Database

The Security Database is a component of the kernel that is responsible for maintaining the security labels for all objects on the host where the kernel executes. The Security Database provides a set of routines to clients for accessing the security labels. These routines may be invoked from inside the kernel by the Object Database and the Message Switch; or outside the kernel by the System Manager user (who is responsible for setting the security labels of objects) and remote kernels. When one of the routines is invoked from outside the kernel using the object-oriented invocation protocol, the target object of the invocation is the host object on which the security database resides.

The following operations are defined by the Security Database:

- **CreateSDBEntry**: used to create an entry for a new object. This operation is invoked by the Object Database as part of the **CreateODBObject** operation for new object instances or by the Configuration Manager as part of configuring new types or hosts into the system. In the case of a newly created object instance, the security label of the object to be created in the entry must be dominated by the security label of the object's type. The parameters to **CreateSDBEntry** are:
 - *object* (in): object identifier;
 - *label* (in): label of the object;
 - *type* (in): type of the object;
- **RemoveSDBEntry**: deletes an entry for an object being removed from this host or destroyed. This operation is invoked by the Object Database as part of the

RemoveODBObject or by the Configuration Manager as part of changing the system configuration. The parameters to **RemoveSDBEntry** are:

- *object* (in): object for which entry is to be removed.
- **ModifySDBEntry**: used to modify the attributes of an object. This operation can be invoked by the System Manager. The parameters to **ModifySDBEntry** are:
 - *object* (in): object for which label is to be modified;
 - *label* (in): new label;
- **ReadSDBEntry**: used to return the Security Database entry of an object. This operation is invoked by the Message Switch and the Object Database when checking the existence and security label of objects. The parameters to **ReadSDBEntry** are:
 - *object* (in): object for which entry is desired;
 - *SDBentry* (out): entry being read.
- **ReplicateSDBEntry**: used to make an SDB entry for an object that exists on another host. This operation is invoked by the Object Database. The SDB is responsible for reading the label of the object being replicated from the remote SDB, and for informing other SDB's of the new location of the label. The parameters to **ReplicateSDBEntry** are:
 - *object* (in): object for which entry is desired;
 - *host* (in): host where object currently resides.
- **DereplicateSDBEntry**: used to remove the label of the object from the local SDB even though it exists elsewhere. This operation is invoked by the Object Database. The parameters to **DereplicateSDBEntry** are:
 - *object* (in): object for which entry is to be removed;

3.5.2.2 Object Database

The Object Database is a component of the kernel that is responsible for maintaining the representation of abstract objects on secondary storage on the host where the kernel executes. The Object Database provides a set of routines to object managers to create, retrieve, update, replicate and dereplicate objects. The Object Database ensures these accesses conform to the mandatory security policy. When one of the routines is invoked from outside the kernel using the object-oriented invocation protocol, the target object of the invocation is the host object on which the security database resides.

The Object Database enforces several properties on the objects it stores. First, the Object Database contains an entry for each object managed on the host. Second,

it ensures that a manager may only access objects of types that it is responsible for managing. Third, it ensures that the Security Database entries for a replicated object are all consistent with one another. And last, the Object Database ensures that the accuracy of the list of hosts where an object resides is properly maintained. This list indicates the hosts where copies of the object reside if the object is replicated.

The following operations are defined on the Object Database:

- **CreateODBObject:** This operation creates an object and invokes the **CreateSDBEntry** operation to create an entry for the object in the Security Database. Only a manager for the object's type may invoke this operation. The security label of the object is extracted from the message delivering the invocation request. If the security label of the object is not equal to but dominates the security label of the invoker (i.e., it is a *write up* operation), then the operation will always return a positive acknowledgement, even if the object already exists. The parameters of the **CreateODBObject** are:
 - *object* (in): the representation of the object being created;
 - *objectUID* (out): the identifier of the created object.
- **RemoveODBObject:** Removes the object from the Object Database, and causes the **RemoveSDBEntry** operation to be invoked. Only a manager for the object's type may invoke this operation. It causes the Object Database to invoke nested **RemoveODBObject** operations on each host that the object is replicated on. The Object Database checks that the security label of the invoker dominates the security label of the named object. The parameters of the **RemoveODBObject** are:
 - *objectUID* (in): identifier of the object to removed.
- **ReadODBObject:** Returns the state of the object from the Object Database. The Object Database checks that the security label of the manager is the same as the security label of the named object. The parameters of the **ReadODBObject** are:
 - *objectUID* (in): identifier of the object to read;
 - *object* (out): representation of the object.
- **WriteODBObject:** Writes a new state of an object into the Object Database. Only the manager for the object's type may invoke this operation. The Object Database checks that the security label of the invoker is dominated by the security label of the named object. The parameters of the **WriteODBObject** are:
 - *objectUID* (in): identifier of the object to written;
 - *object* (in): representation of the object;

- **ReplicateODBObject:** Creates a copy of an existing object in the local Object Database on a remote Object Database on the named host. This causes the **ReplicateSDBEntry** operation to be invoked by the local Object Database to move a copy of the object's label to the local host. Only a manager for the object's type may invoke this operation. The parameters of the **ReplicateODBObject** are:
 - *objectUID* (in): identifier of the object to replicate;
 - *host* (in): host where a copy of the object already resides.
- **DereplicateODBObject:** Removes a copy of an existing object in the local database. Only a manager for the object's type may invoke this operation. It causes the **DereplicateSDBEntry** operation to be invoked to remove the label of the object from the local SDB. The parameters of the **DereplicateODBObject** are:
 - *objectUID* (in): identifier of the object being dereplicated.

3.5.2.3 Host Monitoring and Control

The majority of host operations for monitoring and controlling a DOS kernel on a host are unaffected by the addition of multilevel security features. The operations are:

- **Restart:** to restart the kernel;
- **Shutdown:** to shut the kernel down;
- **CreateService:** to create a new manager process;
- **RemoveService:** to terminate a manager;
- **ObtainServices:** to list the services configed to execute on the host;
- **ListServices:** to list the services that are started on the host;
- **ListProcesses:** to list the processes executing on the host;
- **SetObjectCache:** to add an entry to the object cache;
- **ClearObjectCache:** to empty the object cache;
- **DumpObjectCache:** to display the contents of the object cache;
- **ReportStatus:** to report the status of the kernel;
- **GenerateUno:** to create a new set of unique numbers (UNOs) used to create unique identifiers (UIDs).

Of these, cache controlling operations are extended to support a multilevel cache, but their interfaces remain unchanged from the Cronus counterparts. Additionally, **ListServices** and **ListProcesses** return status information for the appropriate access class. And a mandatory discretionary policy will limit who can start and stop services and the kernel (e.g., the System Manager) in a more constrained way (i.e., their access control lists will reflect the discretionary security policy on this information). However, no interfaces will change.

3.5.3 Type Process

Process objects are the schedulable entities in the system. They are active, meaning they invoke operations on objects, and they have identities and labels. Their identities determine the objects they are permitted to access. Their labels determine the labels they may assign to messages they send and the set of messages they are permitted to receive.

The operations defined on processes include:

- **SetProcessBindings**: assigns the process a label and an identity. This operation is invoked by the Authentication Manager as a result of a user authentication (the **AuthenticateAs** operation) or the initiation of a service (the **CreateService** operation or automatic initiation of a service by the message switch). The parameters to **SetProcessBindings** are:
 - *bindings* (in): process bindings (i.e., their identity; see Section 3.6.4 for more details)
 - *label* (in): label of the process.
- **ShowProcessBindings**: returns the bindings of the process. This operation is invoked by a manager needing the bindings of a client process for access authorization. The parameters to **ShowProcessBindings** are:
 - *bindings* (out): process bindings.
- **ClearProcessBindings**: removes the bindings of the process.
- **ChangeActiveCCI**: sets the active client contextual identity (see Section 3.6.4 for a complete description). Invoked by a process on itself. The parameters to **ChangeActiveCCI** are:
 - *CCIname* (in): name of CCI to make active.
- **ModifyActiveCCI**: further constrains a client contextual identity. Invoked by a process on itself. The parameters to **ModifyActiveCCI** are:
 - *CCIname* (in): name of CCI to modify;
 - *CCI* (in): new CCI to replace the named one.

- **ObtainProxy**: returns a client contextual identity (CCI) that the process received from a second process. The process that invokes this operation (the invoker) is a manager attempting to do access authorization. It is attempting to obtain a CCI for one of its clients. The process on which this operation is invoked (the invokee) is a manager that invoked an operation on the invoker. A proxy marker for the CCI was passed to the invokee by the client process. The parameters to **ObtainProxy** are:
 - *proxymarker* (in): proxy marker for which CCI is to be returned;
 - *CCI* (out): CCI which the marker identifies.
- **ObtainKey**: returns the public encryption key of the process manager. The parameters to **ObtainKey** are:
 - *publickey* (out): public encryption key of the Process Manager.
- **SetKeys**: sets the public and private encryption keys for a manager on a host. This operation may only be invoked by the System Manager. The parameters to **SetKeys** are:
 - *publickey* (in): public encryption key;
 - *privatekey* (in): private encryption key.
- **ReadACLDescription**: returns the description of the ACL for a type. The invokee of the operation is a manager process. The parameters to **ReadACLDescription** are:
 - *bindings* (out): process bindings.

In addition, the following operations are supported by all managers (they are implemented by type Object and are inherited by all other types):

- **ReportStatus**: show the status of the manager;
- **SetLoggingLevel**: set the level of detail for the manager to print in logging its actions;
- **SetTaskingLevel**: set the number of tasks that may be initiated to handle concurrent requests by this manager;
- **DescribeType**: describes the interface to this type.

These operations are unaffected by the introduction of mandatory security features. However, these operations are no longer generic operations, but are rather operations on a process rather than on a type. As a result, the invokee of these operation should always be a manager process.

3.5.4 Type Principal

A principal is an object which contains an identity and label that becomes associated with processes as part of user authentication or service initiation. The operations defined on principals are:

- **ChangePrincipalPassword:** modify the password of a principal. This operation may only be invoked by the System Manager. The parameters to **ChangePrincipalPassword** are:
 - *password* (in): new password for the principal.
- **LookupPrincipalName:** Return the name of a principal. The parameters to **LookupPrincipalName** are:
 - *prinname* (out): name of principal.
- **LookupPrincipalUID:** returns the UID of a named principal. The parameters to **LookupPrincipalUID** are:
 - *prinname* (in): name of the principal.
 - *prinUID* (out): UID of the principal.
- **AuthenticateAs:** authenticates a particular process as the principal. Causes the password and username to be looked up and compared to the input values. If the operation succeeds, it causes the **SetProcessBindings** operation to be invoked. The parameters to **AuthenticateAs** are:
 - *username* (in): name of user being authenticated.
 - *password* (out): password of user account.
- **ChangePrincipalLabel:** used to modify the label associated with the principal when it is bound to a process. Note that this label is not the label of the principal itself. The parameters to **ChangePrincipalLabel** are:
 - *label* (in): label of processes that are bound to this principal.
- **AddCCI:** adds a client contextual identity to the principal that will be assigned to a process executing as this principal. The parameters to **AddCCI** are:
 - *CCI* (in): new client contextual identity.
 - *CCIname* (in): name of the new client contextual identity.
- **RemoveCCI:** removes a client context identity of a principal. The parameters to **RemoveCCI** are:
 - *CCIname* (in): client contextual identity to remove.

- **SetDefaultCCI:** selects a particular client contextual identity to be set to the the active CCI when it is bound to a process. The parameters to **SetDefaultCCI** are:
 - *CCIname* (in): client contextual identity to set to the default active CCI.

3.5.5 Type Project

A project is a set of principals. When a project is created, it is inserted within an existing project. Thus, projects are organized into a hierarchy. The operations defined on projects are:

- **ShowMembers:** returns the members of a project. The parameters to **ShowMembers** are:
 - *members* (out): list of principals.
- **AddMembers:** adds new principals to a project. The parameters to **AddMember** are:
 - *principals* (in): list of principals to add to the project.
- **RemoveMembers:** removes principals from a project. The parameters to **RemoveMember** are:
 - *principals* (in): list of principals to remove from the project.
- **LookupProjectName:** looks up a project's name from its identifier. The parameters to **LookupProjectName** are:
 - *projectname* (out): name of the project.
- **LookupProjectUID:** looks up a project identifier from its name. The parameters to **LookupProjectUID** are:
 - *projectname* (in): name of the project;
 - *projectUID* (out): UID of the named project.
- **RenameProject:** renames a project. The parameters to **RenameProject** are:
 - *projectname* (in): new name of the project.

3.5.6 Type Object

SDOS types will be implemented within a type hierarchy. This hierarchy supports operation inheritance, in which operations defined by a type are inherited, or reusable, by any of its subtypes. Type Object is the root, or supertype, of the SDOS type hierarchy. Most of the operation interfaces on type object are unaffected by mandatory security features. They will have the same interface as documented in the Cronus User's Manual [Cronus 88]. However, the behavior of operations may be constrained by mandatory rules.

The parameters of many of these operations vary from object to object as they are reimplemented to be specialized for each specific type. We only include parameters for operations that involved discretionary access control.

The operations defined on type object are:

- **Create:** Creates a new object. A label is added to the interface which specifies the label of the object to create. This label is passed on to **CreateODBObject**. The invoker's label must be dominated by the object's label. The parameters of the operation depend on the type of object being created; this operation is generally refined for each type that inherits it.
- **Remove:** Removes an object. See the Cronus User's Manual for the Interface description. The invoker's label must be dominated by the object's label. There is generally no operand beyond the object being removed.
- **Locate:** Returns the location of an object. There is no operand being the object being located. The object's label must be dominated by the invoker's label.
- **ReadACL:** Returns the contents of the access control list of the object (see section 3.6.4). The object's label must be dominated by the invoker's label. The parameters to **ReadACL** are:
 - *ACL* (out): the access control list for the object.
- **AddToACL:** Adds entries to the object's ACL. See the Cronus User's Manual for the interface description. The invoker's label must be dominated by the object's label. The parameters to **AddToACL** are:
 - *ACLentries* (in): A list of ACL entries to be added to the object's ACL.
- **RemoveFromACL:** Removes entries to the object's ACL. The invoker's label must be dominated by the object's label. The parameters to **RemoveFromACL** are:
 - *ACLentries* (in): A list of ACL entries to be removed from the object's ACL.

3.6 Design Specification

This section describes the security-relevant parts of the SDOS design. It covers the modules and databases that participate in the enforcement of the discretionary and mandatory security policies. Some discussion of the reasons for design decisions is included. Readers are assumed to be familiar with Section 3.1, which introduces the concepts of clients, managers and objects. It should also be noted that there are several discrepancies between the formal top level specification presented in the next section and this design. These discrepancies reflect several simplifying assumptions made to simplify the formalization activities; they are discussed further in Section 4.1.

The SDOS mandatory security policy is enforced by the SDOS kernel. Descriptions of the kernel's modules and databases are given below. Additionally, the SDOS trusted computing base, or TCB, is described. The SDOS discretionary security policy is enforced by the individual object managers because some aspects of the discretionary policy are object-type-specific. The reasons for the location of the discretionary policy are discussed in more detail below. The discretionary policy which must be implemented by each manager is described in detail below.

3.6.1 Reasons for Design Decisions

3.6.1.1 DAC in Managers

One of the fundamental decisions in the design of a secure system is the choice of locations for the implementation of security mechanisms. Factors that enter into the decision include architectural considerations (proper layering and performance), assurance of the trusted mechanisms, and minimization of TCB size. The decision to implement mandatory controls in the SDOS kernel was quite clear: mandatory controls in a message passing system are naturally placed in the component which routes messages, as this is the central point of mediation between the correspondents.

For discretionary controls the choice was more difficult. The SDOS discretionary control scheme allows, in general, for a different set of discretionary control privileges to be defined for each object type. The SDOS discretionary control scheme is discussed in great detail below. Briefly, an ACL consists of a list of entries, each of which consists of some client identity information, and a list of the operations (privileges) which the bearer of that identity is permitted to invoke on the object. The set of legal operations is different for each object type. Therefore the set of operations that can appear in an ACL entry is different for each object type, and the code that searches and interprets an ACL must be different for each object type (at the very least, driven from different tables). This argues for placing ACL interpretation in the manager of each object type.

However, managers are assumed to have different levels of assurance, being writable by different users. The idea that it is acceptable for DAC to have lower assurance than mandatory controls has gained some acceptance recently. We suggest that each type

have a discretionary security policy which determines the required assurance for that type. For developers of untrusted and unreliable application software, discretionary access controls will have little assurance. Developers with the expertise to write multilevel secure application software, may extend the system TCB to encompass the discretionary controls. Additionally, as discussed in 3.6.4, we suggest ways that application developers with limited expertise can achieve a high assurance of discretionary controls when appropriate mechanisms are supported at lower levels. Placing discretionary controls in managers provides the application developer with flexibility over both the functionality of the controls and their assurance.

3.6.1.2 Mandatory Security

There are several design decisions that relate to the SDOS mandatory security controls. Some are dictated by adopting an object-oriented architecture based on message passing, others reflect trade-offs between generality and complexity. This section attempts to make explicit the most basic of these decisions as well as describing some of the complex ones.

Hosts may be multi-level secure (i.e., have access class ranges). This decision is dictated by our implementation strategy to include both single level and multi-level secure hosts in our distributed environment. A distributed system consisting of simply single-level hosts makes it impossible to access (in an automated fashion) arbitrary objects across access classes. This is a severe limitation. Access is restricted by the inability of high-level clients to read down into untrusted managers at lower levels in order to access lower-level objects without leaking information. This limitation is a direct result of the required communication between (frequently) untrusted correspondents in order to invoke operations reliable in SDOS. In contrast, conventional systems allow access to only passive data through trusted channels, and thus avoid this limitation. However, they also make it impossible to define data abstractions in application-specific ways.

Manager processes may be either single level or multi-level secure. Multi-level secure managers must populate a system in order to allow objects to span a wide-range of access classes on a single host when the objects have requirements for high performance (e.g., files). Though the functionality of any multi-level manager can be achieved by a set of single-level managers, the limitations that operating systems place on the number of processes makes SLS managers prohibitive in many instances. Although SLS managers can be instantiated and destroyed dynamically to get around this limitation, the performance of such a mechanism will not always be acceptable. Additionally, building composite MLS structures (for example, a multi-level secure document consisting of paragraphs with different access classes) is only feasibly achieved with MLS managers.

All other (nonhost, nonprocess) objects are single level secure. This choice presented a trade-off between system complexity and understandability; the decision was made to simplify the mechanisms supporting object location and storage. Structures which

are inherently multi-level secure can be implemented a set of single-level secure objects with different access classes. Such a structure can be defined as an object by an MLS manager. The security policy on the MLS manager would dictate that the object be given an access class that dominates the access classes of all of its components.

All SDOS object types are potentially multi-class. That is, the access class of objects of a type are limited only by the assurance of software that manages objects of the type and the access class range of the host on which they reside.

Not all managers will be certified as MLS. This is critical yet straightforward decision. Managers are the implementation of applications and it is assumed that there will be a wide variety of managers in a typical SDOS configuration. Since it will not be feasible in the foreseeable future to demonstrate whether an arbitrary manager satisfies an MLS policy, we can expect that user-written managers will generally not be certified as MLS. These SLS managers will be confined to operate within a single access class; the security policy is imposed on them by the SDOS kernel and the underlying COS.

Single level secure processes cannot reliably communicate with one another if they have different access classes. This is a direct consequence of the security policy applied over a sequence of send message operations. It is based on the design principle adopted from the Cronus DOS that the Message Switch delivers messages without inspection of their content (with the exception of their security label). Kernel minimality and the extensibility of the system make it undesirable and impossible for Message Switches to examine messages. Since the Message Switch will simply forward messages in accordance with the mandatory security policy, it is impossible to transfer a message between SLS processes in different access classes without leaking information. A typical nonmanager client process is assumed to be running completely untrusted code. Such a client can only communicate with a manager at its own access class (or in the case of a MLS manager, one whose range includes the class of the client). An *invoke down* (e.g., a top secret client invoking an operation on a secret object) can only be handled by a MLS manager; with single level managers, the top secret client would be forbidden from communicating with the secret manager. An *invoke up* can return no results, so it can only be used to write up to an object. Invoking up is achieved by the client indicating that the operation is a write up.

It is possible to run more than one SLS manager for a given type on a single host. The generality of most applications will result in objects of a type to be at a wide range of access classes. Requiring these types to be MLS is unnecessarily restrictive. Requiring that a manager for each access class execute on a different host does not scale. Therefore it must be possible to execute managers at each access class on a single host.

Single level managers can be dynamically instantiated in response to demand for them. Since there could be objects at a very large number of different access classes, it is impractical for single level managers at every access class to be running all the time; the COS process tables and memory resources would be exhausted. Exactly how they are instantiated (and disinstantiated), and exactly what constitutes demand for them, are the subjects of a very involved design discussion, which will be summarized later.

With the exception of systems calls to send and receive messages, all system calls conform to the object oriented invocation protocol. This decision represents the selection of increased functionality and flexibility over performance. Systems calls are those operations handled by the kernel (specifically, by the Process and Host Managers). By standardizing these system calls using the object invocation protocol, it is possible to invoke any of these operations from remote hosts. This allows remote access to the Security Database (such as when coordinating the update of an object's security label), and remote access to the Object Database (providing similar functionality as a remote file system).

3.6.2 The SDOS Trusted Computing Base

The SDOS TCB consists of the kernel and a set of multi-level secure managers that provide system services. The trusted managers operate according to the object-oriented abstraction, just as all other managers. Since the kernel is the entity that implements the object-oriented abstraction, that abstraction is not available for use within the kernel.

The kernel consists of the Message Switch, the Locator, the Process Manager, the Host Manager, the Process Table, the Security Database and the Object Database. The trusted managers include the File Manager, Catalog Manager, Authentication Manager, and Trusted Interface Process. The Catalog Manager and File Managers are trusted because they manage many objects of different access classes and need to be MLS for high performance. The Authentication Manager is trusted because of the required assurance of authentication. The Trusted Interface Process is trusted to provide a trusted path between clients and other services (e.g., the Authentication Manager for authentication and to services to modify access control privileges).

The function of each of these TCB components is briefly described below. More detailed descriptions follow in succeeding sections.

- **Message Switch:** Routes messages between entities, both locally and remotely. Enforces the mandatory security policy governing the passing of messages between entities. Communicates with its peer Message Switches on other hosts, and cooperates with them in the passing of messages and the enforcement of the security policy.
- **Locator:** Locates objects that do not reside on the local host. Provides this service only to the local Message Switch, and responds to locate requests from remote objects. Maintains a cache containing the locations (remote host IDs) of recently used remote objects. Remote objects are located by broadcasting a request for the object to all hosts. If no positive response is received after a suitable interval, failure is reported to the Message Switch.
- **Process Manager:** Creates and destroys processes, and maintains (sets and shows) process bindings. Process bindings is the term for the set of information

that includes a user's identity, and mandatory and discretionary access control attributes.

- **Host Manager:** Operations on hosts are used to monitor and control the host, such as creating new services, access the object cache for locating objects, and booting and shutting down the system.
- **Security Database:** The collection of data needed for the enforcement of the mandatory security policy. This information includes, for each entity on a host: an access class label; a switch indicating single-level or multilevel-secure; for a replicated object, the number of replicas in the system; and for an object type, information about its manager, including: whether local managers exist, whether they are active, and the location of their executable code.
- **Object Database:** Provides storage for all objects that reside on the local host. Used by object managers.
- **Process Table:** Contains information about all active processes on the local host, including the process bindings. Maintained by the Process Manager. Consulted also by the Message Switch, when making mandatory access control decisions.
- **File Manager:** A multilevel secure manager, allowing the write-up and read-down operations. Also implements create, delete, open, and close operations.
- **Catalog Manager:** Provides an abstract space of symbolic names for objects. Translates from an object's symbolic name into its UID. (The UID is used to reference an object when invoking an operation.)
- **Authentication Manager:** Implements login and logout requests from interactive users. Sets appropriate process bindings for users logging in.
- **Trusted Interface Process:** Implements a trusted path between the system and an interactive user at a terminal. Maintains the state of the terminal, with respect to whether or not a user is logged in. Relays login requests to the authentication manager. Relays the requests of a logged-in user to other parts of the system.
- **Other MLS Managers:** Developers can implement other MLS managers and add them to the system. The System Certifier is responsible for certifying their MLS properties, and the System Manager configures the managers on hosts.

Figure 3.7 shows the communication paths between client and manager processes and the SDOS kernel. Figure 3.8 shows the direct logical communication paths between these components, hiding the Message Switch and Locator.

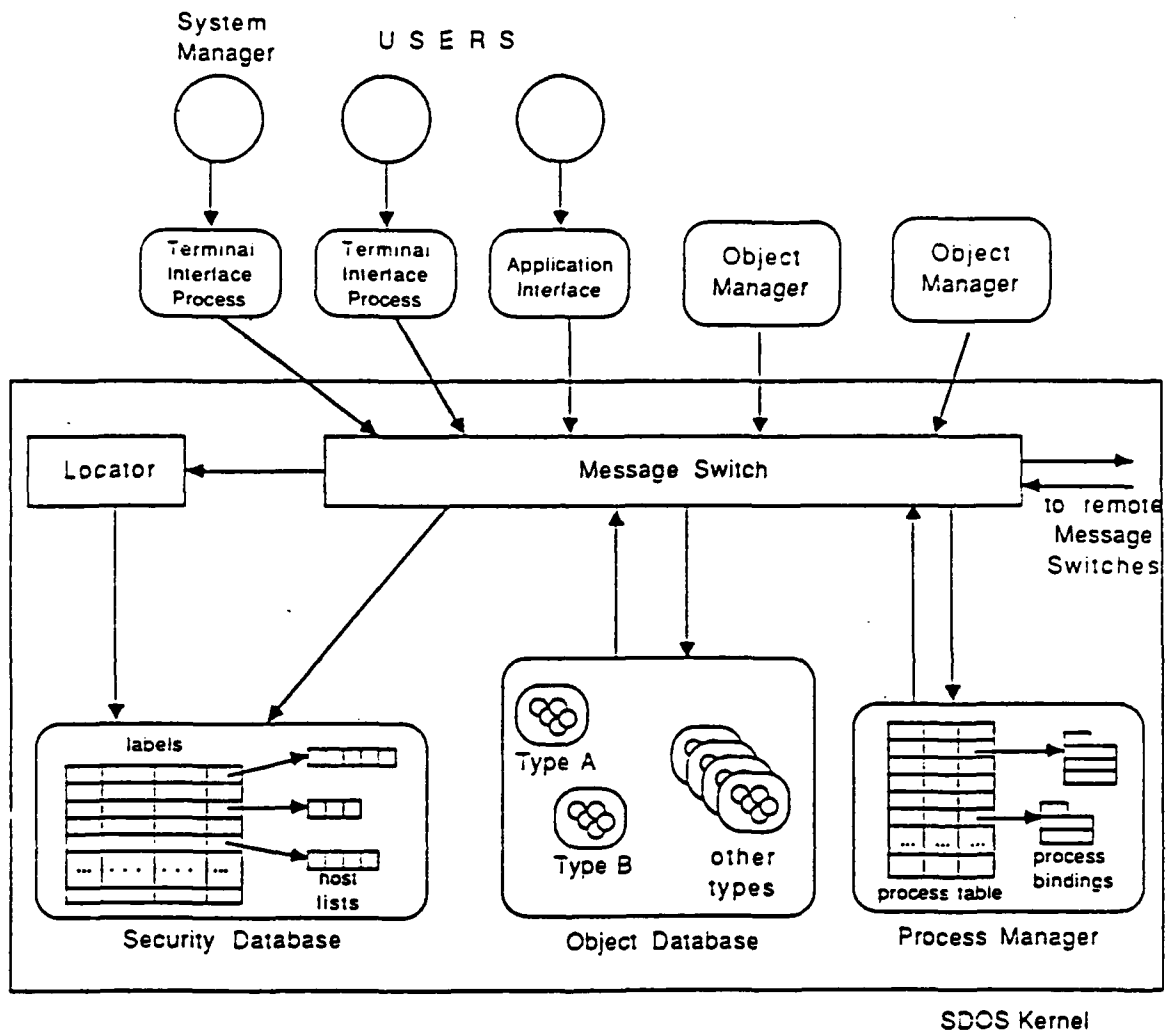


Figure 3.7: SDOS Communication Paths

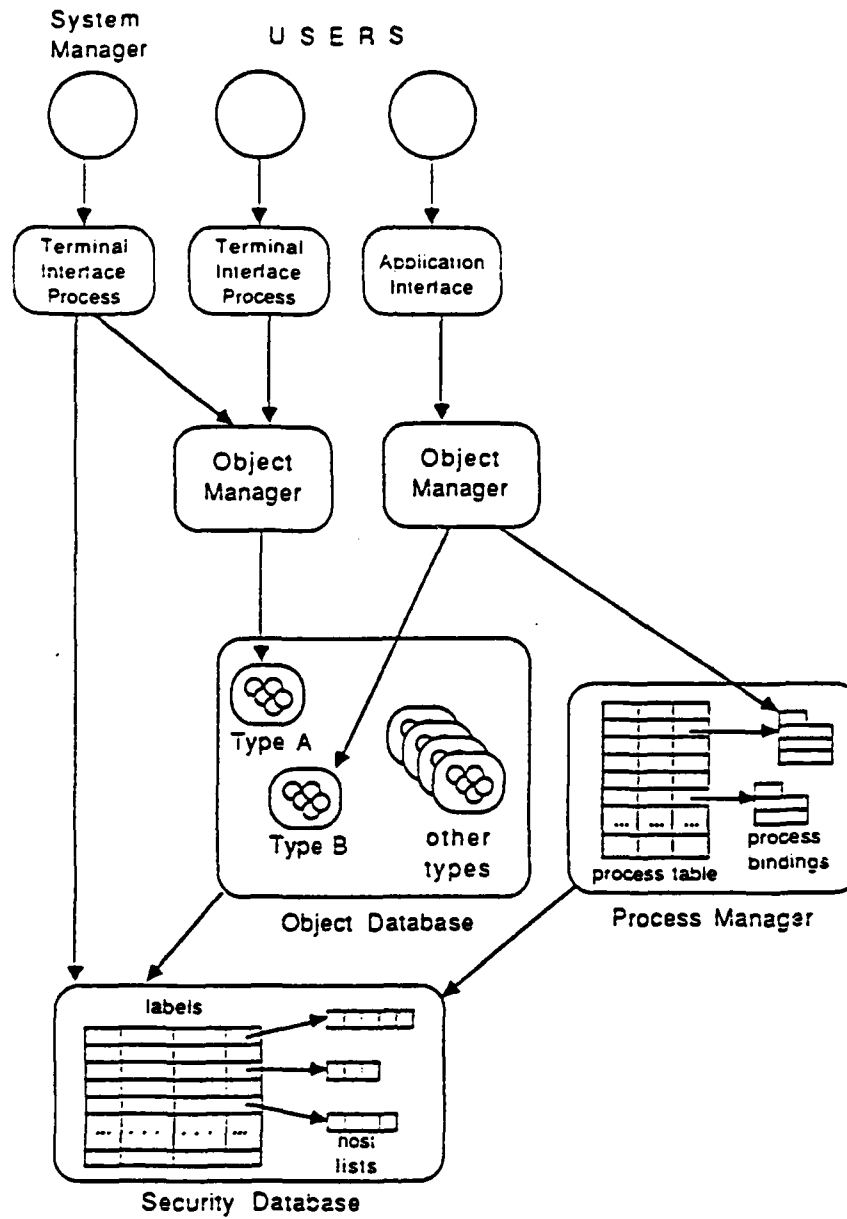


Figure 3.8: Communication Paths, Hiding the Message Switch

3.6.3 Detailed Description of the Major TCB Components

3.6.3.1 The Message Switch

The Message Switch moves messages between processes and between hosts. It provides four major services: checking the security label specified for a message; stamping a security label, host, and process identifier on messages; locating the destination of messages; and transferring messages from their source to their destination. When a process invokes a **Invoke** or **Reply** operation to transfer a message, the following actions are taken by the Message Switch:

- When an operation to transfer a message is initiated, the Message Switch determines if the security label of the message is properly set based on the security label of the process sending the message. The security label of the process is obtained from the Security Database using the identifier of the process. If the process transferring the message does not have the privilege to set the security label of the message to its current value, then the operation returns an error. If the *label* field in the operation is null then the field is set to the security label of the sending process.
- If the client indicates that the operation is a write up, then the label of the object on which the operation was invoked is used as the label of the message.
- The security label, sending process, and local host identifier are added to the message.
- The Message Switch determines the destination of the message. If the **Reply** operation was used to send the message, then the destination is retrieved from the host field of the UID of the process. If the message is being transferred with **Invoke**, then the Message Switch transfers the message to the Locator. Asynchronously, the Locator will return a host destination and label of the object, or an error. If the client indicated that the invoke was a write up operation, then the label of the object is used as the message's label.
- When the destination of a message is on a remote host, the Message Switch on the source host is responsible for relaying the message to the Message Switch on the remote host. The Message Switch will obtain the label of the destination host from its local Object Database. If the label of the messages is within the range of labels of the host then the message is routed to the destination host. Otherwise, the Message Switch returns an error. (Note that this check could also be performed within the communications subsystem.)
- When a message is transferred between hosts, the Message Switch on the destination host checks that the source host has the privilege to send the message based on the security label of the message and the security label of the sending host. The security label of the sending host is obtained from the Security Database. If

the transfer is not secure, then the remote host Message Switch sends a negative acknowledge for the transfer and discards the message. (Note that this check could also be performed within the communications subsystem.)

- When a message is received by the Message Switch on the destination host, the Message Switch determines the manager to receive the message. If the message was sent with a **Reply** operation, it is directed to the process named in the invocation. If the message was sent with the **Invoke** operation, the Message Switch checks the Process Table for the object's type. This entry will indicate whether a new manager process should be instantiated (when no process exists or the manager is a single client manager; see the Process Table discussion below), or the message transferred to an existing one.

3.6.3.2 The Process Table

The Process Table is a structure maintained by the Process Manager which records information about client processes and managers executing on a host, and managers that are configured to execute but may be dormant. The Process Table records the process identifier, an indication of whether it is a client or service, its process bindings, administrative information (e.g., its COS process identifier), and, if the process is a manager, whether the manager processes should be automatically deinstantiated. The Process Table is read by the Message Switch to determine when to instantiate a new manager.

Managers will be disinstantiated in response to requests to instantiate new managers, when COS resources are close to exhaustion. Algorithms for choosing a manager to disinstantiate are similar to page replacement algorithms: choose the least recently used page (or manager), but allow for some system-manager-settable parameters to give more preference to managers that are known to be performance-critical or expensive to instantiate (long initialization times). To increase performance, a few *empty* manager processes could be kept around, to speed up instantiation of managers. These processes would have all of the COS-specific initialization already done, and would only need to run the manager-specific initialization before becoming usable.

3.6.3.3 The Locator

The Locator is a module within the kernel that determines the location of an object when an operation is invoked on it. The Locator needs to be multi-level secure in order to read the Security Database to discover objects' existence. The Locator could be placed in a process outside the kernel at the cost of decreased performance. The Locator maintains a cache of remote objects to expedite the location of frequently referenced objects. The cache is multi-level secure to avoid its use as a covert timing channel (see 4.1.2.5 for a further discussion of these issues).

The following steps take place in the Locator, when the OS asks it for the location of an object:

- It consults its cache; if the object is in it, the cached location is immediately returned to the Message Switch.
- If the object is not in the cache, the Locator issues a **Locate** operation on the object. This operation is broadcast to all hosts.
- On each host, the Message Switch receives the locate and forwards it to the Locator on the host. A Locator will return with the following information:
 - an indication of whether the object is located on the host (obtained by examining the Security Database);
 - the execution status of the manager (obtained from the Security Database and the Process Table):
 - * whether a manager for the object is already executing on the host; if so, the manager process is returned;
 - * whether a manager may be dynamically instantiated on the host;
 - load or storage status of machine, for resource control (this is obtained from performance measures that could be maintained by the Kernel);
 - any type dependent data that was registered with the Locator (such as how to handle specific operations); this may be obtained from the Security Database;
 - the label of the object (if found).

This information is obtained from the Process Table.

- The Locator originating the locate request waits for responses. If several valid destinations are returned, then the object is replicated. In this case, the Locator chooses the *best* one. The Locator uses information about the application (such as which copy of the object is the primary copy) or run-time information (e.g., performance information) to choose the best copy.

The final destination(s) is stored in the object cache. The best destination (or only destination, in the case of unreplicated objects) destination is returned to the Message Switch, which forwards the message to the host. If no (satisfactory) responses are received by the client's time-out period, the locate fails. The Locator so informs the Message Switch, which passes the word back to the client.

3.6.3.4 The Host Type

The Host Type defines a set of host objects. Each host object resides on the machine it represents, and is managed by a Host Manager on the host. Host objects provide an addressing technique that allows system calls to conform to the standard operation invocation protocol. Its chief advantage is that it allows host operations (i.e., system calls

to the Security and Object Databases) to be invoked from remote hosts. The Locator on a host uses Host operations to maintain the cache. Additionally, the Security and Object Database operations, as described in the Descriptive Top-Level Specification, are Host operations.

3.6.3.4.1 The Security Database The Security Database contains three tables: an Object Table, a Type Table, and a Host Table. All of these tables are accessible through the same interface, and an entry in any of the tables is addressed using the UID of the object associated with the entry.

The Object Table contains an entry for each object in the Object Database of the local host. An object entry includes the security label of the object, and a value indicating whether the object is single-level secure or multi-level secure (MLS). The Object Table is used by the Locator to locate an object, by the System Manager to set the security label of the object, and by the Object Database to check that access to the object is secure.

The Object Table also maintains a list of hosts where a replicated object is stored. Replicated objects need to have replicated labels to ensure their continuous access. Since it is imperative that all copies of an object's label be consistent, the Security Database provides operations to replicate and dereplicate entries in the Object Table.

The Type Table contains an entry for each type that is configured for the local host. If an entry for a type resides in the table, then objects of the type may be created and accessed on this host. A type entry includes:

- the range of security labels at which managers of the type may be instantiated;
- whether a manager should be dynamically instantiated;
- whether managers are multi-level or single level secure; and
- the name of the object file that contains the executable image of its manager.

The Type Table is used by the Message Switch to instantiate manager processes dynamically and by the Host Manager to create new services.

The Host Table contains an entry for each host in the system. A host entry contains the security level of the host. The Host Table is used by the Message Switch to determine if each message received has a security level in accordance with the security level of the host that originally sent the message.

3.6.3.4.2 The Object Database The Object Database (ODB) contains an entry for each object managed on the host. Each Object Database entry contains the stable storage representation of the object, which only can be accessed by the manager of an object.

When the Object Database receives a request to access the database, it first checks that the invoker of the object is a manager of the type of object being accessed. The ODB determines whether or not the manager manages the object's type by accessing the Process Table entry for the manager process.

The Object Database enforces the security policy by accepting **CreateODBObject**, **ReplicateODBObject**, **RemoveODBObject**, **WriteODBObject** and **DereplicateODBObject** operations only from managers whose labels are dominated by the label of the object, and **ReadODBObject**: only from managers whose labels dominate the label of the object.

3.6.4 Discretionary Access Control

Access control is traditionally discretionary in nature, meaning that the privileges to access an object may be modified in unrestricted ways by a designated group of users. The access matrix model is used to formalize discretionary access control. Rows of the matrix indicate accessors, columns indicate objects that can be accessed, and cells indicate the privilege of the accessor to access the object.

Access controls can be implemented either with *capabilities* or *access control lists* (ACLs). Capabilities are formed by grouping matrix cells by row and storing them with the accessor. ACLs group by column and are stored with the object. The symmetry of these mechanisms makes one approach's advantages the other's disadvantages, and vice versa. This, in turn, tends to polarize people's views on the subject.

The capability approach allows accessors to review all of their privileges, allows privilege transfer between processes, and allows accessors to be given access privileges with finer granularity (e.g., spawned processes can inherit a subset of capabilities; inheriting a subset of identities usually does not make sense). Capability-based systems also commonly use capabilities to name objects at the system level. On the other hand, ACLs make it possible to review all of the potential accessors to objects, makes revocation of privileges trivial, and is the widely adopted convention for access control.

Although capability-based systems have received considerable attention in systems research, there has been little experience with their application in distributed systems that have received extensive use. Supporting capabilities is generally more complex than ACLs because access rights are distributed with the clients rather than grouped where objects are managed. Capabilities must be stored with the client and transferred to services in a way that they cannot be forged or corrupted. It is also difficult to integrate separately developed capability-based systems, or a capability-based system with an ACL-based systems. This reflects the difference in the representation and interpretation of access rights in capabilities, and the common use of capabilities for naming. It has been observed that the ability to review who has access to information in computer systems used by the military is far more critical than the ability to review those privileges a particular user has. All of these factors limit the efficacy of capability-based systems for BM/C3 applications. We have adopted ACLs for the SDOS system.

3.6.4.1 Client Identities

A client's identity is bound to its process as part of its authentication prior to its access of objects. In this section we describe how client identities are initialized, maintained, and transferred.

3.6.4.1.1 Principals Clients have been defined as those entities in the system that are capable of requesting access to objects by invoking operations on the objects. Each client is a process that is bound to an identity. The nonvariable part of a client's identity is called a *principal*. A client is only bound to a single principal at a time, and this binding usually is made for the lifetime of the client process. For clients that represent human users, the name of the principal corresponds to the user's name. For a client that represents a manager, it is bound to a principal named after the manager. Principals are first class SDOS objects, and are managed by the Authentication Manager.

3.6.4.1.2 Projects A client is generally working on a clearly identifiable activity, and many of the objects a client uses are specifically associated with the activity. For example, many objects are used simply for evaluating the C2 Internet experiment. The activity may be an application (e.g., C2.Sensor), an administrative domain (e.g. Division_4), or any other organizational unit. We call the unit used to delineate sets of client activities a *project*. A client representing a principal may work in several different projects over a relatively short span of time, and other clients representing other principals may also work in the same project. A project is actually represented as a group of principals. Projects are first class SDOS objects managed by the Authentication Manager. Changing the membership of projects may change the identity of clients bound to principals that are affected by the change.

Projects may be divided into subprojects, forming a hierarchy of projects. If a principal is in a particular project then the principal is a member of every ancestor project of the project. One difference is that nesting within projects is expressed syntactically: *prjA.prjB* indicates that *prjB* is a subproject of project *prjA*. For example, the evaluation subproject in C2 Internet project might be represented as the project *C2.evaluation*.

Although a project is a group of principals, projects can also be thought of as a way to delineate sets of objects. For example, there are likely to be a specific set of objects associated with the C2 Internet project that are only used within that application. Other objects may be accessed by clients executing within many different projects. Who can access an object is dependent on the client identities that appear on the access control list of the object.

3.6.4.1.3 Roles Although individual object types vary, the patterns of object use across many different types of object are similar with respect to privilege sharing and common operations. As examples, one use of all objects is monitoring their activity, and **ReportStatus** is an operation that will be defined on every object. Patterns of

resource usage are called **roles**. Intuitively, roles are the set of operations (possibly across several object types) that are associated with a particular task that is performed by many clients. Roles are not first class SDOS objects and do not require access control.

Roles are created for each object type as a (sub)set of the operations defined by the type. By being defined on the type, roles are defined consistently across all objects of a type. Ideally, the meaning of a role will be the same across many or all object types. For example, a *monitor* role is intended to be used to monitor that status of objects, regardless of the type of object. Defining roles consistently simplifies their use, but has no impact on security.

3.6.4.1.4 Actual and Contextual Client Identities The principal, project and role concepts are applied in SDOS in the form of the Client Identity (CI) that is associated with each client. This association is maintained by the manager of the client processes, the Process Manager. The CI of a client defines its identity. Since access authorization is based on identities, the CI of a client indirectly defines the set of privileges the client has to access resources.

The CI consists of two parts: the *Actual Client Identity (ACI)* and a set of *Contextual Client Identities (CCIs)*. When an operation is invoked on an object, the object's manager obtains the client's ACI and *active* CCI from the client's Process Manager to identify the accessor.

The ACI consists of the principal bound to the client and a flag indicating whether or not the client is a terminal interface process. The designation of terminal interface processes is needed to control the invocation of direct operations (or trusted paths; see the security policy). This attribute is set at process creation time (the mechanism for this has not been designed).

When a client accesses an object, it acts on behalf of a principal, within a project, and generally in a narrowly defined role. We call this information the contextual identity of the client. One client may perform one role with respect to one project, but a different role with respect to another project (or the entire system), all on behalf of the same principal. Therefore, a client commonly has several contextual identities. Furthermore, a client may act on behalf of itself (i.e., on behalf of the principal bound to it), or it may act on behalf of another client (which could be bound to a different principal). The CCI identifies on whose behalf and in what capacity the client is invoking an abstract operation. A CCI is a quadruple of the following form (brackets indicate optional fields): (Principal : Project : Role [: ObjectList]).

A special project and role *ANY* is defined that allows the client to act in any project or role. Similarly, a special project and role *NIL* exists that prevents a process from being a client. If an object list is present in a CCI, the client may only access the listed set of objects using this contextual identity.

A client will have several CCIs in its CI, and all will be visible to the user (for user processes) or programmer. A CCI allows a client to divide his activities into separate

collections. Client contextual identities can be useful both as an organization tool and for self protection (for example, by choosing a *test* project that is used to access only objects being tested). At any point in time the client is associated with a specific CCI, the *active CCI*, which is used to identify the client with respect to a particular invocation. Each CCI entry is named, and a client is free to switch from one CCI to another by invoking the **ChangeActiveCCI** operation on itself and providing a CCI name. In addition to the ability to change their active CCIs, clients may examine and rename their CCIs.

When a client invokes an operation on an object, the active CCI at the time of the invocation is used by the object's manager to identify the client. When an operation is invoked, the client transfers a *CCI marker* indicating its active CCI to the object's manager. This marker simply contains the name of the active CCI and is used by the manager to obtain the client's active CCI.

The following are examples of CCI's:

(*Vinter : Private*): the *Vinter* principal working on private data.

(*Vinter : C2.evaluation : Tester*): *Vinter* working in the the *C2.evaluation* project as a *Tester*.

(*Vinter : C2.evaluation : Reader : Object_A*):

Vinter, again working in the *C2.evaluation* project, this time as a *Reader*, and only able to access *Object_A*.

(*Vinter : NIL : NIL*): *Vinter*, with no project or role, and therefore unable to act as a client.

Client identities can be used to easily represent the special SDOS users designated in the security policy:

System Controller: (*<Somebody> : System : Controller*)

System Certifier: (*<Somebody> : System : Certifier*)

System Manager: (*<Somebody> : System : Manager*)

3.6.4.1.5 Initial Client Identities An initial CI is associated with each principal object. An initial CI is a list of CCIs that can be bound to a client authenticated to have this identity. The principal identifier appears in the principal field of each CCI in the initial CI. Discretionary access controls on principals and projects constrain who may add and remove entries from an initial CI.

An identity is bound to a client in one of the following ways:

- A client representing a user is given a CI based on its initial CI via password authentication with the **AuthenticateAs** operation on the **Authentication Manager**;
- A client that is also an object manager is given a CI when it is initiated by the kernel. A principal exists for each manager defined, and the principal name is the same as the manager name.
- A client representing a spawned process inherits (a subset of) the CI of the process

that spawned it.

3.6.4.1.6 Proxy Client Contextual Identities Clients that are managers frequently act on behalf of the clients they are serving. An example is a client that attempts to print a file makes a request to a printer manager. The printer manager needs to read the file. Thus, the printer manager services printer requests, but acts as a client to the file manager. Giving the printer manager access to all files is undesirable because of the general untrustworthiness of arbitrary managers. Proxy CIs are used to temporarily transfer the privileges to access a set of objects to a manager. When a client invokes an operation on an object, it can designate a proxy CCI (which may be the same or different than the active CCI). The proxy CCI is added to the CI of the client serving the operation request. The manager makes the proxy CCI active when accessing objects on behalf of the client.

CCIs that are not proxies will have a principal field value identical to the principal in the ACI. This property reinforces the idea that a contextual identity for an executing client is either based on an identity received from another client (a proxy) or the identity originally bound to the client.

A client transfers a proxy CCI to a second client by passing a *CCI proxy marker* as a parameter to an operation invocation. This transfer must be visible to the programmer since the marker passed determines the (proxy) identity of the manager while servicing the operation invocation, and there will generally be several CCIs to choose from. The CCI proxy marker contains the following:

- The process UID of the client passing the proxy; this is used by the process possessing the proxy to (indirectly) obtain the CCI of the client.
- Encrypted in the public key of the Process Manager, the following:
 - the name of the CCI that is being passed;
 - the process UID of the client passing the proxy, encrypted in the private key of the Process Manager;
 - filler text, as necessary for security.

The PM public encryption key and the process UID of a client encrypted using the private key of the PM can be obtained by a client from the PM either when the client process is created or with the **ObtainKey** operation defined on each process. To make proxy CCI transfers secure, it is necessary to encrypt them to ensure that they are not forged or changed without detection. By encrypting the message in the public key of the PM, the proxy marker can only be read by the Process Manager of the client that sent the proxy. By encrypting the process UID with the private key of the PM, it is ensured that no process can forge a proxy CCI nor dupe the PM about the process that originally passed the CCI. Upon receiving a proxy CCI marker from an invocation, the manager places the marker in a newly created CCI. In its capacity as a client to a

second manager, the manager may make the proxy CCI active or transfer it to a second manager.

Proxy CCIs never have to be returned to the client originally transferring them because they are placed either in instantaneous managers that only serve a single client, or in perpetual managers that are trusted to destroy the proxy markers when they are finished using them. This is consistent with the definition of managers in SDOS.

3.6.4.1.7 Creating New Client Contextual Identities The CI can be manipulated by invoking operations on the client. Operations include the ability to add a new CCI, destroy a CCI, and change the name of CCIs. A client can create a new CCI in several ways:

- By taking an existing CCI and restricting it. This is generally done for the purpose of testing unreliable programs (limiting your own privilege temporarily), or because the CCI will be passed as a proxy to another client.
- By copying a named CCI from the initial CI into the CI. This is generally done to restore a CCI that has been previously destroyed.
- By placing a proxy CCI marker in a CCI.

A CCI may be restricted by specifying a role or the list of the objects in the CCI. In limiting a CCI to a particular role, a client may replace a role field that contains the *ANY* value with a specific role. Since modifying a CCI in this way limits how the CCI can be used, this is an unrestricted operation to the client. A client may also attach an object list to the end of a CCI. The object list restricts the activity of an identity to this specific set of objects, and is also an unrestricted operation to the client.

3.6.4.2 Identification

When an operation is invoked on an object, the manager of the object is responsible for determining if the client requesting the operation has the authority to invoke the operation. Since authorization is based on identities, when the manager receives an invocation request, its first action is to determine the identity of the client invoking the operation. The identity is determined using the CCI marker sent to the manager as a parameter of the invocation.

Upon receiving an invocation request, a manager (M) obtains the identity of the client (C) by invoking the **ShowProcessBindings** operation on the client C's UID received in the invocation request. The active CCI marker received in the invocation is a parameter to this operation. The Process Manager of client C services **ShowProcessBindings** and returns two values: the ACI and the active CCI of client C. The active CCI of the client may contain a proxy CCI marker. In this case, the PM invokes the **ObtainProxy** operation on the process (P) that originally transferred the proxy CCI marker to client

C. Process P is identified by its UID found in the proxy CCI marker. The **ObtainProxy** operation is serviced by the PM of process P. The proxy CCI returned for process P's PM to client C's PM is then sent to manager M along with client C's ACI. A CCI proxy marker is never replaced by the actual CCI because this would prevent the client lending the proxy from revoking the proxy (which can be done simply by changing its name or by destroying it). [This is an obvious tradeoff between performance and security. Replacing the proxy CCI marker with the actual CCI would avoid the extra step to the second PM on later uses of the CCI.]

3.6.4.3 Access Authorization

Upon receiving the identity of a client invoking an operation on an object, the object's manager then determines whether the client is authorized to access the object. Access is authorized by the manager providing all of the following criteria are met:

- If the operation invoked is a *direct operation*, as designated when the object's type is created, the client must be a terminal interface process. This information about a client is available in the ACI returned from the **ShowProcessBindings** operation.
- If the operation may only be invoked by specific principals, the principal name from the ACI of the client must be one of them. The ability to define the exact principals that may access an object may also be designated when the object's type is created. This is independent of the contents of the access control list for a particular object. For example, only the Authentication Manager may invoke the **SetProcessBindings** operation on a process object, regardless of on whose behalf the Authentication Manager is executing.
- If there is an object list in the active CCI of the client, then the object on which this operation is invoked must appear in this list. This requirement allows a client's activities to be explicitly restricted to specified objects based solely on its contextual identity.
- The CCI of the client must appear in the access control list (ACL) of the object. This is the primary means of controlling access to objects. Unlike the first two criteria, the access control list may change after the object is created to allow flexibility in specifying who may use the object, in what capacity, and with what operations.

We begin by introducing the *type role* associated with each type, which intuitively corresponds to the privileges clients may have to use objects of the type. We then examine access control lists.

3.6.4.3.1 Type Roles A list of type roles is defined for each type when the type is created. The purpose of these type roles are to define the privileges that roles take on

for all objects of the type. The System Control defines the type roles when the type is created.

A type role is of the form: (*Class : Role : List of operations*). The list of operations are a subset of the operations defined on the object's type, and will therefore vary depending on the type. Roles are divided into two classes: *nondiscretionary roles* and *discretionary roles*. Operations for which nondiscretionary (discretionary) control is desired are placed by the System Controller in nondiscretionary (discretionary) type roles. A role's class determines who has the privilege to place entries in an access control list for the role; only the System Controller may place an ACL entry in the ACL for a nondiscretionary role, and only the Controlling Group may place an ACL entry for a discretionary role. This relationship is described in detail in the next section on access control lists. An example of the type roles for a type are:

Class	Role	Operations
Nondiscretionary:	Controller	:ModifyNondiscretionaryACL
Nondiscretionary:	ControllingGroup	:ModifyDiscretionaryACL
Discretionary:	Reader	:Read, Display
Discretionary:	Writer	:Write
Discretionary:	Monitor	:ShowStatus
Discretionary:	Owner	:Read, Display, Write, ShowStatus

The type roles for a generic object might look like this:

Class	Project	Operations
Nondiscretionary:	Controller	:ModifyNondiscretionaryACL, :ModifyNondiscretionaryIACL
Nondiscretionary:	ControllingGroup	:ModifyDiscretionaryACL, :ModifyDiscretionaryIACL
Discretionary:	Creator	:Create
Discretionary:	{ other roles }	: { other operations }

3.6.4.3.2 Access Control Lists An access control list is associated with each object. The ACL is used to authorize access to the object. It consists of a list of identities. Each identity in an ACL will match one or more contextual client identities. An identity entry is of the same form as a CCI, only without an object list, resulting in the following form: (*Principal : Project : Role*). Unlike a CCI, the Principal and Project fields of an ACL entry may contain a wild card character "*" for character matching (as in UNIX). Note that the wild card character does not match "NIL". Examples of identity entries are the following:

Principal	Project	Operations
*:	System	:Controller
Jones:	C2.*	:ControllingGroup
Jones:	C2.*	:Monitor
Andrews:	System	:Monitor
Vinter:	C2.evaluation.*	:Reader, Writer, Monitor, Owner

The meaning of the last identity entry in this example is that principal Vinter, when working within the C2.evaluation project, or any project nested within that project, has the privileges of an owner, a reader, a writer, or a monitor with respect to the object.

Entries in the ACL are divided into two groups on the basis of the *class* field of each entry. The first two entries in the example above contain roles of the nondiscretionary class, while the last three entries contain roles of the discretionary class. This division is provided to control the modification of an ACL. Every object has two operations defined on it: *ModifyNondiscretionaryACL* and *ModifyDiscretionaryACL*. The *ModifyNondiscretionaryACL* operation is used to add or remove ACL entries containing Nondiscretionary roles, while the *ModifyDiscretionaryACL* operation is used to add or remove ACL entries containing discretionary roles.

This discretionary/nondiscretionary role division is consistent with the philosophy espoused in the security policy that certain operations on an object (such as the privilege to change who can access it) require more control than other operations. The security policy defines the controlling group of an object as the set of users that can modify who can access the object. The example above supports this idea by defining a ControllingGroup role that allows users acting in that role to make additions and deletions to discretionary roles. The security policy designates the System Controller as the only user able to modify the controlling group of an object. This control is achieved in the example above by making the ControllingGroup a nondiscretionary role, and defining the Controller as having the privilege to modify the nondiscretionary roles.

3.6.4.4 Initial Access Control Lists

The access control list of a newly created object is determined by an initial access control list (IACL) associated with the object's type. The IACL is intended to automatically set the ACL of a newly created object in a way that is dependent on who its creator is and how the object will be used.

The initial ACL consists of a set of pairs of the form (CCI guard, identity list), and is interpreted in the following way: if the client's identity matches the CCI guard, then add the associated identities to the ACL. The following is an example of an IACL pair that generate the ACL above if the object were created from CCI Vinter:C2.evaluation:

CCI guard: * : C2*

Principal	Project	Role
*:	System	:Controller
Jones:	C2.*	:ControllingGroup
Jones:	C2.*	:Monitor
Andrews:	System	:Monitor
⟨ creator ⟩: ⟨ creator ⟩* :Reader, Writer, Monitor, Owner		

(the ⟨ creator ⟩ value indicates that the value used is taken from the corresponding field of the client creating the object)

The initial ACLs are part of the generic object of a type. The ability to change initial ACLs is controlled in much the same manner as ACLs, with the identities divided into nondiscretionary and discretionary roles. The ModifyNondiscretionaryIACL and ModifyDiscretionaryIACL operations are used to modify the nondiscretionary and discretionary portions of IACLs.

3.6.4.5 Discretionary Control Assurance

Discretionary access control routines that are embedded in Cronus managers are automatically generated from a type (interface) specification. In the specification, the type developer simply provides the rights, or privileges, associated with operations. This is used to generate a mask against which access control checks are made. The implementation of the access controls have a high degree of assurance because of their generation from a single source (that can easily be controlled during the software development cycle).

However, an ingenious implementor can circumvent these checks easily by having the manager, at run time, modify the software that makes the check or the software that sends control to the access check routine; modify the access control list or privilege mask; or modify the client bindings obtained from the client's Process Manager. Thus, it is difficult to protect the trusted access control check software from the actions of the untrusted manager software since they execute in the same process address space.

One solution to this problem is provided by hardware rings. Hardware rings define concentric circles of memory access privilege, where software executing an inner has access to all data in its ring or any outer ring. Rings not only control data access, but also control flow. Inner rings may not call outer rings. Outer rings issue subroutine calls to inner rings, but only through predefined *gates*.

Rings can be used to protect discretionary access controls by placing these mechanisms in an inner ring and the remainder of the untrusted manager software in an outer ring. The efficacy of this approach depends on several requirements of the underlying COS and hardware:

- There must be at least two rings available to the application developer across which a manager may be configured.

- Control flow must be supported from inner rings to outer rings. This requirement is not common among ring-based systems. However, techniques for initiating lightweight processes from inner rings to execute in outer rings have been developed [Schell and Tao 84].

If these two requirements were satisfied, it would allow message parsing and access control to be placed in an inner ring. Should access control succeed, the inner-ring can dispatch a light-weight process to execute the operation and transmit the message parameters. When the operation completes, the light-weight process can signal to the inner-ring software of its completion and the location of the results. Control can continue in the inner-ring to forward the results of the invocation to the client.

Chapter 4

Formal Methods

4.1 The Formal Top-Level Specification

4.1.1 Overview

The Formal Top-Level Specification (FTLS) is a transcription into a formal language of the design of SDOS that was developed in the previous chapter. It is an abstract description of SDOS. Detail that was included in the functional description will generally be reflected in the FTLS, and vice-versa. Of course, neither the functional description nor the FTLS contain the same degree of detail that the final, coded version of SDOS will, but each is intended to present a complete view of the system at a particular level of abstraction.

The formal language chosen for the FTLS is Gypsy [Good et al. 78]. Gypsy is a Pascal-style programming language in which specifications can be stated as embedded assertions. Pre- and post-conditions for procedures and functions are particular cases of embedded assertions, which, in Gypsy, can be exported and used in the proofs of other procedures and functions. Gypsy, unlike Pascal, allows expression of concurrency. Concurrent procedures may communicate through shared buffers, and embedded assertions local to each procedure may express properties of the history of communication with each buffer.

The Gypsy specification expresses the entire SDOS as a cobegin of processes, each process representing a single host. Hosts will then execute concurrently. Communication between hosts is expressed as Gypsy buffers shared between them. No attempt has been made to formally specify the network communications in greater detail, and therefore, the semantics of Gypsy buffer operations have been accepted as part of the specification.

Each SDOS host is then in turn expressed as a cobegin of local processes. It is examples of these local processes which we have included in this report. The local processes include, at a minimum:

- A kernel, composed of a number of interacting entities needed for supporting efficient, object-oriented communication between SDOS entities;
- A file manager, responsible for creating the abstract type "file";
- A catalog manager, responsible for translating user-friendly symbolic names into the unique identifiers (UIDs) which serve as names used by the system;
- An authentication manager, responsible for login and logout of users;
- A trusted interface process (TIP), responsible for the terminal interface between the human user and the system.

The kernel shares a pair of Gypsy buffers (representing 2-way communication) with every other process on its host. Processes other than the kernel share buffers only with the kernel, i.e., they cannot communicate directly between themselves.

The Gypsy specifications of the FTLS are presented in Appendix A. There are comments that accompany each FTLS component. However, since reading code, even commented code, is difficult, we have given an informal introduction to the FTLS in this section. Each subsection here gives an overview of the algorithm of part of the system, discusses security tradeoffs, and informally argues that the Gypsy design is secure. The TIP and the authentication manager are covered together in one section, due to their close interrelation. The four subsections of this chapter, "Kernel", "File Manager", "Catalog Manager", and "Authentication", correspond to the sections A.4 through A.7 of Appendix A. In this chapter the kernel is described first because of its central importance, while it is listed last in the Appendix because its code must be loaded into the Gypsy system after that of the other components.

4.1.2 The Kernel

4.1.2.1 Introduction

The kernel is the key SDOS component that is necessarily part of every host in the network. We will discuss the kernel design by decomposing it into the following entities:

- Message switch
- Security Data Base (SDB)
- Object Data Base (ODB)
- Locator
- Process Manager
- Process Table

In Gypsy, the interaction between the above listed objects has not been modeled by buffer operations. Instead the kernel is treated as an integrated unit and the interactions are function calls. But viewing the interactions among the components as synchronous message passing is identical to the integrated (undecomposed) Gypsy model. Each

function call and each return of control to the function's caller is considered to be a synchronous message passing event. The synchronous message passing model is a more versatile tool for purposes of detailing the model and hence it is the vehicle of choice for the following sections of this report.

4.1.2.2 The Security Database

On each MLS host, there will be a collection of data needed for enforcing the mandatory security policy. This collection is called the Security Database (SDB). A host's SDB contains an entry for each entity residing on the host, and possibly for some remote entities as well. The local entities that must be in the collection include all objects, generic type objects, and all processes. The remote entities that must be in the local SDB include some subset of the SDOS hosts. An entry in the SDB will contain:

- a security label;
- a boolean indicating whether the entity is MLS;
- for the generic object of a type: whether local managers for the type exist, whether they are MLS, whether they are running, and where their executable code can be found;
- for a local replica of a replicated entity: the number of replicas which exist in the entire system.

In the actual system design the SDB may be organized into separate tables for hosts, objects, and types, but in the Gypsy specification we have placed all SDB data in a single table indexed by UIDs. The SDB entry associated with an entity is stored according to the entity's UID, and the entry contains the data described above.

Although security-relevant information may also reside in other system components, the SDB is the primary location for the information, and updates are not complete until the SDB data has been modified.

The SDB defines the following operations, each of which is an invocation on the local host:

1. CreateSDBEntry - generate a new, unique UID for a non-replicated entity, and create an entry for it in the local SDB.
2. RemoveSDBEntry - destroy the entry for the given UID, including all replicas on other hosts.
3. ReadSDBEntry - return the entry in the local SDB for the given UID to the client.
4. LocateUid - returns information pertaining to existence of the object and object manager.

5. `ModifySDBEntry` – modify the entry for the given UID and update all replicas.
6. `ReplicateSDBEntry` – if an entry for the given UID already exists on some remote host, create a copy of its entry locally, and increment the global information on the number of replicas.
7. `DereplicateSDBEntry` – if an entry for the given UID exists on the local host, and the entry is replicated, destroy the local entry and decrement the global information on the number of replicas.

A UID will exist in the SDB if and only if at some previous time that UID was created or replicated on this host. Concurrency control for replicated UIDs has been largely removed from the SDB and placed into other components. Certain assumptions also simplify the concurrency control problem. As required explicitly in the formal model, concurrency control must ensure that the label in an SDB entry reflects any change to its value initiated by events which could potentially influence the present value. Replicas on different hosts therefore must have the same label if the most recent `ModifySDBEntry` operation has been propagated to every host with a replica of that SDB entry. The following assumptions and requirements are sufficient:

- Concurrent `CreateSDBEntry` invocations cannot conflict, since the UIDs they create are assumed to be different. The method used in Cronus for ensuring this is to include a field in each UID which encodes the name of the host on which the UID was created; this method is adopted for SDOS. Therefore concurrent creates must produce different UIDs. A UID containing host field *H* does not necessarily reside on the host named by *H*, since replicate and dereplicate operations may have caused its migration.
- The message switch must ensure that each `ModifySDBEntry` operation, and in fact each operation updating replicated information (`Remove`-, `Modify`-, `Replicate`-, and `DereplicateSDBEntry`), is propagated to all hosts with replicas of the UID. This includes storing for future delivery all updates for currently unreachable hosts.
- Clients invoking `ModifySDBEntry` operations must ensure that they do not conflict with other modifications to the SDB. For example, it is the responsibility of the System Manager to ensure that no two `ModifySDBEntry` invocations made by him for the same UID are in progress simultaneously. Other than invocations made by the System Manager, `ModifySDBEntry` invocations by different kinds of client should not arise, since these kinds of clients are few in number, and they update disjoint sets of SDB entries. For example, updates by the authentication manager for terminal interface processes (TIPs) can never conflict with updates of type information by the process manager, since no type object is a TIP.
- The other operations which can update replicated entries (`Remove`, `Replicate`, and `Dereplicate`) do not conflict since their updates commute. For example, if `ReplicateSDBEntry` and `DereplicateSDBEntry` are concurrently invoked on UID

U on different hosts, every host will eventually be required both to increment and decrement its replica count in the SDB entry for *U*, resulting in no change. Such a simple minded scheme has limitations. If hosts on which the copies exist dereplicate simultaneously, Dereplicate will have the same effect as a Remove operation. Any manager whose operations may depend on the intermediate results of the commuting operations must enforce its own concurrency control on the use of the SDB operations.

The data in different SDB entries may be of differing sensitivities, and the SDB will respond to requests for service at many security levels. Therefore it must be an MLS entity. We considered several schemes for implementing security in the SDB. The schemes differ according to whether the UIDs and the entries associated with them are public information. A secure implementation of the SDB will be described for two main cases. As a first approximation to demonstrating that each implementation is secure, we will associate a level with each piece of data in the design, and enforce Bell-LaPadula-like rules for flow of data. This approach is useful for discussion; a more precise demonstration of security will be given later.

The following discussion gives rules for controlling those information flows through the SDB which would violate mandatory security. However, all of the operations on the SDB will additionally be constrained for other reasons, e.g., by rules of the configuration policy. In both principal schemes, changing the data in the SDB has security consequences other than just information flow: for example, security is violated if an UNCLASSIFIED client is able to rewrite every entity's label to UNCLASSIFIED. In either scheme, the following additional rules hold:

- An existing SDB entry may only be modified by the System Manager (to change labels manually), by the System Certifier (to make a single-level entity MLS), by the Authentication Manager (to change the label of a terminal interface process) or by the process manager (to change current information about generic type objects).
- An SDB entry may only be removed by the object database or by the process manager, in which cases the SDB operation is the by-product of the complete removal of some entity from the system.
- An MLS entity may only be created by the System Manager (for example, to add new MLS hosts to the system), or by the process manager (to start up multi-level processes).

In each scheme, we sought to make the information-flow security valid independent of assumptions about the SDB's clients and about the configuration policy. This maximizes the independent verification of components of the system.

4.1.2.2.1 Scheme I Suppose that the levels associated with the existence of an entity, and the security data in the SDB for the entity, are both *HostLo*, i.e., the meet

of all levels in this host's label. (We assume that *HostLo* itself is a level in the host's label). In other words, knowledge that some UID exists, knowledge that the level of that UID is *l*, knowledge that the UID is replicated, etc., are all publicly available. We see immediately that never will reading any data in the SDB compromise information, and that writing the SDB may only be initiated by requests at *HostLo*. Rejecting either request on the grounds that the UID does not exist will not disclose any information which is not already public.

To prevent the system from becoming unwieldy, it must be possible to create entities via requests at any arbitrary level. However, if create requests are permitted at some level *l* which is greater than *HostLo*, then information of sensitivity *l* will be written into the publicly available attributes of existence and level. This is a channel through which information can be downgraded. We can make the channel unusable, though, by requiring the following:

- A request can never be made to create a particular UID. Each request to create a new entry in the SDB generates and returns a UID not in use. The pool of possible UIDs is assumed to be effectively infinite, so that requests to create UIDs are never denied. Then a requestor can never learn about previous create operations at other levels by being told that "UID already exists".
- New UIDs are generated randomly, or pseudo-randomly with an algorithm which cannot be reproduced by software outside the kernel. The algorithm may be an encryption using a hidden key. Then a requestor can never deduce from the UID returned in response to a create request which other UIDs are currently in use. Once a UID is removed from the SDB, it can be reused in another create operation. But, it is no more or less likely to be used than any other available UID. The mechanism that generates the UID has to intervene this regeneration in some pseudo-random fashion with generating fresh UIDs so that no client can predict what UID would be created next.

These two requirements ensure that although the action of the SDB during a create operation will depend on previous creates at higher levels, the dependence cannot be exploited to downgrade information.

A request to remove a UID entry from the SDB, however, will also take information at the level of the request and put it into the publicly available existence and level attributes. Because the UID being removed must be specified in the request, this opens an exploitable channel: a pattern of existing UIDs can be selectively removed by a high-level Trojan Horse, and this pattern will be publicly visible. To prevent this, we must require all remove requests to be issued at *HostLo*.

Similarly, requests to replicate or dereplicate SDB entries must both read and write the SDB. Therefore, these operations must also be initiated at *HostLo*. It is these severe restrictions on modifying the SDB entries which form the principal drawback of this scheme.

4.1.2.2.2 Scheme II Rather than treat the existence of an entity and its SDB entry as public, in this scheme we treat them as potential containers for sensitive information. We will treat each entry as though it might potentially hold information at any level accessible to the entity with which it is associated.

An arbitrary label may be either single- or multi-level. For any label, define two levels: the *meet level*, which is the greatest level dominated by every level in the label, and the *join level*, which is the least level dominating every level in the label. For a single-level entity, its level equals both its meet level and its join level. In this scheme, the information about an SDB entry is as sensitive as the entity's meet level. Therefore, it will not be publicly available in general.

Under scheme II, there is a clear problem. The existence of higher level objects cannot be revealed to lower level clients. How then can a lower level client invoke an operation on a higher level object? For example, a secret client wishing to write to a top-secret file must first know the file's UID. But the fact that the file is in existence is top secret information. Each local SDOS may be able to hide this fact, forcing the client to invoke the "write-up" operation blindly. However, a DOS which does not perform unnecessary operations will be forced to reveal the existence of the file when forwarding the "write-up" operation to the manager on the host on which the file resides. (see section 4.1.2.5).

The solution we have chosen for all invocations involving such "up" operations is for the client to set a bit in the message indicating to the local message switch that this message is to be treated as an "up" operation. The message switch would then acknowledge receipt of the request by the generic reply "will do my best". The message switch, would ensure that the client receives no other acknowledgement which would divulge information about the existence of this higher level object, and having located the object and determined its level from the SDB, would proceed with the invocation as if it were at the level of the object itself.

As in Scheme I, we will assume that the pool of UIDs is effectively infinite.

The basic SDB operations are implemented as described below. In this scheme, a distinction must be made between cases where the "up" bit is set and otherwise. Also we will talk of "dominates" as being decomposable into a two parts: a strictly "greater than" part and an "equals" part.

1. Create - The creator supplies the label of the new entity as a parameter. The create request will fail only if the level of the invocation is not dominated by the meet level of the new label. This reveals no information, since the client already knows both the level of the invoke and the label of the new UID. In all other cases, the create operation succeeds, and returns a UID not currently in use. The UID is chosen randomly or pseudorandomly as in Scheme I, so it reveals no information about previous create invocations. The label recorded in the SDB is the label supplied by the client. The "up" bit is of no relevance.

2. Remove – If the entity exists and its meet level equals the level of the invocation, or if the meet level dominates the level of the invocation and the “up” bit is set, the SDB entry is in fact removed and the invocation returns a successful acknowledgement. Otherwise, the invocation fails. If the entity exists and if the level of the invocation is greater than the entity’s meet level, the error message “no permission” is returned. In this case, the client can potentially know whether this UID has been removed and therefore learns nothing when the error message implies that it still exists. If the entity does not exist or its existence should not be revealed, an error message “entity does not exist” is returned.

The level of the reply for this and the remaining operations is determined as follows: if the “up” bit is not set, or the error is the case of “no permission”, the reply is at the level of the invocation. If the “up” bit is set and the error is the case of “entity does not exist”, the reply is at *SysHi*. Otherwise, it is at the level of the object.

3. Modify – The security restrictions in this case parallel those for the remove operation. Of course, there are added configuration rules to be enforced such as: only the system manager and the authentication manager can successfully invoke this operation.
4. Read – If the entity exists and either the level of the invocation dominates the entity’s meet level, or the “up” bit is set and the meet level dominates the invocation level (“read-down”) the operation succeeds returning the SDB entry. In all other cases, either the entry does not exist, or its existence should not be revealed, the operation will fail with an error “entity does not exist”. From this message the client cannot deduce whether the entity does not in fact exist, or whether it has been previously created at an inaccessible level.
5. Replicate or dereplicate – The security restrictions in this case parallel those for the remove operation. If “replicate”, the UID must not exist locally; if “dereplicate”, it must exist locally else the invocation will fail with the error message “no permission”. If the UID does not exist to be replicated, or its existence should not be revealed, the invocation will return an error “entity does not exist”.
6. LocateUid – The security restrictions for this operation parallel those for ReadSDBEntry. The operation differs from ReadSDBEntry in the fact that it returns existence information about the object and an object manager capable of servicing the request. Capability is determined by the level of the object manager with respect to the level of the object and the level of the invocation. Single level object managers must be at the join of the object level and the invocation level. MLS managers must span the level of the invoke and the level of the object. Locate operations, though primarily interested in the object’s location, would use the additional information on object managers as an aid in optimizing the choice of a host to service a particular invocation on the object.

One disadvantage of Scheme II is that the error message “does not exist” does not

always mean that an SDB entry does not exist. It may be that the entity did exist and that either the "up" bit was not set in the invocation or the entity's level was incomparable to that of the invocation. Also all "up" operations are totally blind. However, operations conducted entirely at the level of an entity will proceed exactly as though there were no MLS constraints on the system.

A more serious problem arises in sending a message to an entity at a higher level but not on the local host. The remote entity cannot normally be located by broadcasts of messages at the client's level, since the remote SDBs will claim the entity does not exist. Therefore the "up" bit has to be set for the locate. But, if the remote entity's level is higher than the maximum level of the client's host, its location can never be securely found because the client's host cannot handle the reply from the Locate operation.

Therefore, there is a trade-off between Scheme I and Scheme II: either deleting will be an operation which must be invoked at the lowest security level, or special features must be introduced to implement "up" operations. The discussion of the locator in section 4.1.2.5 shows that Scheme II is workable even for the "up" operations. Therefore, our treatment of other kernel components will assume that Scheme II is being used, as we believe its problems are easier to live with than those of Scheme I.

4.1.2.2.3 Alternate Schemes There are alternatives both to Scheme I and Scheme II. Each alternative has drawbacks which we felt made them unsatisfactory.

One approach is to treat the data in an SDB entry as though it were as sensitive as the request to create it. This is a more "accurate" approach than Scheme II, since in fact the creator of a UID knows that the UID has existed at some time, even if the entity it names is classified at a higher level. For example, client *C* at level *c* may create object *O* at level *o*, where *o* dominates *c*. The fact that *O* exists at the moment of its creation is information at level *c*, even though the content of *O* is information at level *o*. Scheme II effectively upgrades the level of the SDB entry, and permits higher-level clients to remove *O*. Therefore, in Scheme II, *C* cannot know in the future whether *O* exists.

This alternative approach requires that the SDB remember the level of the creator of each entry, since it may be different than the level of the entity itself. The creator's level may be stored either in the SDB, or perhaps as part of the UID itself. This approach is feasible for implementing security. However, it entails extra storage requirements for the creation levels. It also has the same problem with "up" operations as Scheme II, if it is assumed that UIDs of entities might be used by clients at levels not dominating the creator's.

Another alternative seeks to avoid the drawbacks of both Scheme I and Scheme II, by using any of several hybrid approaches: Let the existence of a UID be public information, as in Scheme I. This allows locates to succeed always. Let other data associated with the UID, such as level, and/or replication count, and/or the existence of any ODB entry for the UID, be kept at the entity's level. Then this data can be deleted at the creator's

level, saving storage space, even if the UID itself cannot.

The "advantages" of these hybrid alternatives are deceptive. First, they make it possible to locate UIDs which are no longer associated with any entity. Attempts to fix this by creating new public attributes to indicate "has been deleted" (basing locates on existence of an ODB entry, for example, rather than the SDB) reintroduce the difficulty of Scheme I: deletes cannot be invoked at the entity's level. Second, replicate and dereplicate operations, which must both read and write components of the SDB (and ODB) entries, cannot be carried out at a single level. Therefore, these hybrid schemes are unworkable.

In yet another variant, the client may choose whether a new UID being created is public or classified at the level of the entity. So some SDB entries are treated as in Scheme I, some as in Scheme II. This variant allows locates for "write-up" operations on public UIDs to proceed entirely at the level of the invocation. However, since some UIDs may be classified, a mechanism for handling locates in Scheme II must still exist, and because it cannot be known in advance whether a particular UID is public, that mechanism will be invoked every time a UID cannot be located at the client's level. This variant offers some small advantage, but because it greatly complicates the locate algorithm, we have not considered it further.

4.1.2.3 The Object Database

The object database and manager (ODB) implements an internal representation of data objects. This internal representation is used by many of the SDOS managers to create their own abstract data types. The ODB keeps the data object representations on stable storage, and therefore must use the services provided by the host's COS. This interaction with the COS is not explicitly represented in the Gypsy specification; rather, the stable storage is represented in the specification by local variables.

To maintain security, the ODB must manipulate the contents of the SDB. The SDB is also part of the kernel, and so the ODB may make direct calls to it. The ODB uses the SDB to record a security level for every object in the ODB; this level represents the sensitivity of the object. We will assume that no object in the ODB has a label which is multi-level. Multi-level objects, such as the SDOS catalog, can be constructed by MLS managers, but the kernel will not provide any special support for them.

The ODB implements seven operations. For those operations which never involve potential "write-ups" to the ODB, success will depend on whether the appropriate SDB operation will succeed when invoked at the level of the ODB's client. In these cases, security is straightforward, as all messages will be passed at exactly one level and will never interact with other invocations. On the other hand, operations which may require a "write-up" to the ODB (`WriteODBEntry`, `RemoveODBEntry`, `CopyODBEntry`), will need to know whether a UID exists at all, not just whether it is visible at the client's level. Therefore, since SDB security Scheme II is assumed, these operations will be required to set the "up" bit in the `ReadSDBEntry` invocation and proceed with the

remainder of the invocation at the level of the reply from the SDB. Note: the ODB will set the "up" bit in its communications with the SDB only if the client has set the "up" bit in the original invocation. Security depends on the ODB not revealing any more about SDB entries than the SDB itself would if queried at the client's level.

Since ODB operations generally make secondary invocations on the SDB, we will distinguish the original invocation by referring to its level as the client's level.

1. **CreateODBEntry** – create a new UID of a particular type, and return its UID to the client. This operation invokes **CreateSDBEntry** at the client's level and with the same parameters. It succeeds if and only if the SDB invocation does, and creates a local ODB entry.
2. **RemoveODBEntry** – destroy the data associated with the given UID. This operation invokes **ReadSDBEntry** at the client's level. The "up" bit is set if the client had set it in the original invocation. If the object's level equals the level of the successful reply from the SDB, the ODB entry is deleted. If the SDB entry is replicated, then **RemoveSDBEntry** is invoked locally at the level of the SDB's reply and **RemoveODBEntry** is invoked on all other hosts, else **RemoveSDBEntry** is invoked locally. If the client's level is strictly greater than the object's, the invocation returns an error. Otherwise, if the object does not exist or its level is incomparable, the invocation returns an unsuccessful acknowledgement "object does not exist".
3. **WriteODBEntry** – associate new data with the given UID. The security restrictions in this case parallel those for **RemoveODBEntry**.
4. **ReadODBEntry** – return the data associated with the given UID to the client. **ReadSDBEntry** is invoked at the client's level. The "up" bit is set, if it is set in the original invocation. If the client's level dominates the level of the SDB's successful reply, then **ReadODBEntry** will succeed, and data from the ODB entry is returned to the client at the level of the SDB's reply. Otherwise, the client's invocation returns an error.
5. **CopyODBEntry** – copy the data associated with the given object into another object. This operation is provided to reduce the number of message exchanges between the kernel and the various object managers. Suppose it is invoked by a client at level c with UID U_1 as the source and UID U_2 as the target. **ReadSDBEntry** is invoked first at the client's level for UID U_2 . The "up" bit is set if the client had set the same in the original invocation. The remainder of the transaction is at the level of the reply from the SDB. Upon successful return from the SDB, **ReadSDBEntry** is invoked on the SDB for UID U_1 . The **CopyODBEntry** invocation returns an error if and only if either of the **ReadSDBEntry** invocations fail or l_1 strictly greater than l_2 , or c strictly greater than l_1 or c strictly greater than l_2 . The error message is at the level of the client's invocation. On the other hand, the transfer of data from U_1 to U_2 actually takes place if both source and target exist and l_2 dominates l_1 and l_1 dominates c . Note that the "up" bit must be set

by the client in the original invoke to exploit the strictly greater than component of dominates.

6. **ReplicateODBEntry** – replicate the ODB entry associated with the given UID. **ReplicateSDBEntry** is invoked at the client's level. The "up" bit is set if the client had set the same on the original invocation. If this fails, the error is echoed to the client. Else, an **ReadODBEntry** is broadcast at the level of the SDB's reply. The reply to the **ReadODBEntry** is then used to create a local ODB entry. A successful acknowledgement is returned at the level of the SDB's reply.
7. **DereplicateODBEntry** – dereplicate the data associated with the given UID. **DeReplicateSDBEntry** is invoked on the SDB at the client's level. If this returns successfully, the local ODB entry is deleted and a successful acknowledgement is made at the level of the reply from the SDB. Else the error message from the SDB is echoed to the client.

The ODB enforces no concurrency control. Ensuring that different replicas of an object are consistent is the responsibility of the object's manager.

4.1.2.4 Message Switch

4.1.2.4.1 Overview The Message Switch is the component of the kernel responsible for routing messages in accordance with the security policy. The Message Switch has the same security label as the host on which it resides and is an MLS entity if and only if the host is an MLS entity. Messages could be of three types: 1) Direct IPC, 2) Invocation of abstract operations on objects, and 3) Response to messages either of the earlier types. The kernel also supports a multicast operation; multicasting is sending a message to many hosts (that are candidates by some evaluation scheme) in the system.

4.1.2.4.2 Routing The Message Switch, on receiving a client's request, does the following:

- By interaction with the Process Table (port information), determines if the request is local or external.
- If local then,
 1. decodes the incoming message to determine the client UID and the level of the request.
 2. Client processes cannot be trusted to correctly report their levels or even their UIDs in the appropriate message fields. So the message switch has to check the validity of these fields. The correct UID can be determined when the Message Switch reads the port information in the Process Table. The correct label can be determined from the SDB. The Message Switch has the option to either correct the wrong fields and proceed with the invocation or discard that invoke request. Since the relative trade-offs are unclear, we have chosen

to discard the incorrect invoke. For non-local messages, it is assumed that the remote message switch has already established the client's credibility.

The Message Switch checks with the SDB to determine if the level of the client's request is legal. This is accomplished by an exchange of messages with the SDB. The SDB has the same label as the Message Switch. The request to read the SDB is at the level of the client's request to the Message Switch.

Case 1 SDB returns an error or true level is not the level reported in level field of the message.

No action is taken on behalf of this invocation.

Case 2 SDB returns OK and true level is the level reported in the level field of the message.

As outlined earlier, the messages to be routed could be direct IPC, invocation of abstract operations on objects or responses to the same. The distinction is detected by the Message Switch by examining the *control* field of the message. The three scenarios are as described under:

Scene 1: Invocation on objects

If the "up" bit is set then the message switch replies immediately to the client. The reply is the generic acknowledgement "will do my best". In either case, the Message Switch determines (by checking with the SDB) if the object is available locally. If so, the UID of the manager for the type of object is determined (from the SDB). If the manager is not active (determined by interaction with the process table) then one is activated (by an invoke to the process manager). The manager is either MLS (includes the level of the client and object) or single level at the level of the invocation. Note: if the "up" bit were set, the invocation would be at the level of the reply from the SDB (which would be the level of the object; see section 4.1.2.2 on the Security Database for details) and not at the level of the client's invocation. (Note: Object managers are in general untrusted pieces of code. If the level of the client's request is less than the level of the object, a single level object manager at the object's level would not be able to respond to the client. This is the critical check on the "up" operation and explains the need to make invocations at the level of the object instead of the level of the client's invocation. So the only acknowledgement that a client can get is the generic reply from the Message Switch.) If the object is not available locally, the Message Switch transfers control to the Locator to determine the location of the object.

Scene 2: Direct IPC

The message switch by checking with the SDB/Process Table determines if the process is running locally. If the process is available locally, then its level is determined (from the SDB). If the level of the

destination process dominates the level of the message, the message is routed to the destination process. Else an error message 'object does not exist' is returned to the sender. If the process is not running locally the Message Switch transfers control to the Locator to determine the location of the object.

Scene 3 Responses to invocations

The only difference between this and the previous scene is that the message switch assumes (by virtue of the *control* field) that the destination address is available. Hence locates are not necessary. The messages are directly routed to the destination process. The level of the destination process must dominate the level of the message.

- If not local then,
 1. Determines the level of the message.
 2. If the level is not within the set of permissible levels of the Message Switch and cannot be upgraded to be with the set, no action is taken on behalf of that message.
 3. If the level is within the set of permissible levels of the Message Switch or has been upgraded to be within the set, the message is routed to the appropriate local component, either within the kernel, or a local process known in the Process Table. (The external messages may either be invokes, IPC, or responses to invokes routed to local processes as under the local case).

4.1.2.4.3 Multicast Details The issue of interest in the Multicast operation is the level at which the Message Switch broadcasts the Locator's request to locate an object. The situation of interest is easily demonstrated by an example. Let *A* and *B* be two hosts on the network. A client on host *A* at level *a* wants access to object *O* at level *b*. Now, host *A* is an MLS host, but host *B* is a single level host at level *b*. Further, let *b* not dominate *a*. If object *O* is not on host *A* then the Locator on host *A* has to broadcast a locate message. Now what should the level of the broadcast be? It cannot be *a* because host *B* can only handle level *b* requests. If the broadcast is at *b*, then users at level *b* can potentially determine a level *a* intent of interacting with object *O*; which renders the system insecure. The problem is best resolved by the following scheme:

- The multicast messages always go out at the level of the invocation.
- The set of hosts capable of receiving the invocation is determined. This is the target set for the multicast.
- If no host on the system responds positively to the request (single level hosts not at the level of the request would not respond at all) then an error message is sent to the client. A more elaborate scheme could call for the locator to determine the client's principal and invoke a potentially insecure locate at the principal's discretion.

Under Scheme II, discussed in the section on the Security Database, LocateUid invocations would fail to locate on remote hosts if the level of the object is higher than

the level of the local host. See section 4.1.2.5 on the Locator for more details.

4.1.2.5 Locator

4.1.2.5.1 Overview The Locator is responsible for locating UIDs that are not available in the local host (either the generic type is not supported, or the object does not exist). Locating an object means determining the UID of a host whose SDB has an entry for the object in question. When the Message Switch determines (by interacting with the local SDB) that the object is not available locally, it sends a request at the level of the original invocation to the Locator.

The Locator has a local cache where results of previous locate requests are stored. If an entry is found in the cache, then the Locator returns the corresponding host UID to the message switch. The level of the reply is that of the Message Switch's request. If there is no entry in the cache, the Locator sends a multicast request to the Message Switch. The "up" bit is set if the client had set it in the original invocation. The abstract operation invoked is `LocateUid`. This is serviced by the SDB's in the hosts targeted for the multicast. The level of the invocation is the level of the client's request. The positive responses to the multicasts are at least at the level of the object. (see section 4.1.2.2 for details of `LocateUid` operation.) These are relayed to the Locator by the message switch. Since more than one host could positively acknowledge the broadcast request,¹ the Locator collects responses till it finds a host that meets a minimal criterion for being the candidate. A likely criterion could be that the object and the manager are both available. It then makes an appropriate entry in the local cache. (See section 4.1.2.5.2 for cache details.) The host UID is then sent as the reply to the Message Switch. The level of the reply is the level of the relay from the Message Switch.

4.1.2.5.2 Cache Details The cache is an MLS entity. Each entry in the cache (the object UID in question and the UID of the host which met the criterion described in the previous section) is at the level at which the reply to the `LocateUid` invocation for that particular object. Suppose, for example, that Client *A* at level *a* tries to access object *B* at level *b*. Object *B* is neither available on the local host nor is there a cache entry for it. Hence a `LocateUid` broadcast is made. On receiving an acknowledgement that meets the set criterion a cache entry is made. Let the level of the reply be *r*. The cache entry for object *B* is at level *r*. Now, from the semantics of the `LocateUid` operation (see section 4.1.2.2 on the Security Database) it is clear that *r* is *a* if *a* dominates *b*, else *r* is *b*.

If there is a subsequent request by client *C* at level *c* to access the object, the response would depend on the relation between *c* and *r*.

case 1: *c* dominates *r*

¹The multicast operation has the option to force every host which receives the broadcast to reply to the request.

In this case the request at c can read the cache entry at r for object B . Therefore, there is no need to multicast a request to other hosts. The Locator would return the host UID on which object B exists.

case 2: c does not dominate r

The request at c cannot read the cache entry at r for object B . Therefore the Locator would have to go through the broadcast procedure outlined earlier.

Explanation:

The SDB does not give out existence data at any level, so neither should the cache. If the cache data were stored at the level of the object without consideration to the level of the invocation, then information about whether locates were needed or not can leak out in the ordering of responses to invocations. Since a cache entry has been made, future operations on that object would be handled faster than ones without cache entries. Given assumptions about handling of messages by hosts, this timing channel can be converted into a constraint on sequences of replies. The covert channel is detailed in the following example: As in the previous example, client A at level a accessed object B at level b on a remote host. A locate operation was performed and the cache entry for object B was made at level b . There is a subsequent request by client C at level c to access object B . Further let level c dominate level b and level c not dominate a . Now clients at c can potentially read SDB entries at level b . Hence they know that object B does not reside on their local host. Client C at level c sequences the invocation on object B with one on object $B1$ (which he knows is on the local host). Client C has also determined (by looking over previous histories) that there have been no requests at levels dominated by c to access object B . Also all hosts on the network behave deterministically, in handling requests in the order they were received. Now studying the order of the responses to two invocations client C can infer an earlier invocation on object B at a level not dominated by c .

A similar covert channel can be demonstrated for cache entries made at the level of the invocation without heed to the level of the object.

Therefore cache entries should be labeled at the join of the invocation level and the object level. This is precisely the level of the reply from the LocateUid operation.

4.1.2.5.3 Migrating Objects If an object has migrated since the last locate, then the cache information will be faulty and an invoke based on this information will fail. The failure will be thrown back to the initial message switch, which in turn would force a broadcast to locate the object. The cache may contain several entries at different security levels for a single object. An optimum protocol has to be set up to read the cache to determine the location of the object (this could involve time stamps, weights to different levels of information, etc.) Also, once a new location for the object M has been determined, say by invoke at level k , then all previous entries in the cache for object M that dominate level k must be deleted. We have not pursued this any further.

4.1.2.6 Process Manager

4.1.2.6.1 Introduction The following operations are handled in the process manager to provide the needed mechanisms to create and kill processes and also to modify their attributes.

- CreateProcess
- KillProcess
- ChangeActiveCCI
- DetermineClientId
- ShowProcessBindings
- SetProcessBindings
- ObtainProxy

Process bindings for MLS process are stored for every level that the process is active. Therefore requests to read or modify the process bindings would be treated as requests on a single level entity.

4.1.2.6.2 CreateProcess Rule governing *CreateProcess* operations:

- Client *A* at security level *a* can create a process *B* at security level *b* if and only if meet level of *b* dominates *a*.

An error message is returned, at the level of the invocation, if violating the rule is attempted.

The client requesting to create a process cannot pick its UID, as discussed in the section on the Security Database. The Process Manager does the following as part of the CreateProcess Operation: 1) issues a CreateSDBEntry request (at the level of the invocation), which in turn makes updates to the SDB and 2) makes an entry in the Process Table.

4.1.2.6.3 KillProcess Rule governing *KillProcess* operations:

- Client *A* at security level *a* can delete a process *B* at security level *b* if and only if meet level of *b* dominates *a*.

To service the request, the Process Manager does the following as part of the Kill-Process operation: 1) issues a DeleteSDBEntry request (at the level of the invocation), which in turn updates the SDB and 2) updates the local Process Table. The "up" bit is set in the invocation to the SDB if the client had set the same in the original invocation.

The level of the reply from the SDB is the level at which the entry is made in the Process Table. If the SDB returns an error, the error is echoed to the message switch. If the "up" bit were not set by the client, the error message would reach him. (see section 4.1.2.4 for details of "up" operations).

4.1.2.6.4 ShowProcessBindings Rule for *ShowProcessBindings* operations:

Suppose that client *A* at level *a* invokes an operation on object *B*. Let *MB* be the object manager for *B*.

- Process *MB* can invoke a *ShowProcessBindings* on Process Manager for process *A* at any level that dominates the level of process *A*.

The rationale for error messages is similar to the instances of ReadSDBEntry operation. A low level invoke to read bindings of a higher level process would lead to an error message 'process does not exist'.

The operation essentially involves reading the process table entry for the client *A* process. Since the entry for client *A* is at level *a*, the level of the invocation should dominate the level of the entry, for the read to be secure.

A process *MB*, therefore, can issue a *ShowProcessBindings* invocation on the process manager of client *A* at any *x* that dominates *a*. If *MB* invokes the operation on its own initiative, then the level of the invoke should dominate the level of the client *A*. The Process Manager's reply is at the level of the process *MB*'s invocation.

The "up" bit has no significance in this operation.

4.1.2.6.5 SetProcessBindings Rules governing *SetProcessBindings* operations:

- Only the System Manager and Authentication Manager can invoke this operation.
- The level of the process should dominate the level of the invocation.

Attempts to violate the above rules would result in an error message.

Since this is a basically a write operation, the level of the entry should dominate the level of the invocation. The Authentication Manager and System Manager can, therefore, issue a *SetProcessBindings* request at a level that is dominated by the level of the client process. The reply from the Process Manager is at the level of the client process.

To exploit the "strictly greater than" part of dominates the "up" bit will have to be set in the invocation. The reply from the Process Manager can never reach the process making the invocation unless it were MLS.

4.1.2.6.6 *ChangeActiveCCI* Rule governing *ChangeActiveCCI* operations:

- Client *A* can invoke a *ChangeActiveCCI* operation on itself at its own level.

All other attempts would result in an error message.

4.1.2.6.7 *ObtainProxy* Rule governing *ObtainProxy* operations:

- Process Manager *X* can invoke a *ObtainProxy* operation on the Process Manager of process *A* at a level that dominates that of process *A*.

All other attempts would result in an error message 'process does not exist'. The rationale for this is same as discussed for the *ReadSDBEntry* operation in section 4.1.2.2.

Client *A* at level *a* invokes an operation handled by manager *M1*. *M1* in turn has to request services of manager *M2* to complete action on the invocation. Now manager *M2* needs to determine the CCI of client *A* in order to enforce discretionary access controls. Client *A* has passed a proxy marker to manager *M1* which has been added to the manager's CCIs in the Process Table. This marker is passed as *M1*'s active CCI marker to manager *M2*. Manager *M2* invokes *ShowProcessBindings* on the Process Manager for *M2* with the CCI marker as a parameter. Process Manager for *M2* in servicing the request determines that the CCI is a proxy and therefore issues a *ObtainProxy* operation on the Process Manager for client *A*, which in general could be different from the that of *M1*. (client *A*'s UID and the CCI marker are visible to *M2*'s Process Manager.)

It is true that the operation of *ObtainProxy* would succeed at any level that dominates the level of the client *A*. But there is nothing to gain by making this correspondence different from level of the invoke. In fact, it would cause problems of "write-down" when replying to the client. Therefore, it is more prudent to have all exchanges at the level of the client's request.

4.1.2.7 Process Table

The process table is the data structure maintained by the process manager to store the process bindings for each active process on the local host. The table also has information about the assignment of ports on the local host for processes to communicate through.

The level of all process bindings information is at the level of the process. The process table, therefore, is an MLS entity. As stated earlier, for MLS processes, there

will be separate entries for each level at which the process is active. The port assignment information is HostLo.

The Process Manager accesses all other information in the table in response to invocations by other processes. (These are routed through the message switch).

4.1.3 The File Manager

4.1.3.1 Introduction

This section serves as an informal but exact description of the multi-level-secure file-manager.

The operations handled by the file manager are

- Openfile
- Createfile
- Readfile
- Closefile
- Deletefile
- WriteFile

The possible access modes are *read*, *write* and *readwrite*. *Writes* are exclusive², but *reads* are not.

In what follows the access mode *write* may be a blind write. This depends on the level of the client and the level of the file that he is writing into. The file-manager receives invocations at the level of the object, even though the original invocations may have been at a lower level with the "up" bit set. (See section 4.1.2.4 for details of Message Switch mediation for "up" operations). In these cases even though the file-manager acknowledges completion of the task, the client never learns about it. (The action of the file-manager would amount to a write-down which would be a violation of the security policy and hence would be disallowed by the message switch). Also, since the level of invocations on the file manager are at least at the level of the object, *readwrite* and *write* access modes will be governed by the same security considerations.

Write encompasses both overwrite and append. The ability to overwrite introduces integrity issues which can often be handled by assigning integrity levels to clients and objects. The security and integrity levels have been amalgamated in the security policy to form one set of levels. The terms 'level' and 'security level', used interchangeably in this section, refer to this amalgamated level. Discretionary access controls may also be used to alleviate these integrity problems.

The operation of *read* and *write* should be preceded by opening the file and followed by closing the file. The file-manager maintains data structures that indicate client id, file id, access mode and level of access for files that are open. (The level of access is the level of the kernel's invocation which may be different from the level of the client's original request.). The file-manager acknowledges successful and unsuccessful completions of all

²A client with write access excludes all other writers.

invocations. As explained earlier, these may or may not reach the clients who originated the invocation.

Before delving into details of file-manager operations, it may be appropriate to get an appreciation for the kinds of security issues that arise in the file-manager: The following discussion will serve as an illustration:

Example 1: Suppose client *A* is writing to file *X* at level *a*.

All invocations to the file-manager are at least at the level of the object. It is all right for users at levels that dominate *a* to know that the file is being written into. Since there can be no invocations below *a* no special security considerations are in play.

It would be a disservice to clients at higher level if they are not allowed to access objects when lower level clients are accessing the same.

Example 2: Suppose client *A* is reading file *B* at level *b*.

Writes cannot be allowed when another client (whose level dominates the new request) is reading the file. This is not for security reasons, but because the information being read will be constantly changing and would be an unacceptable hindrance to application programs and human users. In addition, the file being read may not be in a consistent state, since the writer may not have completed the update. But, the client cannot be told that the file is being read by a higher level client because this would be an overt channel.

The sections that follow try to outline the rules for each of the file-manager operations and outline the rationale for the way special situations are handled.

It may also be appropriate to note the following restrictions:

- All communications with the ODB are through the Message Switch as opposed to direct interactions with the COS.
- It is assumed that files are single level and not replicated.
- The discretionary access rights are fixed.
- There is one channel per client per object per level.

4.1.3.2 Operations

4.1.3.2.1 Openfile Rules governing *openfile* operations:

- Client *A*'s invocation at security level *a* can open a file *B* to *read* if and only if *a* dominates security level of file *B*.
- Client *A*'s invocation at security level *a* can open a file *B* to *write* if and only if security level of file *B* equals *a*.

The request to open translates into a ReadSDBEntry invocation on the kernel. Once the level of the object is determined, the invocation is checked against the above rules. If there are no violations, the invocation succeeds and an acknowledgement is sent at the level of his invocation.

Success of the invocation means different things for *read* and *write* access mode requests. For *read* requests, the first step is creating a ghost file at the level of the invocation. The ghost file would be a copy of the file that the client intended to access. Relevant data structures would be updated to map the client's read request onto this ghost file. For *write* requests, no ghost files need be created. In both cases data structures are updated to reflect the client's access to the object.

Any error message would depend on the nature of the infringement. The case of interest is:

- access mode is *write* and level of object does not equal the level of request.

Reply is the error message 'No permission'. Since the level of the request equals the level of the object there is no violation of security to acknowledge that denial was due to lack of permission.

4.1.3.2.1.1 Special Case - File already open File *B* is the object being accessed. The security level of file *B* is *b*. *C* is a client accessing file *B* at level *c*. (There may be other clients accessing file *B*). The current request is from Client *A* at level *a*.

- **Current client already has access to file at a different level**

client *C* = client *A* but $c \neq a$

Since there is more than one channel per client per object, the request is handled as a fresh request.

- **Client (with no previous access) issues open to read request**

client *A* $\neq$ client *C*

Since read requests are handled by creating ghost copies of the file, read requests would always succeed, regardless of whether the file is being accessed by another client in the read or write mode.

- **Client (with no previous access) issues open to write request**

1. *Request to open a file that is being accessed at least by one other client in the write/readwrite mode.*

There is no breach of security to inform the client that the file is in use.

2. *Request to open a file that is only being accessed by other clients in the read mode.*

Since all the read requests were handled by creating ghost files the true object is free to be written into. Therefore, the request would succeed.

4.1.3.2.2 Readfile Rule governing *readfile* requests:

- Client *A*, at security level *a*, can read file *B* if and only if there is an entry in the file-manager's tables to indicate that *A* has *B* open at level *a* to *read* or *readwrite*.

An error message is issued to clients attempting to violate the above rule.

4.1.3.2.3 Writefile Rule governing *writefile* requests:

- Client *A*, at security level *a*, can write to file *B* if and only if there exists an entry in the file-manager's tables to indicate that *A* has *B* open at level *a* to *write* or *readwrite*.

An error message is issued to clients attempting to violate the above rule.

4.1.3.2.4 Createfile Rule governing *createfile* operations:

- A client at security level *x* can create files at security level *y* if and only if *y* dominates *x*.

An error message is issued to clients attempting to violate the above rule.

The client cannot choose the UID for the file being created. The file-manager relays the UID created by the kernel to the client.

4.1.3.2.5 Deletefile Rule governing *deletefile* requests:

- A client of security level *x* can delete files at security level *y* if and only if *y* dominates *x*.

An error message is issued to clients attempting to violate the above rule.

4.1.3.2.5.1 Special Cases: File in use Client *A* issues a request at level *a* to delete file *B*, whose level is *b*. File *B* is being accessed by client *C* at level *c*.

Even though delete-ups are allowed by setting the "up" bit, the invocation reaching the file-manager would be at least at the level of the object. (see section 4.1.2.4 for details of "up" operations). Therefore, level *b* = level *a*.

If *C* is reading the file, then the actual reads are occurring on a ghost file. Therefore, deleting the actual object will not affect the read operation in progress.

If *C* were writing to the file, then the file would be deleted. There is no point in allowing continuation of the write operation because client *A* has demonstrated intent to delete the file.

4.1.3.2.6 Closefile Rule governing *closefile* requests:

- Client *A*'s invocation at security level *a* can close file *B* if and only if an entry exists in the file-manager's tables to indicate that *A* has *B* open at level *a*.

An error message is sent to clients attempting to violate the above rule.

4.1.3.2.6.1 Special Cases

- File being closed is a ghost.

This is the scenario when client closes a file that was open to read. The ghost file is deleted.

4.1.4 The Catalog Manager

The purpose of the catalog manager is to provide an abstract space of symbolic names for objects. The catalog manager functions by translating from an object's symbolic name to its UID. The association of a symbolic name with the UID to which it can be translated will be called an *alias*.

The internal name for each SDOS object, the name by which it is known to the kernel, is its UID. The UID is chosen by the kernel as an almost arbitrary string of bits, and therefore tends to be hard for users to remember. Users of the system will usually prefer to choose and to work with alphanumeric names: symbolic names tend to be more memorable than UIDs. The catalog manager provides the connection between these sets of names. The system-wide collection of aliases which relate symbolic names to UIDs is called the *catalog*.

For example, it may be desirable to let the symbolic name "bar" stand for the UID U_1 . The catalog manager will allow this alias "*bar* : U_1 " to be stored in the catalog and recalled.

The catalog is divided into a hierarchically-organized set of sub-catalogs called *directories*, which may be thought of as containers for aliases. These directories are the objects managed by the catalog manager: all invocations handled by the catalog manager are invocations on directories to create, destroy, lookup, or manipulate aliases. A symbolic name is translated by finding the directory in which it is stored, and returning the UID to which it is aliased. A complete symbolic name will comprise a list of identifiers, called the *directory path* or simply *path*. Each proper initial subsequence of the path is itself a symbolic name for a directory. By taking successively longer initial subsequences, the path gives the proper sequence in which directories should be searched, starting at the root directory, in order to find the directory containing the UID to which the symbolic name is aliased.

For example, let the symbolic name `"/usr/foo/bar"` be a directory path, where slashes separate the identifiers in the path. To successfully translate this symbolic name to a UID, the catalog manager must first search the root directory for the identifier `"usr"`. `"usr"` should translate to the UID of another directory, which is searched for the identifier `"foo"`. `"foo"`, in turn is translated to the UID of a third directory, which will yield an alias, such as `"bar : U1"`. U_1 is then the translation of `"/usr/foo/bar"`.

A UID may have more than one symbolic name as an alias. In other words, different directory paths may translate to the same UID.

The SDOS catalog manager has some similarities to its Cronus namesake. In addition to providing aliasing and lookup capabilities, it optimizes the lookup process by replicating some of the directories (and therefore aliases in those directories) at more than one SDOS host which is running the catalog manager. The SDOS catalog manager differs from the Cronus catalog manager in that a replicated directory need not have replicas at every host. In particular, a host will not contain a directory replica whose level is not in the level set of that host. The directory hierarchy forms a tree. As a practical matter, nodes of that tree which are closer to the root will be used more frequently in lookups and therefore are more likely to be replicated, but the catalog manager does not enforce any rules concerning which nodes must be replicated.

Also unlike the Cronus catalog manager, the SDOS catalog manager will not communicate directly with the underlying COS. Therefore, it must send messages through the kernel message switch in order to use stable storage provided by the COS. The directories will reside on stable storage, and operations invoked on the directories will generally result in requests made by the catalog manager to the ODB.

The catalog manager we have defined is an MLS entity. In other words, it will receive and securely reply to requests for catalog operations at more than one security level. The SDOS security policy therefore requires that the catalog manager prevent information about higher-level requests from affecting its behaviour at lower levels. It is possible to implement SDOS without an MLS catalog manager: copies of a single-level catalog manager would then be created interactively for each request or for each level of request. Although this scheme would work, the catalog manager services are used so commonly that the resulting system would pay a high price in efficiency.

Under the current design, the catalog manager is also able to deal concurrently with simultaneous invocations from different clients. This fact both decreases and increases the complexity of the manager's algorithm. While waiting for the kernel's response to an ODB request, for example, it is not necessary for the catalog manager to buffer new invocations which arrive. On the other hand, since an invocation on a directory will usually require intermediate steps which are secondary invocations on the ODB, the manager must store the intermediate states of processing for each such invocation.

Two problems must be overcome in implementing replicated directories. First, replicas of a directory must appear on different hosts under the same UID. A replication operation, `ReplicateODB`, is assumed to be provided by the kernel. This operation

allows an object on one host to be duplicated on another, with the duplicate object retaining the same UID. This is a secure operation if carried out at the level of the object being duplicated.

Second, replicas of a directory must be kept consistent. This is the responsibility of the collection of catalog managers rather than of the kernel. The catalog manager must record, for each directory D , whether that directory is replicated, and if it is, the name of the host on which the 'primary' replica resides. The catalog manager on this 'primary' host will be responsible for broadcasting and serializing updates to D . This method of concurrency control was chosen purely for simplicity. It has the obvious drawback that updates to D will be denied if the primary host fails.

4.1.4.1 Operations

The abstract operations defined and implemented by the catalog manager are as follows:

4.1.4.1.1 lookup This operation will attempt to return the UID of an object given a symbolic name for that object. Every lookup is an invocation on a given UID, which is the directory which serves as a starting point for the lookup search. The symbolic name is then a directory path relative to the starting UID. Since the catalog is hierarchically structured, any such relative lookup could also have been expressed as an absolute path starting at the root directory. Symbolic names which are expressed relative to a UID other than the root serve as a shorthand, to reduce the number of directories which must be visited during the lookup.

For example, let the UID of the root directory be U_1 , and let the UID associated with symbolic name `"/usr/foo/bar"` be U_2 , where the identifiers in the directory path are separated by slashes. Then an object, "file", in directory U_2 can be translated either by a lookup invoked on U_2 with parameter "file", or by a lookup invoked on U_1 with absolute pathname `"/usr/foo/bar/file"`.

Upon receiving the lookup invocation, the catalog manager attempts to read the ODB entry for the starting directory. If that attempt fails, the lookup fails and an error is returned to the client. If that attempt succeeds, the starting directory is searched for the first identifier of the relative path. If the directory holds no such identifier, the lookup fails. Otherwise, suppose that the UID to which the identifier is aliased in this directory is U . If the identifier is the final identifier in the symbolic name, then the lookup is complete and the UID U is returned to the client as the translation of the symbolic name relative to the starting directory. If not, the catalog manager attempts to continue translating the path. Continuing the translation is equivalent to a lookup invocation on UID U with a relative pathname which is the original pathname with the first identifier removed.

Rather than explicitly make this secondary invocation, however, the catalog manager tries to optimize the search by immediately reading object U from the local ODB. We

suppose that a child directory can often be found on the same host as its parent, and therefore this ODB request will often succeed. If it does not succeed, the catalog manager will explicitly make the invocation on object U , permitting the local message switch to locate a host on which U resides.

Lookup operations on behalf of several clients can be interleaved by storing a list of concurrently progressing operations. This list must record, for each operation, the depth to which the relative directory path has been translated.

A lookup may fail for security reasons at any stage of translation of the path. All ReadODB requests are sent to the kernel at the level of the original client's invocation. If a directory in the pathname cannot be read at this level, the ReadODB request will fail, and the lookup will be aborted. The client will be told of the abortion at the same level as the original invocation.

There is, in general, no restriction on the level of any directory. The level of a directory represents an upper limit on the sensitivity of the aliases it contains. These aliases may be either names and UIDs of objects or of subdirectories. There is no necessary relation between the level of one directory and level of its parent directory which contains its alias. Consider the example shown in Figure 4.1.

The ovals in the figure represent directories, and the symbols outside each are their UID and security level. Inside each oval is a list of aliases. The root directory, U_0 , is public. Typically, directory structure will involve monotonically increasing security levels as the directory tree is descended. An example is the left half of the tree. The symbolic name `"/usr/vinter/file1"` can be translated, starting at the root, to UID U_9 by a client invoking a lookup at CONFIDENTIAL, since every directory in the path can be read at CONFIDENTIAL. The right half of the tree shows a less common situation. The symbolic name `"/usr/weber/file1"` cannot be translated by a CONFIDENTIAL lookup starting at the root, since directory U_2 cannot be read at this level. (The names of mailbox directories are SECRET in this case). However, the symbolic name `"file1"` can be translated if the lookup is started at directory U_5 , resulting in UID U_7 being returned.

The situation in which the level of a child directory does not dominate the level of its parent causes two problems. First, the catalog manager described in this section cannot create such a child directory in a single operation, but must require the client to act at two separate levels (as described under the CreateDirectory operation in section 4.1.4.1.4). Second, for the child directory to be useful at its own level, users must be provided with its UID directly. This is possible if it was a user at the directory's level who created it, or if it is initial environment information at login. For example, the UID may be a user's home directory.

4.1.4.1.2 CreateAlias A directory UID, an object UID, and a symbolic identifier (with no directory path structure) are given as arguments to a CreateAlias invocation at level l . The catalog manager attempts to add the new alias consisting of the identifier and the object UID to the directory. During the CreateAlias invocation, the catalog

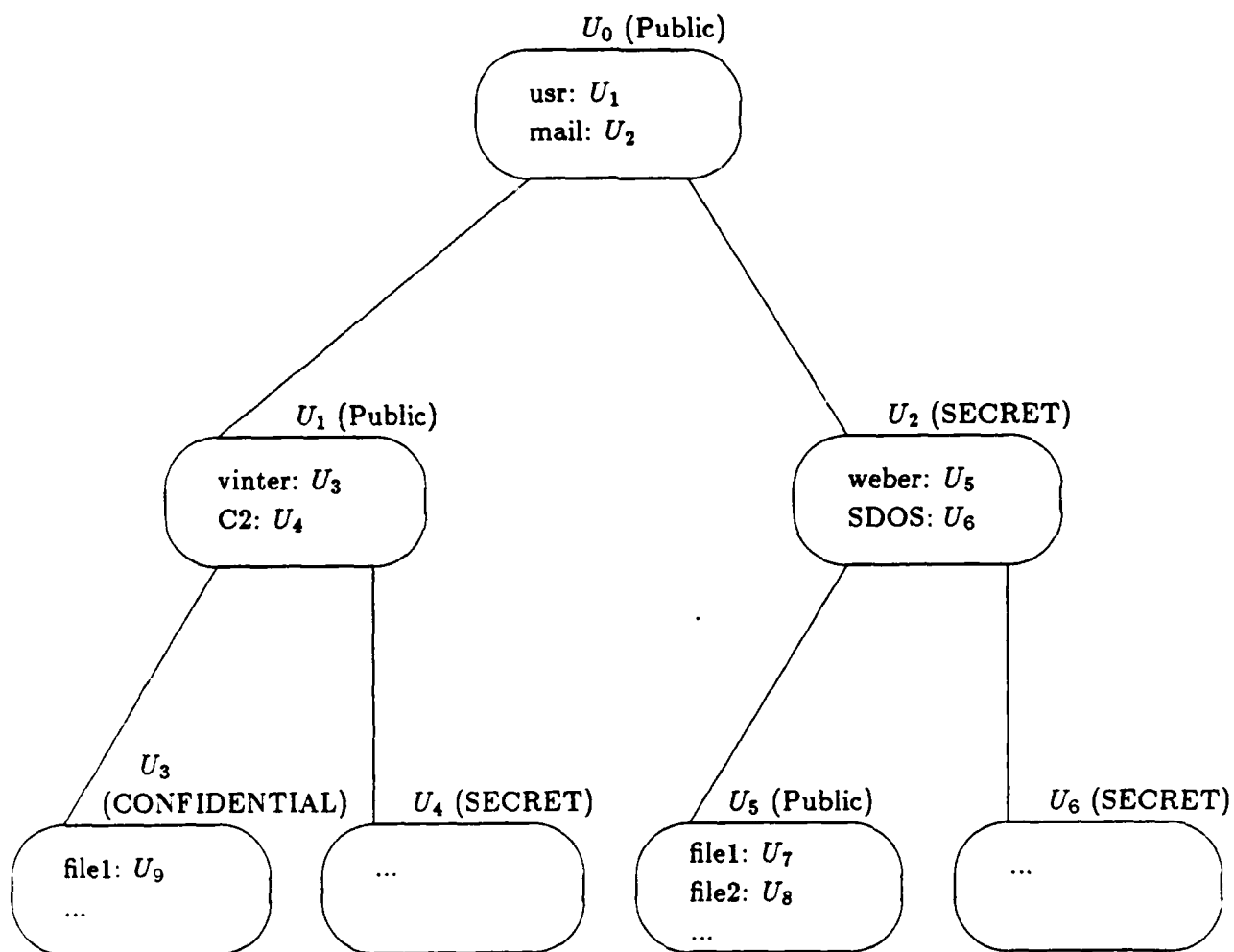


Figure 4.1: A sample directory structure

manager will invoke ReadODB and WriteODB on the host. The CreateAlias invocation will therefore fail if level l does not equal the level of the directory. It will also fail if the symbolic identifier already exists in the directory.

The invocation of CreateAlias on directory D by a client is routed by the message switch to any host on which D resides. This host must be able to start an instance of the catalog manager, M_1 , whose level set includes the level of D . Manager M_1 determines whether D is replicated by querying the SDB. If it is not, then the CreateAlias operation can be carried out purely locally.

On the other hand, if D is replicated, then manager M_1 determines the location of the primary copy of D . From a list of other catalog managers which M_1 maintains, M_1 determines that M_2 is the instance of the catalog manager on that host. Assume that $M_1 \neq M_2$. M_1 sends (not an invoke, but direct IPC) a copy of the CreateAlias request to M_2 . M_2 holds the request until all previously arriving updates to D have completed. It then broadcasts a copy of the request to all catalog managers. Each catalog manager, upon receiving such a request, attempts the CreateAlias operation at level l on its local copy of D . Assuming that the kernel has successfully maintained consistent SDB information for D , and that the catalog manager has maintained consistency among the replicas of D , every instance of CreateAlias will have the same effect on each local replica. When M_2 receives replies from each catalog manager, it replies to M_1 . M_1 then replies to its client.

If $M_1 = M_2$, the communication between M_1 and M_2 described is not necessary.

All messages sent during the above process occur at level l .

4.1.4.1.3 RemoveAlias A directory UID and a symbolic identifier are given as arguments to a RemoveAlias invocation at level l . The catalog manager attempts to remove the alias for that identifier from the directory. During the RemoveAlias invocation, the catalog manager will invoke ReadODB and WriteODB on the host. The invocation will therefore fail if level l does not equal the level of the directory. It will also fail if the symbolic identifier does not exist in the directory. If the directory is replicated, the concurrency control mechanism for updating it is as described for CreateAlias.

4.1.4.1.4 CreateDirectory A directory UID d , a level x , and a symbolic identifier (with no directory path structure) are given as arguments to a CreateDirectory invocation at level l . In order that this invocation succeed, the catalog manager must successfully interact with the local ODB three times:

1. The catalog manager invokes ReadODB on directory d to read its aliases.
2. The catalog manager invokes CreateODB to create a new object (of type 'directory') at level x ; if successful, a new UID u is returned by the ODB.

3. The catalog manager invokes WriteODB on directory d to add the new alias consisting of the symbolic identifier and new UID u to directory d .

The CreateDirectory invocation succeeds if and only if every invocation on the ODB succeeds. The invocation will therefore fail if level l does not equal the level of d , if l is not dominated by x , or if the identifier already exists in d .

Note that the CreateDirectory operation cannot be used in a single invocation to create a directory whose symbolic name is kept in a higher-level directory. With the catalog manager operations as described, this is only possible in two steps: a CreateDirectory invocation at one level, and a CreateAlias invocation at a higher level, using the newly-created directory's UID as a parameter.

If directory d is replicated, the concurrency control for updating it is as described for CreateAlias.

4.1.4.1.5 RemoveDirectory A directory UID d and a symbolic identifier are given as arguments to a RemoveDirectory invocation at level l . In order that this invocation succeed, the catalog manager must successfully interact with the local ODB three times:

1. The catalog manager invokes ReadODB on directory d to read its aliases.
2. The catalog manager invokes RemoveODB to delete the directory whose UID is the translation of the symbolic identifier.
3. The catalog manager invokes WriteODB on directory d to delete the alias containing the symbolic identifier.

The RemoveDirectory invocation succeeds if and only if every invocation on the ODB succeeds. The invocation will therefore fail if level l does not equal the level of d , or if the symbolic identifier does not exist in d .

If directory d is replicated, the concurrency control for updating it is as described for CreateAlias.

4.1.4.1.6 ReplicateDirectory The invocation to replicate a directory has as its object the generic directory object on a particular host, H . It has as its parameter the UID of the directory to be replicated, and it occurs at some level l . In response, the catalog manager on H invokes ReplicateODB on the local kernel at the same level l . The ReplicateDirectory invocation succeeds if and only if the ReplicateODB invocation does.

4.1.4.1.7 DereplicateDirectory The Dereplicate Directory operation is identical to ReplicateDirectory, except that DereplicateODB is invoked on the kernel.

4.1.4.2 Other Remarks

The multi-level security of the catalog manager is straightforward. Each message received by the manager results in zero or more messages sent, all at the same level. Each message sent is purely a function of those previous messages involved in the current invocation. Each decision whether a particular directory is visible to clients at a given level is made within the ODB and returned to the catalog manager at the level of the current invocation. Therefore, processing at different levels will never interact. See section 4.3.3.

Several simplifications have been made in this catalog manager design.

- There are no discretionary access controls defined or implemented by the catalog manager as written. These could be added.
- The operations of Create- and Remove- Directory and Aliases are defined for a particular directory, expressed as a UID. These operations could have been made more general by defining them for arbitrary symbolic directory paths. However, the same general effect can be achieved were the client first to use the lookup operation to translate the path to the UID of a directory, and then to invoke Create or Remove on that UID.
- The set of catalog managers is assumed to be static, rather than dynamically changing as hosts are started or halted. In either situation, what is important is that each catalog manager keep a list of the UIDs of all other catalog managers. In the dynamic case, this list must be updated. These updates, and the means for initializing the list, are not reflected in this specification.

4.1.5 Authentication

4.1.5.1 The TIP and Authentication Manager

There are two processes important to proper authentication. They are the TIP (terminal interface process) and the Authentication Manager. The TIP transmits requests between the user and the system and the Authentication Manager resets the SDB and CI entries of the TIP for each new user.

The Authentication Manager supports at least the following operations: AuthenticateAs (sometimes referred to as Login) and Logout.

For the AuthenticateAs operation, the Authentication Manager receives a login request from the user which contains the principal, the password and the desired security level. (A user with multiple accounts may have different passwords for different security levels.) This request is checked to see if it is allowed and the message label is checked to see if the request came from a terminal ready to login. If the request is not permitted the TIP is informed that the request failed. If it is permitted, first the SDB entry is

updated and then the CI fields are updated (by requests to the kernel). If this succeeds the TIP is informed the request succeeded; otherwise it reports failure.

A Logout request will first set the CI to minimal rights and then set the SDB back to 'terminal ready to login' authority. The TIP is informed when the operation is completed.

Local password data is initialized at the time the Authentication Manager is started. It is anticipated that a change password operation will be implemented.

The TIP serves two roles. It must handle login and logout requests as well as normal transactions between a validated user and the system. This exchange of information involves converting the messages from a form the user recognizes to one which the system recognizes (and conversely from the system to the user). In particular, when converting a message from the user to the system, the correct security level is set. However, there is no explicit specification to ensure that a user request gets converted to a system request with the same semantic content.

The TIP also plays a role in authentication. The TIP may reside in one of the following states:

READY: waiting for a login request to send from the user to the system

LOGININ: waiting for acknowledgement of login from the authentication manager

ACTIVE: messages are passed between the user and the system

LOGINGOUT: waiting for acknowledgement of logout from the authentication manager

Initially the TIP is in the ready state. A request coming in from the user to login will cause the request to be forwarded to the authentication manager and the TIP to enter the loginin state. If the reply comes back indicating a success, then the TIP is put into the active state; otherwise it returns to the ready state. When the TIP is active it will send and receive messages at the requested login level. If one of the requests is a logout, then the TIP will enter loggingout state and wait for the reply of the authentication manager. When successful, the TIP returns to the ready state. Note: The TIP is specified so that it will reject a login request accidentally routed to it. But encryption of logins may be added if extra robustness is desired.

Part of the functioning of the TIP is provided by a screening mechanism which we will call the Filter. The purpose of the Filter is to ensure that the identity of a user does not change in the middle of a terminal session. This process may be partially implemented by hardware or people. Because there is such a wide number of possible designs, we will mostly concern ourselves with enumerating what conditions it must satisfy. A user may request access to a terminal, and then the reply will be permission granted, or permission denied (depending on whether the terminal is active). We will call such a message a 'line request' or a 'terminal request'. A user must explicitly release the terminal so that the next person can use it. Such a message will be called a 'line

release request' or 'terminal release request'. The release request will cause a logout, and put the terminal back in the ready state.

4.1.5.2 Achieving Security

We would like the authentication process to be totally secure. Unfortunately, this is not possible. The source of the problem is that an unauthorized user is permitted to make a login request. Even a negative reply may yield a small bit of sensitive information. (Namely, that a certain password for a given user does not work at that security level.) If this could be repeated enough times then an unauthorized user could learn enough to identify himself as someone else. Thus, the methodology used to prove that the system is essentially secure must take this into account.

There is also another complication. We would like to show 'true restrictiveness'. That is, we want to show restrictiveness based on the true level of messages rather than the level indicated in the label field. Now, a login request may or may not contain information about a real password, user, and security level. If a combination of these is valid then knowing this information could in turn give you real information at that level. Hence the true level of a legal login request will be considered to be the level of the security requested. But suppose the request is not legal. If the information is random then it may tell you nothing. But if it comes from a valid user who misspelled a password, then perhaps it is at the level of the request. But more troubling, a top secret user might try to login at unclassified, but accidentally use the top secret password. Rather than trying to identify the level of this message we will simply give it a level independent of any other user data. We will call the level of an illegal login request to be level X, or the ready-state level. Note that legal logins will also be sent with their actual label set to X. Hence, demonstrating 'true' restrictiveness will be a bit complicated. Fortunately we can restrict the authority to receive label X messages to only the TIP, the Authentication Manager, the message switch, part of the kernel, and System Managers.

A related problem occurs with logout acknowledgement. If it is a line release acknowledgement then its level is the lowest one available to users (since the line release is this low level and this acknowledgement is a consequence of it). If it is just a simple logout acknowledgement, then it is at the level of the user who issued the logout. However, in both cases the kernel will send the message with the actual label set to X. As above, this complication is sufficiently contained, so as to not pose too great a problem.

Users will be trusted not to pass information to lower security levels. This could potentially happen in two ways. First they may login with a low level account and write high information into it. Secondly they may acquire and release the terminal based on secret information. The first problem should be obvious to the user and so should be avoidable. The second one is somewhat more delicate. Namely, the fact that some secret request took a long time, may influence the user to release the terminal later. Hence a trojan horse process might be able to manipulate the user response to release

a line. Indeed it may be able to do this as a storage channel rather than as a timing channel (for instance by using an unclassified process as a pseudo-clock). However, even if the user unintentionally violates this principle, it is unlikely that the size or quality of the information will be significant. In the proofs which follow we will assume that a user does not do any write down. The potential pitfall will be included in the summary.

In practice, it may be possible to avoid implementing any filter, and we will discuss the possibilities.

4.1.5.3 Authentication and System Security

The authentication manager will correctly identify valid logins and will thus be able to attach correct security labels to messages. The authentication manager will be restrictive.

The TIP's outputs are solely a function of its state and the input (it is a finite automata). A correct login will setup the TIP to "stamp" labels on future messages going out and screen out messages coming in from the system. Because the user is trusted, all messages to the system will be labeled correctly. (The Filter will only allow one user at a time to use the terminal.) When the user does a logout or terminates a session, the TIP will await confirmation from the system and then return to a ready state. The TIP will also be restrictive.

Hooking the TIP and authentication manager to the rest of the system, and by applying the hook-up theorem, the entire system will be restrictive.

A user trying to login with an incorrect login will get no reply or a reply saying error. By the external axioms, the response rate will be sufficiently slow and the passwords will be of such complexity that the probability of an accidental login will be approximately zero.

Hence the system will be essentially secure.

4.2 Verifying MLS Properties

One of the goals of this project has been to formally verify that the SDOS design meets the requirements of the SDOS security policy. This would give high assurance that the design is "secure". Much of our work toward this goal has focussed on verifying the part of the mandatory policy requiring constraints on information flow. This part of the policy is stated in section 2.1.2.3.

We have formalized the design of SDOS as a Gypsy program. (See section 4.1 and Appendix A.) To prove, using the Gypsy methodology, that a program meets its requirements, one must express those requirements as assertions that are true at particular times during the execution of the program. We found, however, that the

information flow constraints of the SDOS policy cannot be directly expressed in this way. Other approaches needed to be found. This section presents the approach we took and the theory that supports it.

4.2.1 A New Security Methodology

Our emphasis on mandatory information flow security is a result of the emergence of a new methodology for security verification. The aim of this methodology is security verification through analysis. In other words, large designs can be decomposed into smaller ones, and the security of the larger can be inferred once the properties of the smaller are known. This approach has obvious merits, but it is only recently that it has been applied to formal specifications for information flow security.

The work of McCullough [McCullough 87] is a particular case of this new methodology. In his work, system components are defined in terms of their possible behaviours, in a way that simplifies and slightly modifies the approach of CSP [Brookes et al. 84]. The hook-up, or composition, of two components is defined as in CSP. McCullough searched for, and found, a property that captures many desirable features of information flow security and is also a *composable* property: the hook-up of two components with the property is a new component with the property. We have called his security property *restrictiveness*, or *restriction*.

The verification work presented in this report ties into the new methodology. The SDOS security policy requires that the entire assured MLS part of the system be restrictive. We have endeavored to show that the multi-level secure processes that comprise SDOS are each restrictive, so that the restrictiveness of the entire system could be inferred from composability. However, the problem of demonstrating the restrictiveness of each MLS process remains. One way in which we have handled this problem is to find other, simpler, properties, which when taken together, imply the restrictiveness property. These simpler properties are then proved, using Gypsy, for each component. We needed to develop special techniques for proving some of the simpler properties using Gypsy. This section discusses these techniques and others.

In section 4.2.2 we present a CSP-like framework for discussing processes. We will later make an informal connection between these processes and designs that can be expressed in Gypsy. Within the framework, we formally define the restrictiveness security property. Several other properties are introduced that are "security-like", in the sense that they also limit deducibility, and hence, information flow. Of primary importance is the property called *weak non-interference* (WNI). The limits on deducibility made by this property are similar to those intended in the Goguen-Meseguer model [Goguen and Meseguer 82]. However, WNI is both weaker than restriction, and not composable. By conjuncting several other simple properties with WNI, we can infer restriction. A proof of this claim is given.

Another "security-like" property, *strong non-interference* (SNI), is introduced. This property is neither weaker nor stronger than restriction, but it is composable. Since it is

composable, if it were used as the high-level security requirement for a system, it could also be used as the security requirement for individual components as well. However, it is too strong to be generally useful, since it does not allow components that upgrade information from one security level to a higher one.

We found that Gypsy is ill-suited to direct verification of properties such as restriction. The reason for this is discussed more fully in section 4.2.3. However, simply changing the specification language was unlikely to solve the problem: other popular specification methodologies used for proving invariant properties of state machines would fare no better. Section 4.2.3 is devoted to showing a way to apply the theory of section 4.2.2 within Gypsy. The restrictiveness of a component is to be inferred from the fact that it satisfies WNI, plus other simpler properties. Decomposing restrictiveness into simpler properties can now be seen as an advantage, since these simpler properties are easier to handle in Gypsy. The hardest is WNI, and we present an algorithm for verifying WNI in Gypsy.

In verifying the information flow security of various SDOS components, we found that the definition of security as restrictiveness was not always appropriate. We needed generalizations of the property to permit the following:

- limited downgrading of information via covert channels;
- special protocols used in communication between components;
- assumptions about the boundary between the assured system components and process and users with limited or no assurance.

Section 4.2.4 contains some successes in this direction. However, the subject is far from closed.

4.2.2 Basic Theory

4.2.2.1 Notation

An event is an action performed by a system. Each event will represent a synchronized message-passing communication between systems, or a communication between subsystems from which a single system is composed. Possible behaviours of each system will be represented by sequences of events.

We will need a few notations concerning sequences. E and F will stand for arbitrary sets of events, e a particular event, α and β sequences of events, and $f : E \mapsto F$ a mapping from E into F . Let l be an arbitrary security level, and suppose that each event is associated with some security level. We define:

- $\bar{E}$ is the set of events not in E ;

- E^* is the set of all possible sequences formed from the events in E ;
- $\alpha \wedge \beta$ is the concatenation α followed by β ;
- $\alpha \sqsubseteq \beta$ iff β is an initial subsequence of α , while $\alpha \sqsupset \beta$ iff β is a proper initial subsequence;
- $last(\alpha)$ is the last event in α (undefined if α is empty), while $nonlast(\alpha)$ is the sequence with the last event removed; $first(\alpha)$ is the first event in α (undefined if α is empty), while $nonfirst(\alpha)$ is the sequence with the first event removed;
- $\alpha \upharpoonright E$, the projection of α with respect to set E , is the sequence obtained from α by deleting all events not in E and preserving the order of the events that are left;
- $\alpha \upharpoonright l$ is the projection of α with respect to the set of events associated with security levels less than or equal to level l ;
- $\langle e \rangle$ is the sequence with the single event e , and $\langle \rangle$ is the empty sequence;
- f^* is the mapping from E^* into F^* of application of f componentwise.

4.2.2.2 Processes

A trace is a sequence of events which is a possible behaviour of some system.

An event system is a structure of four sets, $\langle E, I, O, T \rangle$, where E is a set of events, $I \subseteq E$ a set of input events, $O \subseteq E$ a set of output events, and $T \subseteq E^*$ a set of traces. Elements of I are events which are offered as a choice to be made by the environment of the event system. Elements of O are events which may simultaneously be inputs to other event systems. We will generally require that the sets I and O be disjoint, i.e., $I \cap O = \{\}$. There may also be events in E which are neither inputs nor outputs; these events are internal to the event system, and can be thought of as communication events between constituent event systems. The set T includes exactly those sequences which are traces, i.e., possible behaviours of the event system.

A view is an arbitrary set of events. For any event $e \in v$, we say that e is *visible* in the view v . Each view separates the universe of events into a set of events *visible* to some viewer, and a set which is *invisible* or hidden. A particular view which is important for multi-level security is the set of all events associated with security levels less than or equal to level l .

Definition 1 An event system $\langle E, I, O, T \rangle$ is a process iff the empty sequence is a trace, and for any possible trace of the event system, all initial subsequences are also traces. So,

$$\langle \rangle \in T \text{ and} \\ \forall \alpha, \gamma \in E^*, \alpha \wedge \gamma \in T \rightarrow \alpha \in T.$$

Definition 2 A process $\langle E, I, O, T \rangle$ is **input-total**, or **simply total**, iff any trace may always be extended with any input. That is,

$$\forall \alpha \in T, \forall e \in I, \alpha \wedge \langle e \rangle \in T.$$

Two processes, viewed as executing in parallel, may in some cases be hooked together to form another process. If an input event of one process is also an output event of the other, then this mutual event is neither an input nor an output of the hooked-up process but is an internal communication event. We must rule out the possibility that events shared by two processes are not communication events of this form.

Definition 3 Processes $P_1 = \langle E_1, I_1, O_1, T_1 \rangle$ and $P_2 = \langle E_2, I_2, O_2, T_2 \rangle$ are **coherent** if

$$E_1 \cap E_2 = (I_1 \cap O_2) \cup (I_2 \cap O_1).$$

Definition 4 The **hook-up** of processes $P_1 = \langle E_1, I_1, O_1, T_1 \rangle$ and $P_2 = \langle E_2, I_2, O_2, T_2 \rangle$ is defined if P_1 and P_2 are coherent, and yields a new process $P_1 \parallel P_2 = \langle E, I, O, T \rangle$ with

$$\begin{aligned} E &= E_1 \cup E_2 \\ I &= I_1 \cup I_2 - E_1 \cap E_2 \\ O &= O_1 \cup O_2 - E_1 \cap E_2 \\ \forall t \in E^*, t \in T &\leftrightarrow (t \upharpoonright E_1) \in T_1 \text{ and } (t \upharpoonright E_2) \in T_2. \end{aligned}$$

A sequence will be a trace of a hook-up process if and only if its projection on the set of events of each component process is a trace of that component.

The internal communication events of the hook-up, the events that are shared between the two processes, form a sequence. This sequence is the same for both processes. Each shared event represents synchronous communication, i.e., the two processes must simultaneously agree to engage in the event.

4.2.2.3 Security Properties

In this section, several variant definitions of security for processes are given. Composability is defined, and the composability of each security property is determined. The set v represents an arbitrary view.

Definition 5 A process $\langle E, I, O, T \rangle$ satisfies **strong non-interference (SNI)** with respect to view v iff for every trace, the visible projection of that trace is also a trace. That is,

$$\forall \alpha \in T, \alpha \upharpoonright v \in T.$$

Definition 6 A process $\langle E, I, O, T \rangle$ satisfies **weak non-interference (WNI)** with respect to view v iff for every trace, there is another trace with the same visible projection but with no invisible inputs. Formally,

$$\forall \alpha \in T, \exists \alpha' \in T, \alpha' \upharpoonright v = \alpha \upharpoonright v \text{ and } \alpha' \upharpoonright I \upharpoonright \bar{v} = \langle \rangle.$$

Strong non-interference clearly implies weak non-interference.

Definition 7 A process $\langle E, I, O, T \rangle$ is restrictive with respect to view v iff it is input-total and for any sequences $\alpha, \gamma \in E^*$, and input sequences $\beta, \beta' \in I^*$, if

$$\begin{aligned} \alpha \wedge \beta \wedge \gamma &\in T \text{ and} \\ \beta \uparrow v &= \beta' \uparrow v \text{ and} \\ \gamma \uparrow I \uparrow \bar{v} &= \langle \rangle \end{aligned}$$

then there exists a sequence $\gamma' \in E^*$ such that

$$\begin{aligned} \alpha \wedge \beta' \wedge \gamma' &\in T \text{ and} \\ \gamma' \uparrow v &= \gamma \uparrow v \text{ and} \\ \gamma' \uparrow I \uparrow \bar{v} &= \langle \rangle. \end{aligned}$$

Each of these definitions limits deducibility. Each holds for an arbitrary trace, which may be thought of as the actual system history, and asserts that there could also exist a history with the same visible behaviour, but with possibly different invisible behaviour. If the invisible behaviour could be different, then one is prevented from deducing that the actual invisible behaviour occurred.

The three properties differ in which alternative histories they claim are possible. SNI constructs an alternate history by removing all invisible events, WNI constructs an alternate history by removing all invisible inputs, and restriction constructs an alternate history by arbitrarily changing a block of the most recent invisible inputs. Restriction implies WNI, since one could remove all the invisible inputs in a history by starting with the most recent and working backwards. The converse is not true, and counterexamples are easily found.

Definition 8 A property of processes is composable (or, is a hookup property) iff, for any coherent processes P and Q , that property holds for $P \parallel Q$ whenever it holds for P and it holds for Q .

Theorem 1 The property of SNI with respect to v is composable.

Proof: Let $P_1 = \langle E_1, I_1, O_1, T_1 \rangle$ and $P_2 = \langle E_2, I_2, O_2, T_2 \rangle$ be arbitrary coherent processes satisfying SNI with respect to v , and let t be an arbitrary trace of $P_1 \parallel P_2$. Then $t \uparrow E_1 \in T_1$ and $t \uparrow E_2 \in T_2$. Using SNI with respect to v for process P_1 we have $t \uparrow E_1 \uparrow v \in T_1$, and therefore $t \uparrow v \uparrow E_1 \in T_1$; similarly for process P_2 , we have $t \uparrow v \uparrow E_2 \in T_2$. Then $t \uparrow v \in T$. $\square$

The property of being restrictive with respect to view v is also a hookup property. This has been proved in [Ulysses 87].

WNI is not a composable property. As a simple counterexample, consider the following two non-input-total processes. Process A has input events S and P , but no output events. It has the trace

$$\langle S, P \rangle$$

and every initial subsequence of this. In other words, P can be accepted only after S . Process B has no inputs, outputs U and P , and traces

$$\langle P, U \rangle$$

In other words, if U is output, P must have been output first. Let v be a view that includes only the event U . (U may be thought of as an unclassified event, while the other two events are secret.) Both processes are WNI. For the first process, no event is ever visible, so even if secret inputs happened, this fact cannot be deduced. Therefore A is WNI. For the second, no secret input is ever possible anyway, so B is also WNI. But their hook-up is not WNI, since the history $\langle S, P, U \rangle$ that produces the visible unclassified sequence $\langle U \rangle$ must have started with secret inputs.

4.2.2.4 Buffers

A buffer is a process which inputs messages, and may later output those same message in the same order. The events of inputting a message and outputting the same message are not the same, but there must a mapping which relates one to the other. A simple specification for a buffer requires that the history of outputs is an initial subsequence of the history of inputs, appropriately mapped.

Definition 9 *The process $\langle E, I, O, T \rangle$ is an infinite buffer iff $I \cap O = \{\}$ and there is a one-to-one and onto mapping $\text{map} : I \rightarrow O$, with*

$$\forall \alpha \in E^*, \alpha \in T \leftrightarrow \forall \alpha' \sqsubseteq \alpha, \alpha' \upharpoonright O \sqsubseteq \text{map}^*(\alpha' \upharpoonright I).$$

In addition, the buffer preserves view v iff

$$\forall x \in I, x \in v \leftrightarrow \text{map}(x) \in v.$$

The idea behind preserving a view is that if any given message is either visible or invisible, then it is natural that the events of inputting and outputting a given message either both be visible or both be invisible. This requirement may also be stated: for any trace α , $\text{map}^*(\alpha) \upharpoonright v = \text{map}^*(\alpha \upharpoonright v)$.

Note that there may be internal events of the buffer, but since the definition involves only projections on I and O , internal events may be arbitrarily added to or removed from any trace.

Lemma 1 *Every infinite buffer is input-total.*

Proof: Let $B = \langle E, I, O, T \rangle$ be an infinite buffer with mapping map , $t \in T$ a buffer trace, $e \in I$ an input, and α' an initial subsequence of $\alpha \wedge \langle e \rangle$. If $\alpha' \sqsubseteq \alpha$, then we already know that $\alpha' \upharpoonright O \sqsubseteq \text{map}^*(\alpha' \upharpoonright I)$. On the other hand, if $\alpha' = \alpha \wedge \langle e \rangle$, then $\alpha' \upharpoonright O = \alpha \upharpoonright O \sqsubseteq \text{map}^*(\alpha \upharpoonright I) \wedge \text{map}(e) = \text{map}^*(\alpha' \upharpoonright I)$. So B is input-total. $\square$

The previous lemma showed that a buffer trace may always be extended by any input. The following lemma gives sufficient conditions for extending a buffer trace by an output.

Lemma 2 *If process $B = \langle E, I, O, T \rangle$ is an infinite buffer preserving v , then for any visible output event $e \in O \cap v$, sequences $\mu, \nu \in E^*$, if*

$$\begin{aligned} \mu^\wedge \langle e \rangle &\in T \text{ and} \\ \nu &\in T \text{ and} \\ \mu \uparrow v &= \nu \uparrow v \end{aligned}$$

then there exists an output sequence $\lambda \in O^*$ for which

$$\begin{aligned} \nu^\wedge \lambda^\wedge \langle e \rangle &\in T \text{ and} \\ \lambda \uparrow v &= \langle \rangle. \end{aligned}$$

Proof: Because $\mu^\wedge \langle e \rangle \in T$ is a buffer trace, we have that $\text{map}^*(\mu \uparrow I) = \text{map}^*((\mu^\wedge \langle e \rangle) \uparrow I) \sqsupseteq (\mu^\wedge \langle e \rangle) \uparrow O = \mu \uparrow O^\wedge \langle e \rangle$. Because $\nu \in T$ is a buffer trace, we have that $\text{map}^*(\nu \uparrow I) \sqsupseteq \nu \uparrow O$. Then there is a sequence $\kappa \in O^*$ such that $\text{map}^*(\nu \uparrow I) = \nu \uparrow O^\wedge \kappa$. Projecting on view v , $\text{map}^*(\nu \uparrow I) \uparrow v = \nu \uparrow O \uparrow v^\wedge \kappa \uparrow v$. But by using the facts that B preserves v and that $\mu \uparrow v = \nu \uparrow v$, we also have

$$\text{map}^*(\nu \uparrow I) \uparrow v = \text{map}^*(\nu \uparrow I \uparrow v) = \text{map}^*(\mu \uparrow I) \uparrow v \sqsupseteq \mu \uparrow O \uparrow v^\wedge \langle e \rangle = \nu \uparrow O \uparrow v^\wedge \langle e \rangle.$$

So $\text{first}(\kappa \uparrow v) = \langle e \rangle$ and $\kappa = \lambda^\wedge \langle e \rangle^\wedge \lambda'$ where $\lambda, \lambda' \in O^*$ and $\lambda \uparrow v = \langle \rangle$.

It now remains to show that $\nu^\wedge \lambda^\wedge \langle e \rangle$ is a buffer trace. Let ν be an initial subsequence of $\nu^\wedge \lambda^\wedge \langle e \rangle$. On the one hand, if $\nu' \sqsubseteq \nu$, then we already know that $\text{map}^*(\nu' \uparrow I) \sqsupseteq \nu' \uparrow O$, because ν is a trace. On the other hand, if $\nu \sqsubset \nu' \sqsubseteq \nu^\wedge \lambda^\wedge \langle e \rangle$, then

$$\text{map}^*(\nu' \uparrow I) \sqsupseteq \text{map}^*(\nu \uparrow I) = \text{map}^*((\nu^\wedge \lambda^\wedge \langle e \rangle) \uparrow I) \sqsupseteq \nu \uparrow O^\wedge \lambda^\wedge \langle e \rangle \sqsupseteq \nu' \uparrow O.$$

and the lemma is proved. $\square$.

4.2.2.5 WNI and Determinism

The property WNI is weaker than restrictiveness. For one thing, WNI does not require input-totalness, whereas restrictiveness does. More importantly, WNI allows one to produce a new trace from an old one by removing invisible inputs. In comparison, restrictiveness allows new traces to be produced both by removing and inserting invisible inputs within a block of inputs. This is an intuitive reason that WNI is less powerful than restrictiveness. However, if we require that a process satisfy some simple properties in addition to WNI, we gain the ability to create new traces of that process by inserting invisible events, including inputs. In particular, we will need to require that a WNI process also be deterministic, and that it can produce some response to any given input sequence.

Definition 10 A process $\langle E, I, O, T \rangle$ is *input-live* iff the set I is nonempty and any trace may be properly extended into another trace ending in an input. That is,

$$\forall \alpha \in T, \exists \alpha_x \in T, \alpha_x \supset \alpha \text{ and } \text{last}(\alpha_x) \in I.$$

Note that because $\langle E, I, O, T \rangle$ is a process, input-liveness also implies that there is a proper extension ending in an input, and which has exactly one additional input.

Definition 11 A process $\langle E, I, O, T \rangle$ is *input-universal* iff, when it is possible to accept some input, any input may be accepted. That is,

$$\forall \alpha \in T, \forall e_1, e_2 \in I, \alpha^\wedge \langle e_1 \rangle \in T \rightarrow \alpha^\wedge \langle e_2 \rangle \in T.$$

The following lemma states that every trace of an input-live and input-universal system can be extended in response to every non-empty sequence of inputs, and that the extension can be made to end in an input.

Lemma 3 If process $\langle E, I, O, T \rangle$ is input-live and input-universal, then for every trace $\alpha \in T$, every input sequence $\beta \in I^*$, and any input $e \in I$, there exists a sequence $\gamma \in E^*$ such that

$$\begin{aligned} \alpha^\wedge \gamma^\wedge \langle e \rangle &\in T \text{ and} \\ \gamma \upharpoonright I &= \beta. \end{aligned}$$

Proof: Use induction on the length of the input sequence β . Suppose $\text{length}(\beta) = 0$; then $\beta = \langle \rangle$. By input-liveness, there is a sequence α_x which is a proper extension to α , such that $\alpha^\wedge \alpha_x \in T$, and $\alpha_x \upharpoonright I = \text{last}(\alpha_x)$. Let $\gamma = \text{nonlast}(\alpha_x)$; then $\gamma \upharpoonright I = \langle \rangle$. By input-universality, $\alpha^\wedge \gamma^\wedge \langle e \rangle \in T$ is also a trace. Now suppose the lemma holds for all input sequences with $\text{length} < n$, and β is any input sequence with $\text{length}(\beta) = n$. Then the induction hypothesis, with input sequence $\text{nonlast}(\beta)$ and input $\text{last}(\beta)$, shows that there is a sequence λ such that $\alpha^\wedge \lambda^\wedge \langle \text{last}(\beta) \rangle \in T$ and $\lambda \upharpoonright I = \text{nonlast}(\beta)$. Again, by input-liveness and -universality, there is a proper extension α_x such that $\alpha^\wedge \lambda^\wedge \langle \text{last}(\beta) \rangle^\wedge \text{nonlast}(\alpha_x)^\wedge \langle e \rangle \in T$ is a trace, and $\alpha_x \upharpoonright I = \langle \rangle$. Let $\gamma = \lambda^\wedge \langle \text{last}(\beta) \rangle^\wedge \text{nonlast}(\alpha_x)$. Then $\gamma \upharpoonright I = \beta$. $\square$

There is more than one useful definition of 'determinism'.

Definition 12 A process $\langle E, I, O, T \rangle$ is *asynchronously deterministic* if every non-input event in a history is uniquely determined by the preceding sequence of events. That is,

$$\forall \alpha \in E^*, \forall e_1, e_2 \in \bar{I}, \alpha^\wedge \langle e_1 \rangle \in T \text{ and } \alpha^\wedge \langle e_2 \rangle \in T \rightarrow e_1 = e_2.$$

For a process obeying this form of determinism it is possible that a trace can be extended either by an input or a non-input. This is the natural statement of determinism for a process which is input-total (or "asynchronous"). A stronger form of determinism, not applicable to input-total processes, can additionally require that the choice between accepting an input and performing an output or internal event be made uniquely on the basis of the preceding history.

Definition 13 A process $\langle E, I, O, T \rangle$ is *synchronously deterministic* iff

$$\forall \alpha \in E^*, \forall e_1, e_2 \in E, \alpha^\wedge \langle e_1 \rangle \in T \text{ and } \alpha^\wedge \langle e_2 \rangle \in T \rightarrow \\ (e_1, e_2 \in I) \text{ or} \\ (e_1, e_2 \in \bar{I} \text{ and } e_1 = e_2).$$

Lemma 4 If process $\langle E, I, O, T \rangle$ is *synchronously deterministic* then

$$\forall \gamma, \delta \in T, \gamma \uparrow I = \delta \uparrow I \rightarrow \gamma \sqsubseteq \delta \text{ or } \delta \sqsubseteq \gamma.$$

Proof: Let $\gamma = \zeta^\wedge \eta_1$ and $\delta = \zeta^\wedge \eta_2$, where either η_1 or η_2 is $\langle \rangle$, or $\text{first}(\eta_1) \neq \text{first}(\eta_2)$. In the latter case, $\text{first}(\eta_1) \in I$ and $\text{first}(\eta_2) \in I$, by the definition of synchronous determinism. But then $\gamma \uparrow I = \zeta \uparrow I^\wedge \eta_1 \uparrow I \neq \zeta \uparrow I^\wedge \eta_2 \uparrow I = \delta \uparrow I$. This is a contradiction, and so $\eta_1 = \langle \rangle$ or $\eta_2 = \langle \rangle$, as was to be shown. $\square$

The following corollary is useful for showing that two traces are equal, given that their inputs are equal.

Corollary 1 If process $\langle E, I, O, T \rangle$ is *synchronously deterministic* then

$$\forall \gamma, \delta \in T, \gamma \uparrow I = \delta \uparrow I \text{ and } \text{last}(\gamma) \in I \text{ and } \text{last}(\delta) \in I \rightarrow \gamma = \delta.$$

Proof: Suppose $\delta \sqsubset \gamma$, then let $\gamma = \delta^\wedge \eta$, where η is non-empty. Then $\gamma \uparrow I = \delta \uparrow I^\wedge \eta \uparrow I$, so $\langle \rangle = \eta \uparrow I$. But $\text{last}(\eta) = \text{last}(\gamma) \in I$, which is a contradiction. By interchanging δ and γ , we see that $\gamma \sqsubset \delta$ also leads to a contradiction. Combining with lemma 4, we have $\gamma = \delta$. $\square$

The following lemma shows the invisible behaviour of a deterministic, WNI process may be altered without changing its ability to accept an input as its next visible event.

Lemma 5 If process $\langle E, I, O, T \rangle$ is *input-live, input-universal, synchronously deterministic*, and satisfies WNI with respect to v , then for any visible input event $e \in I \cap v$, sequences $\mu, \nu \in E^*$, and input sequence $\beta \in I^*$, if

$$\begin{aligned} \mu^\wedge \langle e \rangle &\in T \text{ and} \\ \nu &\in T \text{ and} \\ \mu \uparrow v &= \nu \uparrow v \text{ and} \\ \beta \uparrow v &= \langle \rangle \end{aligned}$$

then there exists a sequence $\lambda \in E^*$ such that

$$\begin{aligned} \nu^\wedge \lambda^\wedge \langle e \rangle &\in T \text{ and} \\ \lambda \uparrow I &= \beta \text{ and} \\ \lambda \uparrow v &= \langle \rangle. \end{aligned}$$

Proof: According to lemma 3, the trace ν may be extended in response to input sequence $\beta^\wedge\langle e \rangle$ to produce the trace $\nu^\wedge\lambda^\wedge\langle e \rangle \in T$, where $\lambda \uparrow I = \beta$. Because the process satisfies WNI, there exists a trace $\nu' \in T$ such that $\nu' \uparrow v = \nu \uparrow v^\wedge\lambda \uparrow v^\wedge\langle e \rangle$ and $\nu' \uparrow I \uparrow \bar{v} = \langle \rangle$. Similarly applying WNI to the trace $\mu^\wedge\langle e \rangle$, there exists another trace $\mu' \in T$ such that $\mu' \uparrow v = \mu \uparrow v^\wedge\langle e \rangle$ and $\mu' \uparrow I \uparrow \bar{v} = \langle \rangle$. Since μ' and ν' are traces of a process, they may be shortened to traces μ_1 and ν_1 respectively, each of which ends in event e . We then have that

$$\begin{aligned} \mu_1, \nu_1 &\in T \\ \mu_1 \uparrow v &= \mu \uparrow v^\wedge\langle e \rangle \\ \nu_1 \uparrow v &= \nu \uparrow v^\wedge\lambda \uparrow v^\wedge\langle e \rangle \\ \mu_1 \uparrow I \uparrow v &= \nu_1 \uparrow I \uparrow v = \langle \rangle. \end{aligned}$$

Calculating,

$$\begin{aligned} \nu_1 \uparrow I &= \nu_1 \uparrow v \uparrow I = \\ \nu \uparrow v \uparrow I^\wedge\beta \uparrow v^\wedge\langle e \rangle &= \\ \mu \uparrow v \uparrow I^\wedge\langle e \rangle &= \\ \mu_1 \uparrow v \uparrow I &= \\ \mu_1 \uparrow I. \end{aligned}$$

Then the corollary to lemma 4 shows that $\mu_1 = \nu_1$.

Equating $\mu_1 \uparrow v$ with $\nu_1 \uparrow v$, we find $\mu \uparrow v^\wedge\langle e \rangle = \nu \uparrow v^\wedge\lambda \uparrow v^\wedge\langle e \rangle$. Since $\mu \uparrow v = \nu \uparrow v$, it follows that $\lambda \uparrow v = \langle \rangle$. Therefore, this λ satisfies the required properties. $\square$

This next lemma shows that a WNI process satisfying the conditions of the previous lemma will permit arbitrary invisible changes to a history without changing the next visible non-input event which is possible.

Lemma 6 *If process $\langle E, I, O, T \rangle$ is input-live, input-universal, synchronously deterministic, and satisfies WNI with respect to v , then for any visible non-input event $e \in (E - I) \cap v$, and sequences $\mu, \nu \in E^*$, if*

$$\begin{aligned} \mu^\wedge\langle e \rangle &\in T \text{ and} \\ \nu &\in T \text{ and} \\ \mu \uparrow v &= \nu \uparrow v \end{aligned}$$

then there exists a sequence $\lambda \in (E - I)^*$ such that

$$\begin{aligned} \nu^\wedge\lambda^\wedge\langle e \rangle &\in T \text{ and} \\ \lambda \uparrow v &= \langle \rangle. \end{aligned}$$

Proof: Divide the proof into two cases, depending on whether the process has any visible inputs in which it may engage.

Case I: $I \cap v \neq \{\}$

Let $x \in I \cap v$ be a visible input. Then by lemma 3 with input sequence $\langle x \rangle$, trace ν can be extended to $\nu^\wedge\rho^\wedge\langle x \rangle$ where $\rho \uparrow I = \langle \rangle$. By WNI, there exists a trace $\nu' \in T$ such

that $\nu' \uparrow v = \nu \uparrow v \wedge \rho \uparrow v \wedge \langle x \rangle$ and $\nu' \uparrow I \uparrow \bar{v} = \langle \rangle$. Without loss of generality, we may choose ν' to be the shortest such trace, the one which has no high-level events following x . So $\text{last}(\nu') = x$.

Similarly, trace $\mu \wedge \langle e \rangle$ can be extended to $\mu \wedge \langle e \rangle \wedge \sigma \wedge \langle x \rangle$ where $\sigma \uparrow I = \langle \rangle$. By WNI, there exists a trace $\mu' \in T$ such that $\mu' \uparrow v = \mu \uparrow v \wedge \langle e \rangle \wedge \sigma \uparrow v \wedge \langle x \rangle$ and $\mu' \uparrow I \uparrow \bar{v} = \langle \rangle$. Again, without loss of generality, $\text{last}(\mu') = x$.

Calculating,

$$\begin{aligned} \nu' \uparrow I &= \nu' \uparrow v \uparrow I = \\ &= \nu \uparrow v \uparrow I \wedge \langle x \rangle = \\ &= \mu \uparrow v \uparrow I \wedge \langle x \rangle = \\ &= \mu' \uparrow v \uparrow I = \\ &= \mu' \uparrow I. \end{aligned}$$

By the corollary to lemma 4, $\mu' = \nu'$. Hence, $\mu \uparrow v \wedge \langle e \rangle \wedge \sigma \uparrow v \wedge \langle x \rangle = \nu \uparrow v \wedge \rho \uparrow v \wedge \langle x \rangle$. Therefore, there exists $\lambda \in E^*$ such that $\rho \sqsupseteq \lambda \wedge \langle e \rangle$, with $\lambda \uparrow v = \langle \rangle$. Since $\rho \in (E - I)^*$, we have $\lambda \in (E - I)^*$. Since $\nu \wedge \rho \wedge \langle x \rangle$ is a trace, so is its initial subsequence $\nu \wedge \lambda \wedge \langle e \rangle$.

Case II: $I \cap v = \{ \}$

First we will show that μ has no inputs. Suppose otherwise; let its first input be $x \in \bar{v}$. Then $\mu = \mu_1 \wedge \langle x \rangle \wedge \mu_2$, where $\mu_1 \uparrow I = \langle \rangle$. Applying WNI to trace $\mu_1 \wedge \langle x \rangle \wedge \mu_2 \wedge \langle e \rangle$, there exists a trace $\mu' \in T$ such that $\mu' \uparrow v = \mu_1 \uparrow v \wedge \mu_2 \uparrow v \wedge \langle e \rangle$ and $\mu' \uparrow I \uparrow \bar{v} = \mu' \uparrow I = \langle \rangle$. Lemma 3 shows that there exists a sequence μ_x such that $\mu' \wedge \mu_x \wedge \langle x \rangle \in T$ where $\mu_x \uparrow I = \langle \rangle$. We now have two traces which contain x as their last event and only input; thus by the corollary to lemma 4, $\mu_1 \wedge \langle x \rangle = \mu' \wedge \mu_x \wedge \langle x \rangle$. Then $\mu_1 \uparrow v \sqsupseteq \mu' \uparrow v$. But from the construction of μ' above, we also know that $\mu' \uparrow v \sqsupset \mu_1 \uparrow v$, which is a contradiction. So μ contains no inputs.

Similarly, suppose that ν contains an input; let its first input be $y \in \bar{v}$. Then $\nu = \nu_1 \wedge \langle y \rangle \wedge \nu_2$ where $\nu_1 \uparrow I = \langle \rangle$. Use lemma 3 to extend trace $\mu \wedge \langle e \rangle$ to trace $\mu \wedge \langle e \rangle \wedge \mu_x \wedge \langle y \rangle \in T$, where $\mu_x \uparrow I = \langle \rangle$. We now have two traces which contain y as their last event and only input; thus by the corollary to lemma 4, $\nu_1 \wedge \langle y \rangle = \mu \wedge \langle e \rangle \wedge \mu_x \wedge \langle y \rangle$. Then $\nu_1 \uparrow v \sqsupset \mu \uparrow v = \nu \uparrow v$. But from the decomposition of ν above, we also know that $\nu \uparrow v \sqsupset \nu_1 \uparrow v$, which is a contradiction. So ν contains no inputs either.

By lemma 3 with input y , extend ν to $\nu \wedge \nu_x \wedge \langle y \rangle \in T$ where $\nu_x \uparrow I = \langle \rangle$. Both this trace and $\mu \wedge \langle e \rangle \wedge \mu_x \wedge \langle y \rangle \in T$ contain y as their last event and only input; thus by the corollary to lemma 4, $\mu \wedge \langle e \rangle \wedge \mu_x \wedge \langle y \rangle = \nu \wedge \nu_x \wedge \langle y \rangle$. Projecting with respect to v and using the fact that $\mu \uparrow v = \nu \uparrow v$, we see that $\nu_x \uparrow v \sqsupseteq \langle e \rangle$. So there is a $\lambda \in (E - I)^*$ such that $\nu_x \sqsupseteq \lambda \wedge \langle e \rangle$ and $\lambda \uparrow v = \langle \rangle$. Thus λ is the required extension to ν . $\square$

4.2.2.6 Strengthening WNI

In this section, we show that if a process satisfying WNI is effectively made input-total by buffering all its inputs, and if it is also required to be deterministic and to accept any input sequence, then the hook-up of the WNI process and its associated buffers

is restrictive. Thus we have a means for proving that a process is restrictive, without resorting directly to the definition of restrictiveness.

In the following theorem, these objects will appear:

- an arbitrary view v ;
- a nonempty array of infinite buffers preserving v , $B_1, \dots, B_n$, with buffer $B_i = \langle E_i, I_i, O_i, T_i \rangle$, $I_i \cap O_i = \{\}$ and $i \neq j \rightarrow E_i \cap E_j = \{\}$;
- process $B = B_1 \parallel B_2 \parallel \dots \parallel B_n = \langle EB, IB, OB, TB \rangle$, the hookup of all the buffers, in which
 - $EB = \bigcup_{(1 \leq i \leq n)} E_i$;
 - $IB = \bigcup_{(1 \leq i \leq n)} I_i$;
 - $OB = \bigcup_{(1 \leq i \leq n)} O_i$;
 - $t \in TB$ iff $t \upharpoonright E_i \in T_i$ for $1 \leq i \leq n$;
- process $A = \langle EA, IA, OA, TA \rangle$, with $IA = OB$ and $EA \cap EB = IA$;
- process $W = A \parallel B = \langle EW, IW, OW, TW \rangle$, the hookup of all the processes, where
 - $EW = EA \cup EB$;
 - $IW = IB$;
 - $OW = OA$;
 - $t \in TW$ iff $t \upharpoonright EA \in TA$ and $t \upharpoonright E_i \in T_i$ for $1 \leq i \leq n$.

Theorem 2 *If A is input-live, input-universal, synchronously deterministic, and satisfies WNI with respect to v , then W is restrictive with respect to v .*

Proof: To show that W is restrictive, we must first show that it is input-total. Let $t \in TW$ be an arbitrary trace, and e any input event for W . We will show that t may be extended by e . $e \in IW$, and therefore $e \in I_k$ for some k , but e is an element of no other I_l , nor is it in EA , since these sets are disjoint from I_k . Since B_k is an input-total process, $(t \wedge \langle e \rangle) \upharpoonright E_k \in T_k$. But we also have $(t \wedge \langle e \rangle) \upharpoonright E_l = t \upharpoonright E_l \in T_l$ for $l \neq k$, and $(t \wedge \langle e \rangle) \upharpoonright EA = t \upharpoonright EA \in TA$. Therefore $t \wedge \langle e \rangle \in TW$.

We must also show that inputs to W can be modified in certain ways without altering visible behaviour. Suppose that $\alpha \wedge \beta \wedge \gamma \in TW$, with $\beta \in IW^*$, and $\gamma \upharpoonright IW \upharpoonright \bar{v} = \langle \rangle$. Also suppose that $\beta' \in IW^*$ with $\beta' \upharpoonright v = \beta \upharpoonright v$. We need to show that there is some $\gamma' \in EW^*$ such that $\alpha \wedge \beta' \wedge \gamma' \in TW$, and $\gamma' \upharpoonright v = \gamma \upharpoonright v$, and $\gamma' \upharpoonright IW \upharpoonright \bar{v} = \langle \rangle$.

We will proceed by induction on the length of γ . For the base case, suppose $\text{length}(\gamma) = 0$ and hence $\gamma = \langle \rangle$. Let $\gamma' = \langle \rangle$. Because W is a process and $\alpha \wedge \beta \wedge \gamma \in TW$, we know that $\alpha \in TW$. Because W is input-total and β' is purely inputs to W , we see

that $\alpha \wedge \beta' = \alpha \wedge \beta' \wedge \gamma' \in TW$. Trivially, $\gamma' \uparrow v = \langle \rangle = \gamma \uparrow v$ and $\gamma' \uparrow IW \uparrow \bar{v} = \langle \rangle$. Therefore the theorem holds in the base case.

For the induction step, suppose $length(\gamma) = n > 0$ and that the theorem holds for all shorter lengths. Let $\gamma = \delta \wedge \langle e \rangle$. Since W is a process, $\alpha \wedge \beta \wedge \delta \in TW$. By the induction hypothesis, there exists a $\delta' \in EW^*$ such that there is an altered trace $\alpha \wedge \beta' \wedge \delta' \in TW$ and $\delta' \uparrow v = \delta \uparrow v$ and $\delta' \uparrow IW \uparrow \bar{v} = \langle \rangle$.

Suppose that the event e is not visible, i.e., $e \in \bar{v}$. Then let $\gamma' = \delta'$. As noted above, this γ' is a possible extension of $\alpha \wedge \beta'$. We also see that $\gamma' \uparrow v = \delta' \uparrow v = \delta \uparrow v = \gamma \uparrow v$, and $\gamma' \uparrow IW \uparrow \bar{v} = \langle \rangle$. So the theorem holds in this case.

Suppose alternatively that the event e is visible, i.e., $e \in v$. We now divide the proof into four cases, depending on which processes engage in the event e . The event is either (i) the input to some buffer, (ii) an internal event for some buffer, (iii) an output from some buffer and simultaneously an input to process A , or (iv) a non-input of process A .

Case I: $e \in IB$

Let $\gamma' = \delta' \wedge \langle e \rangle$. Process W is input-total, and therefore $\alpha \wedge \beta' \wedge \gamma' \in TW$. In addition, $\gamma' \uparrow v = \delta' \uparrow v \wedge \langle e \rangle \uparrow v = \delta \uparrow v \wedge \langle e \rangle \uparrow v = \gamma \uparrow v$ and $\gamma' \uparrow IW \uparrow \bar{v} = \delta' \uparrow IW \uparrow \bar{v} \wedge \langle e \rangle \uparrow \bar{v} = \langle \rangle$.

Case II: $e \in EB - IB - OB$

Again let $\gamma' = \delta' \wedge \langle e \rangle$. e occurs as an internal event in exactly one buffer. The projection of $\alpha \wedge \beta' \wedge \delta'$ on that buffer's event set can always be extended by an internal event. The internal event affects no other process. Therefore the argument given for Case I applies here as well.

Case III: $e \in OB = IA$

Since the buffers have no events in common, event e is in exactly one buffer history; suppose $e \in O_k$. To see that the altered trace can be extended at some point by e , apply lemma 2 with

$$\begin{aligned} \mu &\sim (\alpha \wedge \beta \wedge \delta) \uparrow E_k \\ \nu &\sim (\alpha \wedge \beta' \wedge \delta') \uparrow E_k \in T_k \\ (\alpha \wedge \beta \wedge \delta) \uparrow E_k \wedge \langle e \rangle &\in T_k \\ (\alpha \wedge \beta \wedge \delta) \uparrow E_k \uparrow v &= (\alpha \wedge \beta' \wedge \delta') \uparrow E_k \uparrow v. \end{aligned}$$

Then there exists sequence $\rho \in O_k^*$ such that

$$\begin{aligned} (\alpha \wedge \beta' \wedge \delta') \uparrow E_k \wedge \rho \wedge \langle e \rangle &\in T_k \\ \rho \uparrow v &= \langle \rangle. \end{aligned}$$

ρ is now a new sequence of inputs to A . Apply lemma 5 with

$$\begin{aligned} \mu &\sim (\alpha \wedge \beta \wedge \delta) \uparrow EA \\ \nu &\sim (\alpha \wedge \beta' \wedge \delta') \uparrow EA \in TA \\ \beta &\sim \rho \in IA^* \\ (\alpha \wedge \beta \wedge \delta) \uparrow EA \wedge \langle e \rangle &\in TA \\ (\alpha \wedge \beta \wedge \delta) \uparrow EA \uparrow v &= (\alpha \wedge \beta' \wedge \delta') \uparrow EA \uparrow v \\ \rho \uparrow v &= \langle \rangle. \end{aligned}$$

Then there exists a sequence $\sigma \in EA^*$, which is the response of process A to inputs ρ , such that

$$\begin{aligned} (\alpha \wedge \beta' \wedge \delta') \uparrow EA \wedge \sigma \wedge \langle e \rangle &\in TA \\ \sigma \uparrow IA &= \rho \\ \sigma \uparrow v &= \langle \rangle. \end{aligned}$$

Let $\gamma' = \delta' \wedge \sigma \wedge \langle e \rangle$. Then $(\alpha \wedge \beta' \wedge \gamma') \uparrow EA \in TA$ by construction. Also we have $(\alpha \wedge \beta' \wedge \gamma') \uparrow E_k = (\alpha \wedge \beta' \wedge \delta') \uparrow E_k \wedge \rho \wedge \langle e \rangle \in T_k$ by construction. For other buffers with $l \neq k$, $(\alpha \wedge \beta' \wedge \gamma') \uparrow E_l = (\alpha \wedge \beta' \wedge \delta') \uparrow E_l \wedge \rho \uparrow E_l = (\alpha \wedge \beta' \wedge \delta') \uparrow E_l \in T_l$. Thus $\alpha \wedge \beta' \wedge \gamma' \in TW$ is indeed a trace. Its visible projection is unchanged, since $\gamma' \uparrow v = \delta' \uparrow v \wedge \langle e \rangle = \gamma \uparrow v$. Finally, $\gamma' \uparrow IW \uparrow \bar{v} = \langle \rangle$ since no inputs to W were added.

Case IV: $e \in EA - IA$

Apply lemma 6 with

$$\begin{aligned} \mu &\rightsquigarrow (\alpha \wedge \beta \wedge \delta) \uparrow EA \\ \nu &\rightsquigarrow (\alpha \wedge \beta' \wedge \delta') \uparrow EA \in TA \\ (\alpha \wedge \beta \wedge \delta) \uparrow EA \wedge \langle e \rangle &\in TA \\ (\alpha \wedge \beta \wedge \delta) \uparrow EA \uparrow v &= (\alpha \wedge \beta' \wedge \delta') \uparrow EA \uparrow v. \end{aligned}$$

Then there exists a sequence $\sigma \in (EA - IA)^*$ such that

$$\begin{aligned} (\alpha \wedge \beta' \wedge \delta') \uparrow EA \wedge \sigma \wedge \langle e \rangle &\in TA \\ \sigma \uparrow v &= \langle \rangle. \end{aligned}$$

Let $\gamma' = \delta' \wedge \sigma \wedge \langle e \rangle$. Then $(\alpha \wedge \beta' \wedge \gamma') \uparrow EA \in TA$ by construction. Also we have $(\alpha \wedge \beta' \wedge \gamma') \uparrow EB = (\alpha \wedge \beta' \wedge \delta') \uparrow EB \in TB$. Combining these, $\alpha \wedge \beta' \wedge \gamma' \in TW$ is a trace. The visible projection is unchanged, since $\gamma' \uparrow v = \delta' \uparrow v \wedge \langle e \rangle = \gamma \uparrow v$. Finally, $\gamma' \uparrow IW \uparrow \bar{v} = \langle \rangle$ since no inputs to W were added.

Since a γ' with the required properties can be constructed in all cases, the theorem holds. $\square$

4.2.3 Applying the Theory to Gypsy

Consider what it might mean to require the property of restrictiveness for a Gypsy procedure. We will not attempt to relate formally the semantics of processes described above to the semantics of Gypsy procedures. Instead, we will argue informally that there is a connection. We have modeled the design of SDOS in Gypsy as a collection of cobegun procedures communicating via buffers. We want to associate the input and output events described above with the (Gypsy) actions of sending and receiving from a buffer. Assertions about traces will then become associated with assertions about (Gypsy) buffer histories.

How might one verify in Gypsy the hook-up property of restrictiveness with respect to an arbitrary level l ? There are two basic problems:

1. The Gypsy embedded-assertion approach to stating correctness conditions will only allow assertions about the properties of single traces in isolation. More precisely, Gypsy embedded assertions are all of the form:

$$\forall \alpha \in T, P(\alpha)$$

where P is some predicate over event sequences which contains no quantifier over event sequences. Note that each program variable is a function of the past sequence of events, and so relations among program variables fall into this form. The property of restrictiveness is more complicated, since it requires one to show the *existence* of a trace, given the existence of another trace. The embedded-assertion method is not designed to do this.

2. The property of restrictiveness requires a process to be input-total, i.e., always ready to accept another input. This is never true of procedures described in Gypsy, which accept inputs only at "receive" statements.

The solution to the second problem is to associate one or more unbounded buffers with each Gypsy procedure of the design. The combination of a Gypsy procedure and its associated buffers forms an input-total process.

To solve the first problem, we use the theory of the previous section to decompose restrictiveness into WNI and other simpler conditions. Then we give a technique for demonstrating WNI using Gypsy.

Suppose a Gypsy program is composed of a cobegin of procedures, and a collection of unbounded Gypsy buffers visible to those procedures. Each Gypsy buffer will be associated with exactly one Gypsy procedure. Further suppose that each procedure is of the form:

```

loop
  await
    each i:integer, on receive msg from buffer[i];
    prog(msg);
  end;
end;

```

where each $\text{buffer}[i]$ is an unbounded Gypsy buffer associated with this procedure. The code 'prog' contains no "receives", and it may contain "sends" to buffers associated with other procedures.

Then we may (informally) apply theorem 2 to this situation, treating the Gypsy events of "sending" and "receiving" from buffers as the communication events referred to in the theorem. We can then claim that this procedure and the collection of buffers associated with it are restrictive with respect to view v if we can show:

- that each buffer preserves v : the map which takes the event "receive msg" into the event "send msg" is one-to-one and onto. Let a predicate *visible* be defined

on the set of messages, and let a buffer "send" ("receive") be an event in view v if and only if the message sent (received) satisfies *visible*;

- input-universality: this is guaranteed since all input events occur via the 'await' statement;
- input-liveness: this will be guaranteed if can show that 'prog' may terminate for each possible set of preconditions;
- synchronous determinism: this is guaranteed in any Gypsy procedure which makes no cobegins, either directly or indirectly, through calls to other procedures. This can be checked purely syntactically;
- weak non-interference with respect to view v for the procedure.

Since restrictiveness is a hook-up property, if we establish the above facts for each procedure, we can deduce that the entire Gypsy cobegin is restrictive.

Figure 4.2 shows schematically an example of hook-up in Gypsy. The example is of a closed system of two communicating processes. The large ovals in the figure represent the two processes to be proved restrictive. Within each process, all input events are delivered first to unbounded Gypsy buffers. There may be more than one such buffer per process. The buffers pass along their input messages to Gypsy procedures satisfying WNI. In the case of process A , the single internal procedure is WNI; in the case of process B , a collection of communicating procedures and subprocedures, each SNI, hook together to form a procedure that is WNI (SNI is composable, and SNI implies WNI). Once input-liveness is shown for each procedure, we know that both processes A and B are restrictive. Between the outputs of one process and the inputs of another, we may suppose there are arbitrary delays. This will be consistent with the semantics of Gypsy buffer communication if we assume that the delays do not change the order of the messages they deliver. Then each delay is simply an infinite buffer, and therefore restrictive. Each of the five processes in the figure is restrictive, so the entire system is also restrictive.

4.2.3.1 Verifying WNI using Gypsy

This section will describe our efforts to develop a technique to prove weak non-interference in Gypsy. The technique presented is not general; it is easy to find examples of Gypsy procedures that satisfy WNI which cannot be handled this way. However, we have successfully applied this technique to several examples of trusted type managers in the SDOS design.

4.2.3.1.1 Approach WNI states that for every trace α there is another trace α' which bears a certain relation to α . (see section 4.2.2.3). The first task in Gypsy, therefore, is to generate the traces α and α' . This involves, in effect, running two copies

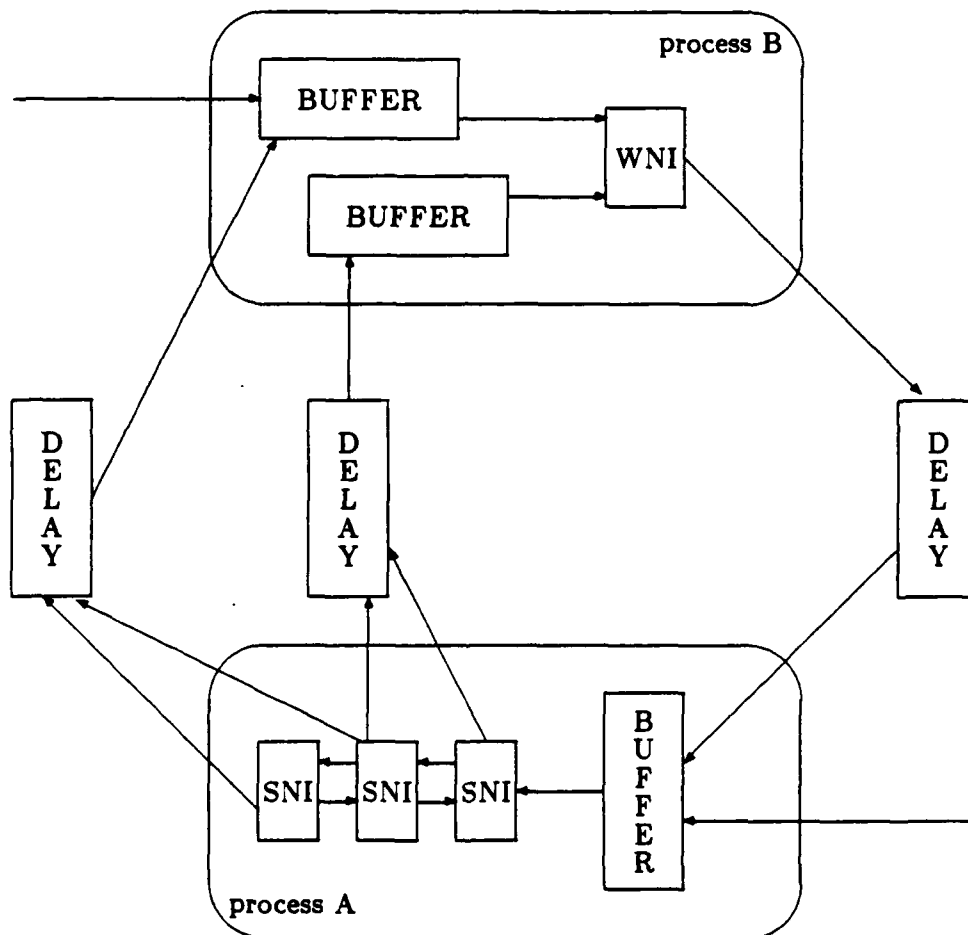


Figure 4.2: Schematic representation of a Gypsy cobegin of two procedures. The cobegin shown will be restrictive if the Gypsy procedures involved meet certain conditions. See text.

of the program simultaneously - one with all inputs which generates trace α and another with no high level inputs, i.e. $\alpha' \uparrow I = \langle \rangle$, for trace α' . The trace α' will be referred to as the reduced history. The verification is then a demonstration that $\alpha \uparrow I = \alpha' \uparrow I$.

Given the Gypsy specifications for a particular procedure, generating trace α is automatic since it is the normal execution of the procedure. The key to generating trace α' is to make a copy of the Gypsy specifications for the particular procedure. Not only the internal program variables but also the buffer variables must be duplicated. The new program and the variables therein will be called the shadow program and the shadow variables respectively. Now running the shadow program with no inputs greater than level I , would give us trace α' . But, since we want to compare histories, the traces must be derived simultaneously. Therefore, further constructions are called for. A *transformed program* is one in which the original and the shadow program have been amalgamated as per the guidelines (program transformation technique) to be described in the following section. The assertions in the transformed program would state, among other things, that the contents of the original buffer history, taking only events less than or equal to level I , will be the same as the shadow buffer history.

4.2.3.1.2 Details of the program transformation technique The Gypsy program for the SDOS design is a collection of procedures that are cobegun and communicate through infinite Gypsy buffers. Each Gypsy buffer will be associated with exactly one Gypsy procedure. For example,

```
cobegin
  procedure A(buffer_a,a);
  procedure B(buffer_b,b);
  procedure C(buffer_c,c);
end;
```

Further, let each procedure be of the form

```
loop
  await
  each i:integer, on receive message from buffer[i] then
    body;
  end;
end;
```

where each $\text{buffer}[i]$ is an unbounded Gypsy buffer associated with this procedure. The code *body* contains no "receives" though it may contain any number of sends to buffers associated with other procedures. Also, cobegin statements will not be allowed within the procedure since such a cobegin would introduce non-determinism and thus invalidate one of the assumptions of theorem 2 in section 4.2.2.6.

The way to generate the reduced history is simply to ignore those inputs of inappropriate level:

```

loop
  await
    each i:integer, on receive message from buffer[i] then
      if dominates(l,message.level)
      then
        body_sh;
      end;
    end;
end;

```

where *body_sh* = *body* with a disjoint set of variables and *dominates*(*l*₁,*l*₂) is a boolean function that is true if level *l*₁ is greater than or equal to level *l*₂. *l* is an arbitrary constant level.

Since we want to compare histories, we must run the programs simultaneously. The transformed program, therefore, is of the form:

```

loop
  assert (output from body) ↑ l = (output from body_sh) ↑ l ;
  await
    each i:integer, on receive message from buffer[i] then
      if dominates(l,message.level)
      then
        body;
        body_sh;
      else
        body;
      end;
    end;
end;

```

The security condition, expressed in the assert statement, is that all times the output from *body_sh* restricted to level *l* is the output from *body* restricted to level *l*. This is the weakest loop invariant to imply that the original procedure is WNI. The assert shows that the visible output sequences are equal; the visible input sequences are equal by construction. Since the assert holds at each input, the interleaving of inputs and outputs into the complete sequence of external events must also produce the same visible sequences when restricted to level *l* in both the actual and purged histories. It must also be pointed out that if MLS data structures exist in *body* then the loop invariant must include the fact that the shadow MLS data structures in *body_sh* are the MLS data structures in *body* restricted to level *l*.

Using the Gypsy Verification Environment to show the correctness of the above program fragment will result in a number of *verification conditions* (VCs) to be proved. Each of these VCs is associated with a particular execution path through the program

that begins and ends at one of the assertions. In the fragment above each path is one execution of the loop, beginning at the assert statement and returning to it. The number of paths produced and the complexity of their VCs are of great practical importance to the verifier, since the verification process becomes harder when either of them is increased.

Focus on the running of *body* and *body_sh* simultaneously in the case that dominates(*l*, *message.level*) is true. Let *new_body* refer to the interleaving of steps of *body* and *body_sh*. Running *body_sh* after *body* would be the simplest interleaving. This simplest interleaving results in more VC generation paths than *body* alone. In fact, if there were *n* paths in *body* then there would be n^2 paths in *new_body*.

We would prefer to find an interleaving that reduces the number of VC generation paths. For example, the fragment

if a then b else c end;

in *body*, which has only 2 paths, would transform to

if a then b else c end; if a_sh then b_sh else c_sh end;

in *new_body*, which now has four paths. The hypotheses in VCs that would come about are:

1. H1: a; H2: a_sh;
2. H1: not a; H2 a_sh;
3. H1: a; H2: a_sh;
4. H1: not a; H2: not a_sh;

It is often the case that *a* and *a_sh* are both computed using only values of variables that depend on visible inputs. Then we can additionally assert that *a* iff *a_sh*, and the pair of if ... then ... else statements can be interleaved in a different way, resulting in *new_body* of the form:

assert a iff x_sh;

if a then b; b_sh; else c; c_sh end;

The number of paths in *new_body* is now equal to that in *body*. Since the original and shadow programs refer to disjoint sets of variables, their steps can potentially be intertwined in any manner, so long as the order within each program is unchanged. The following rules show ways of interleaving the steps of various program constructs. Each step reduces the number of VC generation paths. Whether a particular rule is used depends on whether the new assertions generated can be proved.

Cases

- assignments

in *body*:

```

        a := b;
in new_body:
    a := b;
    a_sh := b_sh;

```

- casesplit

```

in body:
    case x is yi:
in new_body:
    assert x = yi iff x_sh = yi_sh;
    case x is yi:

```

- send to buffer

```

in body:
    send x to buffer1;
in new_body:
    send x_sh to buffer1_sh;
    send x to buffer1;

```

- if ... then ... else ...

```

in body:
    if x then y else z end;
in new_body
    assert x iff x_sh;
    if x then y; y_sh else z; z_sh end;

```

- function/procedure calls

```

in body:
    p := function(q);
in new_body:
    p := function(q);
    p_sh := function(q_sh);
in body:
    procedure(x);
in new_body;
    procedure(x);
    procedure(x_sh);

```

- loop

Any of the above constructs embodied within the loop could be transformed as above. Here we consider the interleaving of iterative loops with counters that index arrays, sequences, etc. If the counter and its shadow index equal sequences, we combine both loops into one and use the same counter for both. Then the above cases can be used to interleave the steps within the loop body. On the other hand, when the sequences are unequal the following more general transformation is recommended:

```

in body:
    i := 0;
    loop
        if i ge x then leave;
        else
            i := i + 1;
            actions_on(y(i));
        end;
    end;
in new_body:
    i := 0;
    i_sh := 0;
    loop
        assert x ge x_sh and
            i_sh = size(y(1..i) ↑ l) and
            y(i) = y_sh(i_sh);
        if i ge x then leave;
        elif dominates(l, y(i).level)
        then
            i := i + 1;
            i_sh := i_sh + 1;
            actions_on(y(i));
            actions_on(y_sh(i_sh));
        else
            i := i + 1;
            actions_on(y(i));
        end;
    end;

```

Before closing this section, a brief note on assert statements is probably in order. Consider the following specification segment:

```

assert A;
:

```



```

assert B;
:
assert C;
:

```

If information in A is needed to prove C, then A should be carried through every assert till C, i.e. the specification would look as under:

```

assert A;
:
assert B and A;
:
assert C;
:

```

The design specifications for the various type managers and the kernel are in Appendix A. Appendix B has the transformed specifications for the file manager that was verified using the above technique.

4.2.3.2 Gypsy Subprocedures

The Gypsy design of a process will often involve the use of Gypsy subprocedure calls. When the program transformation method for verifying WNI in Gypsy is used, we find that the subprocedure call can be handled in several different ways. This section will discuss a sufficient technique for decoupling the proof of non-interference for a procedure from the proof of non-interference of its subprocedures.

Consider a procedure of the form

```

procedure caller(inbuf, outbuf: buftype) =
begin
  var mesg: mesgtype;
  var v: vartype;
  loop
    receive mesg from inbuf;
    ...
    P(v, outbuf);
    ...
  end;
end;

```

where the ellipsis may contain sends to outbuf, but no receives. Let the subprocedure *P* be of the general form:

```

procedure P(var v: vartype; outbuf: buftype) =
begin
  body;
end;

```

where the body may contain sends to outbuf. We have shown a single non-buffer parameter, but our result could be generalized to any collection of parameters.

The program transformation approach to verifying WNI will produce the following transformed procedure:

```

procedure caller_trans(inbuf, outbuf, outbufsh: buftype) =
begin
  var mesg: mesgtype;
  var v, vsh: vartype;
  loop
    assert purge(outto(outbuf, myid), l) =
      purge(outto(outbufsh, myid), l);
    receive mesg from inbuf;
    if dominates(l, level(mesg))
    then
      ...
      P(v, outbuf);
      ...
      P(vsh, outbufsh);
      ...
    else
      ...
      P(v, outbuf);
      ...
    end;
  end;
end;

```

where the names ending in "sh" denote objects of the purged history, and the symbol "l" denotes an arbitrary level. In order to prove the assert statement in 'caller_trans', it will usually be necessary to relate the values of the "sh" variables to the values of the ordinary ones. This relation typically will show that 'vsh' is the same as 'v', except purged of contents which would result from processing at levels not less than *l*.

The assert will not be provable without more information about the properties of procedure 'P'. There are three ways in which more information about *P* can be imported into 'caller_trans':

1. Calls on P may be expanded in-line in procedure 'caller' by instantiating the code 'body' with the actual parameters substituted for the appropriate formal parameters.
2. An exit specification for P may be found which is sufficiently strong that security-relevant facts about the output of P can be compared for $P(v)$ and $P(vsh)$. It is sufficient, for example, if the output value of v can be expressed completely as a function of the input value of v .
3. The program transformation technique may be applied directly to P to derive security-relevant facts needed in the proof of 'caller_trans'.

There may also be other methods, but these three are the ones we are familiar with.

The second method is the simplest, and the most in harmony with the Gypsy methodology, since one can use the specifications for P directly in the proof of 'caller_trans'. However, finding a function expressing all the necessary facts about the action of P may be quite tedious. The third method is of interest in the remainder of this section; it allows security verification of P to proceed independently of the security verification of 'caller', without the need to find a functional form for P .

In many cases, the following conditions can be met:

- The elided code in the body of 'caller_trans' can be merged so that $P(vsh, outbufsh)$ is called immediately after $P(v, outbuf)$;
- A function 'call_level(v : vartype) : level' exists, associating a level with each call of subprocedure P ;
- A function 'purgev(v : vartype, l : level) : vartype' exists, which removes from v all data associated with levels not dominated by l .

The third condition may be trivially generalized if the single formal parameter v were to be replaced by a list of formal parameters.

The program transformation technique may now be applied to P itself to produce:

```

procedure P_trans(var v, vsh: vartype;
                  var outbuf, outbufsh: buftype) =
begin
  entry vsh = purgev(v,l) and
    purge(outto(outbuf,myid),l) = purge(outto(outbufsh,myid),l);
  exit vsh = purge(v,l) and
    purge(outto(outbuf,myid),l) = purge(outto(outbufsh,myid),l);
  if dominates(l, call_level(v))
  then
    body

```

```

        bodysh
    else
        body
    end;
end;

```

where 'bodysh' is the code 'body' with 'vsh' substituted for 'v' and with 'outbufsh' substituted for 'outbuf'.

To understand the reason for this transformation, consider the interaction of the caller procedure and its subprocedure *P* to be a sequence of message-passing events. Both caller and *P* are processes which obey the 'subroutine protocol': when one sends to the other, it surrenders control and waits for the other to send new messages (and control) back to it. Consider each transfer of control to consist of a stream of messages, at many security levels. This stream first passes the current contents of variable *v* and information about outbuf, and the stream is then terminated by a single message transferring control.

The contents of variable *v* may be thought of as a set of data items. Different items may be conceptually "labeled" with different security levels. The function 'purgev(*v*,*l*)' must strip away all items whose "label" is not dominated by *l*. The stream of messages which passes the contents of variable *v* is composed of messages for each data item in *v*, and the level of each message will be the join of the item's "label" with the level "call_level(*v*)". (A message in the stream is at least as sensitive as its contents, but also as sensitive as its very existence, which is "call_level(*v*)".) The final, terminating, message has level "call_level(*v*)", and a subprogram returning control sends this control message at the same level as the previous control message sent by its caller.

Verifying the transformed procedure 'P_trans' corresponds to proving the property of strong non-interference (SNI) for procedure *P*. This property says that for any level *l* and any history of message-passing events, a new history could be constructed by purging all events not visible at level *l*, including both parameter passing events and control events.

If the transformed procedure 'P_trans' can be proved, the proof implies that the following procedures could also be proved. Therefore, the specifications for these procedures can be manually imported into the proof of 'caller_trans'.

Single calls *P*(*v*, outbuf) in 'caller_trans' may be replaced by calls *P*1(*v*, outbuf), where

```

procedure P1(var v: vartype; var outbuf: buftype) =
begin
    entry not dominates(l, call_level(v));
    exit (assume purgev(v,l) = purgev(v',l) and
        purge(outto(outbuf, myid), l) =
        purge(outto(outbuf', myid), l) );
    'body' ;
end;

```

end;

Any pair of immediately juxtaposed calls to $P(v, \text{outbuf})$ and $P(\text{vsh}, \text{outbufsh})$ may be replaced by a single call to $P2(v, \text{vsh}, \text{outbuf}, \text{outbufsh})$, where

```

procedure P2(var v, vsh: vartype;
var outbuf, outbufsh: buftype) =
begin
  entry dominates(l, call_level(v)) and
    vsh = purgev(v, l) and
    purge(outto(outbuf, myid), l) =
      purge(outto(outbufsh, myid), l);
  exit ( assume vsh = purgev(v,l) and
    purge(outto(outbuf, myid), l) =
      purge(outto(outbufsh, myid), l) );
  'body';
  'body_sh';
end;
```

Of course, there is no automated Gypsy support for this replacement procedure. If it is more convenient, the procedure $P1$ may be proved directly with the code 'body', and the procedure $P2$ may be proved directly with the code 'body; bodysh'. In fact this method is slightly weaker than the one described, but may still export specifications strong enough to prove 'caller_trans'. We have appended the two conditions together in procedure 'P_trans' to show that a single program transformation step can produce the necessary verification conditions.

How are the functions 'call_level' and 'purgev' to be chosen? They should be chosen to support the proof of WNI or SNI in the calling procedure 'caller_trans'. Almost certainly 'call_level' will be chosen to equal 'level(msg)' in all cases. The function purgev will be chosen on the basis of the effective sensitivity level of data contained in "components" to v . For example, if v is a multilevel set of records, where each record was added in a separate invocation of 'caller', then purgev should remove from v all records added during invocation at levels not dominated by l . If v is a scalar whose content has been influenced by calls at level l' , then purgev should return the constant initialization value of v if l does not dominate l' .

Chaining of subprocedures is possible. In other words, subprocedure P may in its turn call on subsubprocedure Q . If Q is verified separately using the program transformation technique, then conditions about calls on Q may be imported (manually) into transformed procedure 'P_trans'.

4.2.4 Extensions to the Theory

In this section we describe several modifications to the theory of restriction. The first is merely a variation on how one can use the level fields of a message to represent level information and is completely consistent with the standard theory. The second is called input-limited restriction and it is used for hooking together processes that are only secure under certain limitations on the inputs. The last generalizes restriction by allowing some facts about higher level inputs to interfere with lower level processing. A generalized hook-up theorem is proved.

4.2.4.1 "True" Levels for messages

In our normal paradigm for security, each message event is associated with a level. This level indicates the sensitivity of the message and is used to guide the secure handling of messages. A process is restrictive if it stamps levels on output messages so that information does not flow from higher level input messages. Normally levels are associated with messages by recording the level in a special field of the message. However, if for some reason, the value in this field cannot be assumed to be correct, proving a process restrictive will not be enough to know that it is secure. We discuss two situations in which this arises.

In the first case we must use the "true" level of the message rather than the level field in the message when proving restriction. For example, a message from an untrusted process may not have a reliable level field. Therefore, its security label should be set depending on the true level of the sender. In this case we must prove restriction based on the 'true security level' and not the level in the label field.

We can also change the way messages are labelled to aid in building certain kinds of non-mandatory proofs. The basic idea is to use more than one level field. One field is used for proving restriction and the other is used to ensure higher invisibility of the transactions. (In practice, there may not need to be an explicit secondary field, since it may be computable from the content of the message.) A message may have some low level effect, so in order to show restriction we may need to assume the originating request is also low. But the actual information which is visible may be quite small. So by an auxilliary security argument we can more specifically say what will be visible, and hence more specifically describe why a low level message does not truly downgrade information.

4.2.4.1.1 Applications The first technique described above will be used in proving the restrictiveness of the message switch. When the message switch receives a message from a single-level process, it cannot trust the label supplied by that process, and must supply the correct, "true", message label itself.

One potential use for the second technique is to show more carefully the effects of a logout. It is possible to construct the Gypsy specifications for authentication so that the

apparent level of a logout is at the level of the user, while the mandatory label is low. We then show in each of the managers that handle logouts (and their consequent effects), that they are restrictive both in the case when we assume that the level of the logout is the true level and in the case when we assume the level is that in the mandatory level field. One exception to this: a new user of a terminal is able to see the effect of the logout acknowledgement for a previous user. The new user is now able to log in. This is where the mandatory level needed to be low. While the original logout request was in some sense low, its actual impact was just that a resource can now be determined to be available. In fact, even this is really only visible to another user if the logout is a consequence of a line release. We can then tell the user to avoid using a pattern of line releases which will reveal high level information. In the actual Gypsy specifications provided, we have not included the code to fully justify this argument. The reason is that the added degree of assurance gained is not sufficient to justify the added complexity of the proofs. It is anticipated that as the technology of security verification improves, that this sort of technique will be desirable.

4.2.4.2 Input-Limited Restrictive Hookup Theorem

4.2.4.2.1 Motivation It is sometimes the case that a process is restrictive only under the assumption that its inputs will satisfy some limitation. The process is input-total in that it will accept any input. However, it is only guaranteed to behave securely if the inputs are of a specified kind. Hence if we can prove both that a process is input-limited restrictive and that when connected its inputs will in fact be of the right form, then the process will behave "securely". In the following, we first give a formal definition of input-limited restriction and then we provide a justification as to why it is appropriate. We then argue that, under the appropriate conditions, this property is composable (i.e. that we can hookup two input-limited restrictive processes to form an input-limited restrictive process).

4.2.4.2.2 Definitions A condition IL is an **input limitation** on a sequence of inputs if and only if (s satisfies IL implies that every initial subsequence of s satisfies IL).

A process is **input-limited restrictive** with respect to input limitations IL_i on each input line i if and only if

for any $\alpha^{\wedge}\beta^{\wedge}\gamma$ that is a history of the process, such that β is all inputs (possibly an empty sequence), and with the input history of $\alpha^{\wedge}\beta^{\wedge}\gamma$ from line i satisfying IL_i for every i , and for any modification of $\beta^{\wedge}\gamma$ to $\beta'^{\wedge}\gamma'$ by adding or deleting input messages $\leq l$, such that the input history from line i of $\alpha^{\wedge}\beta'^{\wedge}\gamma'$ still satisfies IL_i for every i , it is possible to find a γ'' which is the same as γ' except on the outputs with level $\leq l$, so that $\alpha^{\wedge}\beta'^{\wedge}\gamma''$ is a possible history.

Note that this is equivalent to ordinary restriction if we demand all the IL_i 's be true. Under this case, to see that this property implies restriction, observe that the modifications in the hypothesis of restriction are just a special case. Conversely, to see that restriction implies this property under this case, first remove all of the high level inputs in γ and then use induction to successively add the high level inputs of γ' (making adjustments to the high level outputs).

4.2.4.2.3 Justification The justification of the standard definition of restriction is that a low level user cannot deduce anything about the possible high level inputs. With the new input-limited definition we cannot claim this result. However, if an entire system can be shown to have this property, then the only input limitations will be limitations on users (and other input devices). Therefore, all that can be deduced is that high level inputs must be arranged with respect to the low level history so that the combined history is possible. For typical applications of this property, this will not pose any security problem. For example, suppose an input device on a line which is designated as top-secret only sends top-secret inputs. Now suppose the fact that the line was to be used in this way was made public. Then an unclassified user could infer that no secret message will get sent down that line. But this is really just learning unclassified information. In the applications for this project, the input limitation will be that a user will not do a writedown. An inference based on this fact is clearly not a security leak.

4.2.4.2.4 Hookup Theorem We make the following claim.

Claim: Input-limited restriction is composable, i.e., it satisfies the following hookup property.

Let P_1 have input lines i_1 to i_n and output lines o_1 to o_n . Let P_2 have input lines j_1 to j_m and output lines p_1 to p_m .

Let P_1 be a process with input limitations: I_1 to I_n on lines i_1 to i_n and let P_2 be a process with input limitations: J_1 to J_m on lines j_1 to j_m .

For simplicity, let us suppose that we are trying to hook up P_1 to P_2 with i_1 to p_1 and j_1 to o_1 .

Suppose P_1 and P_2 satisfy input-limited restriction and have the following output-limitations:

- In P_1 we can prove: I_1 and I_2 and ... and I_n for i_1 to i_n respectively $\rightarrow J_1$ for o_1 (i.e. for any trace t whose inputs satisfy I_1 to I_n , the outputs on line 1 satisfy J_1).
- In P_2 we can prove: J_1 and J_2 and ... and J_m for j_1 to j_m respectively $\rightarrow I_1$ for p_1 .

Now, let I_2 and I_3 and ... I_n and J_2 and J_3 and ... J_m be the combined input limitations on the joint process. Then we claim that the joint process is input-limited restrictive with respect to these conditions.

Notice that for any history $\alpha^{\wedge}\beta^{\wedge}\gamma$ which satisfy these limitations, we can use the output assertions to show I_1 and J_1 hold on the internal signals (by induction).

Given the suppositions above, a proof that the processes will hook up and preserve input-limited restriction follows essentially the same form as the hookup theorem for standard restriction. We sketch the basic idea.

Let $\alpha^{\wedge}\beta^{\wedge}\gamma$ be any history of the joint process which satisfies the input limitations (and where β is inputs), and let $\beta'^{\wedge}\gamma'$ be a modification to $\beta^{\wedge}\gamma$ gotten by adding or deleting high level inputs (ie. not below 1) such that $\alpha^{\wedge}\beta'^{\wedge}\gamma' <$ still satisfies the input limitations.

Now, alternately fix the histories on side P_1 and P_2 . At the beginning of each fixed stage we have a history with extra high level inputs added and we can apply input-limited restriction to adjust it. Note that we must use the output assumptions to know that the new input history is still allowed. As in the proof of the standard restriction theorem we must make sure that we make forward progress in accomplishing the adjustments. This is done in exactly the same way. (This is the reason for having a separate β and γ in the definition.) As this is a straight-forward transformation of the proof, we direct the interested reader to [Ulysses 87] to see how this is done in the standard restriction theorem.

4.2.4.2.5 Combining with the WNI Approach In theory, it should be possible to combine this idea with the WNI methodology. Assume the input-limitations on the input ports hold at the beginning of the execution history. Then proceed to use the WNI techniques and where needed use the imported assumption of input-limitation. Also prove the required assertions about the output ports. It may be useful to use auxillary variables to keep track of what has transpired (i.e. state variables). Unfortunately, there is no simple way to do this in Gypsy, because one cannot assume some condition on all of the inputs. Nevertheless, it is hoped that with improved technology such a scheme will be possible.

4.2.4.2.6 Examples The principal purpose of this theorem is to handle possible hookup between system components which do not have labeling. For this project, the principal application of this result is the hookup between users and the system. It may also be useful in eliminating code which will not be used, when a particular combination of inputs cannot occur. We will briefly discuss the possibility of eliminating code for the kernel which would handle multiple **SetProcessBinding** operations for the TIP.

Another possible case where this result would be useful is for communication lines which are assigned one fixed level. In such a case, we should be able to prove that all inputs and outputs along this line are at that level and so should be able to prove restriction without directly checking labels in the actual code. (A slight efficiency gain.)

It may also be convenient to build other kinds of processes which do not have to label all of their messages with an explicit security level.

4.2.4.3 Limited Insecurity

4.2.4.3.1 Motivation Recent work in computer security has centered around the notion of information not flowing in certain ways. For instance, there have been attempts to make precise the idea of information not flowing from one level to another and to verify this property of models of actual systems.

A limitation of this approach is that in most real systems information does flow even between levels where it's not supposed to. This makes it difficult to prove that it doesn't.

One example is that of downgrading. It is common that for the sake of flexibility a system will include a downgrading facility. The effects of this high-level act are clearly visible to a lower-level user, as they are supposed to be.

There is also the case of limited access resources. Some system components can be accessed by only one user at a time, and will return a reject message if another tries to do so. So if a high-level user gets there first then this might be visible later to anyone.

Slightly different from these is the instance of uncertainty of the level of information. When someone tries to log on, it is unclear at first what should be the level of that message. There are any of a number of ways of formally labeling this message, but its real effect will be at the actual level of the user, which can be determined locally only after receiving the acknowledgement from the password database.

We would like to generalize the current theory to handle these cases, in part because some of the work would then be done for us. Yet this desire is not just pragmatic, it also follows from the ideas themselves. The intuition behind restrictiveness, the best current example of a security property, is that all the information possibly available to a user at level l is unaffected by the inputs at levels not less than or equal to l . A crucial part of the formalization of this property is the restriction operator $\uparrow l$, which takes a sequence of messages and returns the subsequence of those messages at a level less than or equal to l . This is used to define the notion "everything that an l -user could possibly know". But if some high-level information does not remain strictly above l , then $\uparrow l$ is not the right restriction operator.

At this point one could attempt a simple generalization of $\uparrow l$. Instead of just throwing away a message with a high label, one could replace it with a message containing all of the information less than or equal to l . In the examples above, the message "downgrade X" would be replaced by "write(contents(X))", possibly with certain items, such as the identity of the user, also deleted. In a limited access process, the high-level command "I want you to do such-and-such" would be replaced by "Somebody wants to use you for something". For a login attempt, however it is actually labeled by the system, we would consider it at the level of the user, assuming that the users and their levels don't change.

Allowing for alterations of messages such as these, we could then define the view of a system to a user, $\uparrow v$, a function from traces to traces, inductively on the length of the trace: $\langle \rangle \uparrow v = \langle \rangle$, and $ax \uparrow v = (a \uparrow v)m(a, x)$, where m is some appropriate

function. Presumably $m(a, x) = x$ if the level of $x \leq l$, and is as suggested by the examples otherwise. Note that we allow the previous history as a parameter to m , as in the downgrading example.

Such an attempt, while mathematically sound, is in some measures inadequate. In the downgrading example, while the locus of information transfer is restricted to that one message, the content of the transfer is really unclear. On what does "contents(X)" depend? For limited access processes, presumably most of the calls to them do not interfere with one another, so by noting them all we carry around a lot of baggage which makes it seem as though more information is being transmitted than actually is. Regarding logins, we had to make the assumption that the users and their passwords are constant, which is related to the problem that the suggested function m cannot be computed locally.

All of these problems are related to the fact that we know what high-level information is available only retroactively. The downgraded message "write(contents(X))" should depend only on the writes to the file. We would like to retain those writes in $a \uparrow v$ and make m a function not of a and x but of $a \uparrow v$ and x . But any file might be downgraded, and saving the writes to all of them would defeat the purpose. m knows to retain writes to a downgraded file only retroactively. Similarly, the only holds on single-user processes of importance are those that later cause a reject message. Therefore m should retain the traces of only those requests, necessarily retroactively. For attempted logins, the situation is the clearest: the level of a login attempt is the level eventually assigned by the acknowledgement.

Another advantage of this more accurate modeling of real systems is that we are interested in not only what data somebody gets, but also when. As an example, when downgrading we would like to know not only that what the low-level user saw depended only on the writes to the file, but also that it didn't depend on even that much until a certain time.

As before, the restriction operator can be defined inductively, using the auxiliary function m . This time, though, whether to append $m(a, x)$ or not may depend on later messages in the sequence. Also, $m(a, x)$ is to be uniformly computable from $a \uparrow v$ and x , so we know that the information leaked is contained in what we have been saving.

In what follows, we present a formalization of this latter approach. Examination of the details of this program reveals manipulations not found in the development of standard restrictiveness nor suggested by the intuitions above. We will try to explain and justify them as they occur.

4.2.4.3.2 Processes and Views A process P is as defined in section 4.2.2.2.

A limitable process is a process P , along with a subset N of E and a function $m : E^* \times E \times E^* \dashrightarrow E$. These extra objects N and m are enough to allow us to define the restriction operator $\uparrow$ and associated function m described earlier, to allow for modeling limited information flow. To do this we first need some more definitions.

For $a \in E^*$ and x an occurrence in a of an event, $a < x$ is the initial segment of a before x and $a > x$ the final segment of a after x . $\langle \rangle$ is the empty sequence, which we also assume to be an event. This way m can return $\langle \rangle$, allowing for $am(b, e, c) = a$. $\uparrow S : E^* \dashrightarrow E$ is the standard operation of restriction to a set S of events, defined inductively: $\langle \rangle \uparrow S = \langle \rangle$, and $ae \uparrow S = [a \uparrow S]e$ if $e \in S$, $a \uparrow S$ otherwise.

As described above, $\uparrow v$ takes a sequence a and replaces each event with its low-level content (as given by m). So $a \uparrow v$ is actually defined using an auxiliary notion $\uparrow v, a$, itself defined inductively on the events in a : $\langle \rangle \uparrow v, a = \langle \rangle$, $a'e \uparrow v, a = [a' \uparrow v, a]m(a' \uparrow v, e, a > e \uparrow E/N)$, $a \uparrow v = a \uparrow v, a$.

Some explanation is in order. In general, m is the identity on some set S , such as the events at or beneath a given level. If m returns $\langle \rangle$ off of S , then $\uparrow v = \uparrow S$. Since we want to allow for some information to trickle through, we have m possibly extracting some information from an event e . On what parameters should this extraction depend? Clearly it depends on m itself, which is assumed to be public knowledge. It should also depend on the previous history, or at least that part which is potentially visible, $(a < e) \uparrow v$, and also the current event e . It also must depend on future events, $a > e$, as described. But if we allow $a > e$ as a parameter, we defeat the purpose of trying to pinpoint the influences upon $\uparrow v$. Using $a > e$ as a parameter, we might permit highly classified information that it contains to trickle through. Therefore we select a presumably large body of events N to be the neutral events. They don't have the power to influence decisions about information flow. We focus all potential factors into the set E/N of non-neutral events.

Notice that we understand E to be sufficiently abstract. Sometimes m will clear its middle argument of much of its information, leaving something which could never be an actual message in a real system but which we consider an event. For instance, m might remove the client and the level from a downgrade message, leaving only that a certain file is to be downgraded. Such an event might never appear in any trace in T by virtue of its ungrammaticality, but we still consider it an event since we need it in the pseudo-histories $a \uparrow v$.

4.2.4.3.3 Examples The problems adduced as motivation were the login procedure, limited access resources, and downgrading. By way of illustrating this approach, we show how to express what is actually happening in these cases using our language.

To model the login, we consider a system with three components: a human user, the local host, and the login authenticator. The human's language includes the output "login request" at level X , the inputs "request approved" at each level l except X , and the input "request denied" at X . The host has all of those events with inputs and outputs reversed, along with the output "login check" at X , inputs "check approved" at each l except X , and the input "check denied" at X . The authenticator has the "check" events of the host, with inputs and outputs reversed.

A login attempt would consist of a request initiated by the human and passed along

to the authenticator. This is at level X since so far no one outside of this small group can know anything about this sequence. The authenticator then consults its database, and either approves the login at a fixed level, or denies it again at an isolated level. This reply is then passed along to the human.

How would we define m to represent the view at level l ? Requests and checks are invisible if they have not yet been confirmed, so $m(a, \text{"login request or check"}, \langle \rangle) = \langle \rangle$. Once the check is approved at level l , the check-event that caused it is visible at l : $m(a, \text{"login check"}, \text{"check approved at } l") = \text{"login check at } l"$. Note that at this point "login check" is visible at l , while the "login request" that caused it is still at X , invisible to l . This is for reasons of coherence. The host now knows enough to reclassify the request; that is, m_{host} could use the non-neutral event "check approved" to reclassify the request it received from the human, but the human couldn't. To retain the coherence of the local m functions, the original request cannot yet be affected. The next event, though, is that the host transmits "request approved at l " to the human, and both processes reclassify the initiating request to l : $m(a, \text{"login request"}, \text{"request approved at } l") = \text{"login request at } l"$. The neutral events are everything but the approvals.

For a limited access resource, consider a file accessible to at most one user at a time. The languages for the clients each include outputs open, close, read, and write, and inputs confirmed and denied, at all levels. The language for the file is the same, with inputs and outputs reversed. The file will confirm an initial "open", then confirm any future sequence through the first "close", and wait to confirm the next "open". Anything else it denies.

When there is no leak, it suffices to use the standard restriction operator: $m(a, x, b) = x$ if level $(x) \leq l$, $\langle \rangle$ otherwise. The only time there is a leak is when l tries to open the file and either it is currently being used by someone $\not\leq l$ or, following an earlier denial, the request is now confirmed. In the first case, the denial is tagged with an identifier for the currently operative "open". This is necessary so that in the inductive definition of $\uparrow$ we know exactly which "open" to retain. $m(a, \text{open}(\text{tag}_0), \text{denial}(\text{tag}_1)) = \text{open}$ if $\text{tag}_0 = \text{tag}_1$, $\langle \rangle$ otherwise, and $m(a, \text{denial}(\text{tag}_1), \langle \rangle) = \text{denial}$. Observe that m does strip off some information from $\text{open}(\text{tag}_0)$ and $\text{denial}(\text{tag}_1)$, since all that matters to the latest request is that somebody somewhere already has it. The second case is handled similarly, with the confirmation tagged with an identifier for the close that made it available. The neutral events here are everything except the denials.

For downgrading, the non-neutral message is "downgrade(X)". It makes visible the previous writes to X , removing all information such as client identities and levels from the writes and leaving only the content: $m(a, \text{write}(\text{tag}), \langle \rangle) = \langle \rangle$; $m(a, \text{write}(\text{tag}), \text{downgrade}) = \text{write}$.

4.2.4.3.4 Generalized Restrictiveness Given $\uparrow v$ as above, there is a corresponding notion of restrictiveness, of limited similarity to the standard one:

$\forall a, a' \in E^*$ and $x \in N$, if

$$a \uparrow v = a' \uparrow v \text{ and} \\ ax, a' \in T$$

then $\exists b' \in E^*$ and $\exists y \in E$ so that

$$ax \uparrow v = a'b'y \uparrow v, a'b'y \in T, \\ b' \uparrow I = \langle \rangle, \text{ and } b'y \in N^*.$$

We can assume without loss of generality that $a \uparrow v = a'b' \uparrow v$, simply by truncating the hypothesized $b'y$.

A restrictive process is a limitable process which satisfies restrictiveness.

First we argue, necessarily informally, for why this is a useful property to use. Then we discuss its relationship to standard restrictiveness.

A sane notion of security is non-deducibility. A certain set of events w is secure from the view v if:

for any trace a and sequence $u \in w^*$
there exists a trace b such that
 $a \uparrow v = b \uparrow v$ and $b \uparrow w = u$.

With this property, a viewer with access only to $a \uparrow v$ can deduce nothing about $a \uparrow w$. Usually the information we want secured from v are the inputs not in some set I . In this context (also assuming $\uparrow v = \uparrow I$), these ideas are intuitive, precise, and their formalization is implied by restrictiveness.

In our more general setting such simplicity does not work. We might try to have w be those inputs with no v -effect. For starters we want more than that. If an input has a v -effect then it would not be in w , but if two have the same v -effect then we would not want to be able to distinguish between them. Even more seriously, "inputs with no v -effect" can not be well-defined, since m depends on the previous and future history a as well as the current message x . Maybe sometimes an input is visible and other times not.

Our way to handle such problems, especially the second, is to consider deducibility of information in context, as a trace is being generated. The system is secured from deducibility if we cannot predict the future, nor find out that a previously reasonable guess as to the actual history was incorrect. This is meant to be necessary only when all the new events are neutral, so we can assume as much. That is, suppose that the real history a has been unfolding, and we have guessed that the actual history is a' , based on our view v : $a \uparrow v = a' \uparrow v$. Then we are given the opportunity to guess those inputs with no v -effect, using only neutral events. Think of unrolling more of a' until all inputs before the next v -visible event occurs. In response, more of a is revealed, up to the next v -event, and including only neutral events. Note that we still have $a \uparrow v = a' \uparrow v$. Then the v -event x is revealed. Since it is also neutral, there is a way of extending a' to catch up with this new event. Without changing our earlier guess,

nor our arbitrary prediction about future inputs, we can extend a' by neutral v -invisible non-inputs b' , and then another neutral event y visible to v . The nature of y cannot be restricted beforehand, since m may be *one - to - one*, determining y completely. Still, in the general case we have circumscribed those events about which we can deduce something to those that are v - *visible*. Of course, given a particular m to analyze we can hope to do even better.

The assumption that all new events in sight are neutral is necessary. Suppose that a and a' are the same except that a includes a session in which a high-level user writes a file. If we extend a by a (non-neutral) "downgrade x ", that will affect the beginning part of $ax \uparrow v$. There's no way that the beginning part of a' can be so affected by any extension. If a non-neutral event is introduced, we may have to revise our earlier guess. As phrased above, we give up the game. A direction for future research is to examine what changes we might be able to introduce to a' to retain some extension property.

Our restrictiveness implies a limited form of standard restrictiveness. Using standard notation, to show standard restrictiveness, we are given certain a, b, c, a' , and b' , and have to find a c' with no inputs out of v . If c has non-neutral events this may not be possible, so assume it doesn't. Consider the events of c one by one. Use our restrictiveness for each to find an appropriate extension with only neutral events, and at most one input out of v , that one being visible. So we can find a c' , not with no *non - v* inputs, but whose only *non - v* inputs are v - *visible*, always avoiding E/N . This is the best we could hope to do, given the set-up, and indeed it works.

4.2.4.3.5 Coherence and the Hook-up If P_0 and P_1 are two processes, they cohere if each common event is an input to one and an output from the other. If they cohere, we can define the hook-up $P_0 \# P_1$ as having events the union of the two sets of events, inputs the union of the inputs minus the common events, outputs the union of the outputs minus the common events, and traces $\{a : a \uparrow E_i \in T_i\}$. If the P_i cohere then $P_0 \# P_1$ is a process.

If the P_i are limitable processes, with associated functions m_i and sets N_i , they cohere if

- they cohere as processes,
- $\text{intersection}(N_0, E_1) = \text{intersection}(N_1, E_0)$,
- for $a, b \in E^*$ and x a common event,
 $m_0(a_0, x, b_0) = m_1(a_1, x, b_1)$
 where, $a_i = a \uparrow E_i$, and similarly for b_i and
- if x is not in $E_{(1-i)}$ then $m_i(a_i < x \uparrow v, a_i, x, a_i > x \uparrow E/N)$ is also not in $E_{(1-i)}$.

If the P_i cohere as limitable processes, we can define the process $P = P_0 \# P_1$ by the first coherence property. Let $N = \text{union}(N_0, N_1)$. By the second clause, we don't lose any

non-neutral events. Let $m : E^* \times E \times E^* \dashrightarrow E$ be $m(a, x, b) = m_i(a_i, x, b_i)$, where x is in E_i . This is well-defined by the third requirement, and induces $\uparrow v : E^* \dashrightarrow E^*$. By the last, $a \uparrow v \uparrow E_i = a \uparrow E_i \uparrow v$.

Incidentally, the final clause is not just an technical convenience. It is necessary for security reasons. If $m_0(a, x, b)$ is a low-level input from P_1 , but $x \notin E_1$, then P_1 does not know to cover up for P_0 's lie. This informal leak can be expressed formally.

4.2.4.3.6 Hook-up and Generalized Restrictiveness We would like to have that the hook-up of two (generalized) restrictive processes be (generalized) restrictive. This is not true, as the following example shows.

Let $E_0 = \langle I', I_2, O_3 \rangle$, $E_1 = \langle I'', O_2, I_3 \rangle$. $I_2 = O_2$, $I_3 = O_3$, and all other symbols are distinct. A symbol with an I is an input, an O is an output. Let $T_0 = I_0^*$ union $I'E_0^*$; $T_1 = I''I_1^*$ union $E_1/\langle I'' \rangle E_1^*$. Let $N_i = E_i$, and $m(a, x, b) = \langle \rangle$ if $x = I'$ or I'' , I/O_2 otherwise.

It is easy to check that each P_i is a process (input-total, closed under initial segments, and disjoint inputs and output) and is restrictive. Furthermore, the processes cohere and the m_i cohere. Nonetheless, $P_0 \# P_1$ is not restrictive. Let $a = I'$, $a' = I''$, and $x = O_3$. Notice that the aspect of retroactive changes is irrelevant here; even in the simpler case of replacing a message x by $m(x)$ we would have the same example.

The problem is that we need a certain amount of coordination between the processes. Each process agrees on what the restricted trace should look like, and can accommodate that with a real trace, but each insists that the real trace contain an input to itself. Neither is willing to put out.

Therefore, we say that a process P puts out if, whenever $m(a \uparrow v, x, \langle \rangle)$ is an output, $ax \in T$ and $x \in N$, there exists b and there exists y so that $aby \in T$, $ax \uparrow v = aby \uparrow v$, $b \in (N/I)^*$, and y is a neutral output.

Theorem: If P_0 and P_1 cohere as limitable processes, and each is restrictive and puts out, then $P_0 \# P_1$ is restrictive and puts out.

Indication of proof: First we show restrictiveness.

Given a, a' , and x , we must find appropriate b' and y . If x is not in $E_{(1-i)}$ then apply restrictiveness to P_i only. This produces b'_i and y_i . Let $b' = b'_i$ and $y = y_i$. The important points to note are that $P_{(1-i)}$ finds this acceptable because all of the shared messages in b'_i are inputs to $P_{(1-i)}$, which is input-total. Also, $\uparrow v$ is unaffected on a' because b'_i contains only neutral elements.

If x is a shared event, let i be such that $m(a \uparrow v, x, \langle \rangle)$ is an output for P_i . Use restrictiveness on $P_{(1-i)}$ to get $b'_{(1-i)}$ and $y_{(1-i)}$. Extend a' by $b'_{(1-i)}$. Any new common event is an input to P_i . Now apply restrictiveness for P_i to a_i and a'_i followed by these new inputs. The latter is a trace by input totality and has the same view as a_i by the coherence of m_0 and m_1 . Extend $a'b'_{(1-i)}$ by b'_i . If y_i is an output, extend again by y_i .

If not, use the putting-out property. This yields b_i'' and y'' , which can be so appended.

The putting-out property is even easier to check. If $m(a \uparrow v, x, \langle \rangle)$ is an output, then x is not a shared event. Therefore one can apply putting-out to the P_i such that x is in E_i .

4.3 Verification of the FTLS

One of the goals of this project has been to formally verify that the SDOS design meets the requirements of the SDOS security policy. In order to do this, both the policy and the design must be stated in precise language. The design was expressed first in section 3.1, but again in the more precise language of Gypsy in section 4.1, the Formal Top-Level Specification (FTLS). The security policy was stated first in section 2.1, but again in the more precise language of possible event histories in section 2.3, the Formal Model. The work that has been done toward proving the FTLS to be a correct implementation is described in this section.

Virtually all of the formal verification work accomplished during this project was directed toward proof of multi-level security of SDOS. In this work, we have used extensively the theory of multi-level security developed in section 4.2. Also, for proving the restrictiveness of SDOS components in Gypsy, we have sometimes used the program transformation technique developed in that section.

Our original intent was to provide an explicit mapping from all of the constraints of the Formal Model to constraints expressed in Gypsy. However, little work was directed toward the constraints other than those on mandatory information flow. There are several reasons for this, some theoretical, some practical. Other properties required by the security policy were considered either straightforwardly provable but ancillary to information flow security, or beyond the scope of the project.

Properties which fell beyond the scope of the project did so because there was no theory which could be used to reduce their proof to manageable complexity, and to ensure that all supporting assumptions were proven. The formal model states these properties for the system as a whole. They could be decoupled into properties for individual hosts; and these into properties of individual processes, etc. As an example, the restrictiveness of the entire system is decomposed into the restrictiveness of each component, using established theory. In principle, the configuration and discretionary policies could be decomposed likewise. However, in practice, this decomposition involves too much detail to be done quickly and with a reasonable assurance of correctness.

4.3.1 Overview

The proofs are organized as follows:

1. We will prove the File Manager is restrictive, using the methodology of 4.2.3.1

2. We will prove the Catalog Manager is restrictive, arguing that the methodology of 4.2.3.1 is sufficient.
3. We will discuss the hypotheses for the processes involved in authentication.
4. We will prove the Authentication Manager is restrictive.
5. We will prove the TIP is input limited restrictive and will show what output property we can export to users.
6. We will briefly mention how to implement the filter.
7. We will discuss the choice of protocols used in authentication.
8. We will show that the user and the TIP can be composed.
9. From the above and assumptions about restrictiveness of the rest of the system we can conclude that the system is input-limited restrictive.
10. Then we show correct authorization. Improper logins are not accidentally considered okay. While this is not a mandatory access control, it is vital for a secure system.
11. We draw conclusions about the security of the system.

4.3.2 Verifying the File Manager design

It has been proved that the file-manager top-level design meets the composable security property of Restrictiveness. The proof was done in Gypsy using techniques discussed in the section 4.2.3.1.

4.3.2.1 A Brief Recap of the design

All open-to-read requests will be honored if the level of the request dominates the level of the object. The actual file would be copied onto a ghost file at the level of the request to open-to-read. e.g. if a top-secret client wishes to open a secret file to read, the file-manager would create a ghost file at top-secret which would be a copy of the secret file. Subsequent read requests by the top-secret client would be mapped onto this top-secret ghost file.

All write requests have to arrive at or above the level of the object. But open-to-write requests would be honored if and only if level of the request equals level of the object. Also, the writes take place onto the actual object, no ghosts are involved. If more than one client issues a valid open-to-write request, the request that is first received is honored. The other request is turned down for reasons that the file is in use.

All "up" operations, like writeup, are allowed if the client sets the "up" bit in the original invocation. Also the reply for these "up" operations from the object-manager

cannot reach the client. (The client will get a cursory response from the message switch that his request is being acted on). This is because the kernel would, as part of the ReadSDBEntry operation, reset the level of the request to the level of the object. Therefore, managers handling the request would be at the level of the object (or be MLS) and hence cannot communicate with the client.

4.3.2.1.1 Intuitive Understanding of the Proof Before delving into the Gypsy proof that the file-manager specifications meets the security policy, it would be helpful to gain an understanding of why the design is secure. Our security policy, in effect, translates into the following: For lower level access requests, the system should behave as if there are no higher level accesses in progress.

As per our design, all valid reads are handled by creating ghost files at the level of the request. So multiple read requests pose no problem. Writes are made directly onto the object. All valid write requests have to be at the level of the object. It is not a violation of the policy for a second valid write request to learn that the object is being written into. Therefore, we elected to refuse the second write request on the grounds that the file was being written into. Read and write requests to the same object would pose no problems because the read would get a ghost file without interfering with the write request. It is also relevant to note that the file-manager always responds at the level of the invocation.

An alternate design that is also secure would call for reads to take place with the actual object and writes to be handled with ghost files. These ghost files would have to be internal to the file-manager (not be a part of the local ODB). Write ghosts would also have the additional overhead of having to be written over the actual object once the write operation is complete and the actual object is free.

4.3.2.1.2 Experiences with the verification The techniques described in section 4.2.3.1, to prove security properties in Gypsy were exercised successfully on the file-manager design. The design specifications are in Appendix A and the transformed specifications are in Appendix B. There were no major deviations from the stated program transformation technique. The loop invariant on the MLS data-structures, however, was handled differently from the scheme described in section 4.2.3.1. Defining a purge function (a function to project out values less than or equal to level l) to operate on MLS data structures was a non-trivial task. Therefore, instead of defining such a function, the loop invariants were stated in the form of equivalence of values returned by functions to which these MLS data structures were parameters. For example, if `openfor` is an MLS data structure and `has_access` is a function that has `openfor` as an input parameter, then the assert statement would read:

```

 $\forall$  call:sendmessage
    (has_access(call, openfor) if and only if has_access(call, openfor.sh));

```

The procedures called in the file manager were of the form where exit specifications

could be stated in terms of the input parameters and therefore posed no problems in the proof (see section 4.2.3.2 for details about handling subprocedures in the overall proof scheme).

As the file manager represents the only component completely verified using the program transformation technique, we will close with a few comments about Gypsy and the use of our techniques to verify the security policy. The time required for VC generation was quite large, and far out of proportion to the conceptual difficulty of the security properties being proved. This results from most variables in the program undergoing parallel updates in both the actual and the shadow histories, and the need to prove that parallel updates leave the relation between the actual and shadow variables intact. The theorem prover needed human intervention for many trivial steps in these proofs. Also, we found a number of bugs in the Gypsy environment during the course of this effort, and although the support and maintenance of Gypsy was quite helpful, the presence of these problems was a frequent irritation, and we often found ourselves contriving ad hoc ways to work around them.

The technique described in section 4.2.3.1 was somewhat tedious to follow. Generating shadow variables and carrying out the program transformation is less than desirable. But this labor intensive task could be automated, thereby weeding out the drudgery from a conceptually simple technique. We have not attempted the automation as that effort was not called for as part of this project. However, success in verifying the file manager design establishes the feasibility of our technique.

4.3.3 Verifying the Catalog Manager Specification

The catalog manager defines the SDOS operations that can be invoked on directories. The Gypsy specification for the catalog manager can be found in Appendix A, and an informal description of that specification in Section 4.1.4. It is the only example we have specified of a replicated MLS object manager, and is noteworthy in that regard since the concurrency control algorithm used does not interfere with multi-level security.

We have not carried out the Gypsy proof of the catalog manager using the program transformation technique, but we expect that it can be done straightforwardly using the methods of section 4.2.3. We assume in what follows that these methods are familiar. At the top level of the program, in main procedure "catalog_manager", the variables "pending_ops" and "used" contain the only part of the manager's state that is stored between one pass of the loop and the next. Each of these is a structure containing components at many levels: "pending_ops" contains records of each transaction in progress, with the level of each record being the level of the message that began the transaction; "used" contains transaction numbers, with the level of each number in use being the same as the message that created the transaction it names. To use the program transformation technique, it will be necessary to prove as invariant the fact that the purged parts of each of these structures, i.e., the components with levels less than some fixed l , are equal in the two histories.

The main procedure calls on a chain of subprocedures to determine what outputs are appropriate in response to an input. These subprocedures are: "handler", "reply_handler", "read_ODB_handler", and "lookup_handler". Each one handles a more specific, possible, situation. Collectively they determine the values of "out", "multicast", and "sndmesg", which in turn determine exactly the outputs that will be sent during the handling of the current input message. By using the subprocedure decomposition technique of section 4.2.3.2, it will be possible to show that the existence of higher-level transactions in "pending_ops" will not affect the values of "out", "multicast", and "sndmesg".

An important part of this proof concerns concurrency control. Normally, an input will be processed by relating it to some transaction, making sure the input is at the same level as the transaction, and basing the output solely on the state of the transaction. However, if the object of the original invocation is locked because more than one update operation has been started for it, then the output of the catalog manager could depend on the entire set of pending transactions. In general, this set will have transactions at many levels, and the resulting outputs could compromise higher-level information. Two facts prevent this:

- The catalog manager need not service any "write-up" operations, since the scheme chosen for the SDB (see section 4.1.2.2) will have already upgraded previously in the message switch any "write-up" operation to the level of its object.
- Any catalog manager operation requiring concurrency control is an operation that must perform a "ModifyODB" operation on its object. This secondary operation will fail unless the object's level dominates the level of invocation. Given the first constraint above, the catalog manager is justified in aborting any such operation unless the level of the object is exactly the level of the invocation.

Therefore, a lock on an object which has more than one pending "update" operation is guaranteed to exist as a result of transactions at the object's level and no other. Even in the presence of this kind of concurrency control, the output at one level need not depend on whether higher-level transactions exist.

We have argued that the catalog manager's response to the current input message can be proved secure, if considered in isolation. Once the current message is handled, though, the "pending_ops" list is searched for other transactions that may become unblocked once the lock on the object of the input message is removed. From the facts about concurrency control presented in the last paragraph, we can conclude that the set of transactions potentially unblocked at this point is the same in both the actual history and the purged history (they are all the level of the input message). The function "find_waiting" chooses which potential transaction is to become unblocked. If this function is deterministic, and it makes its choice based only on the pending transactions at this and lower levels, then it will choose the same transaction in both the actual and purged histories. This transaction then becomes the new input to "handler", and the proof of security proceeds exactly as for the input message at the top of the main loop.

The function "find_waiting" can either be assumed to have the required property, or it can be more completely defined (either with an exit specification or a procedural body) and the program transformation technique can be applied to it as in section 4.2.3.2. Once its non-interference property has been established, the security of the entire catalog manager is proved.

4.3.4 Verifying the Kernel specification

The kernel was the most complex of the SDOS components that were designed to be secure. As can be seen from the kernel specifications in Appendix A, each of the kernel components was isolated as a procedure. Communication between components was modeled by subprocedure calls, as described in section 4.1. The kernel design did severely test our security policy and also the methodology chosen for verification. We are convinced that the design is "secure", in the sense that the aspects of the design that cause failure of restriction are few in number, are known, are desirable features, and are difficult or impossible to exploit. Some generalization of the policy of restrictiveness would be desirable for verifying that these aspects are indeed benign. The program transformation methodology did not prove to be adequate either. The scheme for proving subprocedures in Gypsy (section 4.2.3.2) could not be applied. It was said earlier in section 4.1 that instances of WNI processes may not be provable by the program transformation technique. We did encounter that problem in attempting proof of the kernel.

In the light of the above difficulties, we were unable to conduct the formal verification of the kernel specifications. The sections that follow will elaborate on the problems and provide an intuitive understanding of why the kernel design is secure.

4.3.4.1 Intuitive understanding of kernel security

The key points to note about the design of the kernel components:

- The SDB does not release information about entities at levels higher than the request. Write ups and other "up" operations will get their responses back at the level of the object, which cannot reach the client. If the object did not exist, the reply would be at the level of the object, but this cannot be relayed to the client because it would violate our security policy. Therefore no operation at level l , say, will base its actions on information at a level greater than l and respond at l . In other words, responses to clients will always be at the join of the level of the information that is used to act on the request and the level of the request itself. The locator cache and the process table are also protected by their managers behaving under the same rationale as the SDB.
- Since clients and managers are untrusted entities, in general, care must be taken to avoid security breaches through covert channels. The message switch always checks the validity of the level of the client's request with the SDB before proceeding with

the invocation. In case of the client misstating his level, the level of the request is rectified. Also to prevent managers from relaying information back to clients, single level managers who handle client requests would always be at the join level of the level of invoke and the level of the object. This way, the manager cannot communicate any information that is invisible to the client himself. For "up" operations, this is ensured by the ReadSDBEntry returning information at the level of the object.

- The only entity within the kernel that can cause a change in level of reply (from that of the invoke) is the SDB. The SDB behaves securely, as described earlier. Since the invocation comes from the message switch, the level of the invocation can be trusted. All components use the SDB to determine the level of the objects that they are dealing with. The level of the SDB's reply will be the level of further actions by these components. Hence, as discussed in the previous paragraph, the actions are secure.
- ModifySDBEntry can be successfully invoked only by the System Manager and the Authentication Manager. Since these are both trusted entities, the unintentional downgrading will not occur.

4.3.4.2 Reasons for not conducting formal verification

The reasons for our inability to conduct formal verification on the kernel design stemmed from weaknesses in our policy and from the choice of the methodology for verification. This section is devoted to discussing the details.

- The policy does not handle the situation in which the levels of some events are unclear. Specifically relating to the kernel, client processes are not necessarily trusted to stamp their levels correctly. Therefore, the level of the message reaching the message switch is unclear. The level can be determined only after the message switch interacts with the SDB. See section 4.2.4 for advances towards generalizing the theory to formalize the determination of such levels. The problem with the policy as it stands now can be highlighted by an example. If clients mislabelled their messages to the message switch, then a reasonable message switch design would call for resetting the label to the correct level before proceeding with the invocation. For instance, suppose that unclassified client A sends his invocation labeled as secret. The message switch would reset the level of the invocation to unclassified and proceed with the invocation. This, obviously, will violate our security policy. It must be noted that there is no security breach. The legitimate action shows up as a violation because the client's invocation was mislabelled.

One could attempt to circumvent this problem by forcing the message switch to disregard all messages that have been mislabelled, i.e., to take no action on behalf of the request. We would then need to prove that the message switch satisfies this mechanism in addition to restriction. But this would be introducing some denial of service that is really unnecessary.

- The security policy does not permit downgrading.

Downgrading is the phenomenon wherein an entity's level is reset to a lower level. This action might be needed to reduce the overhead of maintaining secret information. It is possible that some information, weather data, perhaps, that is a few hours or days old need no longer be secret. The ModifySDBEntry invocation serviced by the SDB would be a vehicle to achieve downgrading. But such an action would violate the policy.

- CreateSDB operation introduces non-determinism.

The CreateSDBEntry operation returns randomly generated UIDs, thus making the SDB non-deterministic. This makes theorem 2 inapplicable, and prevents the use of the program transformation technique of section 4.2.3.1. Thus, even though the SDB process is WNI and restrictive (if ModifySDBEntry is excluded), our methods are inadequate for demonstrating this in Gypsy.

- The scheme for handling Gypsy subprocedures (section 4.2.3.2) calls for these procedures to be SNI. The procedures that describe the kernel components are not SNI. Therefore, our only available scheme for proving the kernel secure is to use inline expansion (with some optimization). But, this would cause an explosion in the number of VC generation paths and an already time-consuming task would become infeasible.

4.3.5 Proof of processes involved in Authentication

There are a relatively large number of assumptions and assertions in this section. This is due to two reasons. First, the capabilities of Gypsy are somewhat limited and so much of the mandatory proof must be done external to the specifications. Secondly, much of what we need to show are not mandatory properties.

4.3.5.1 TIP specifications

I. State Transition of TIP

The following is a complete description of the state transition diagram for the TIP.

The possible states are READY, LOGINGIN, LOGINGOUT, and ACTIVE.

There are also two state variables. One, called LastLogin gets updated when there is a login and confirmed when receiving the login reply from the system. The other is Curlevel, which keeps track of the security level of the current user.

Each time a new request comes in from either the user or the system, there is a state transition (possibly just to itself). The variable fromuser is set to true iff the message was from the user. Incoming user messages are stored in ucall and incoming system messages are stored in incall.

There are 6 boolean functions which indicate the kind of message:
IsLogin, IsLogout, IsLoginReply, IsLogoutReply, LogoutAck and IsLoginOk.

Assume that:

1. IsLogin(ucall) iff ucall is a login request
2. IsLogout(ucall) iff ucall is a logout request
3. IsLoginReply(incall) iff incall is a reply to a login request and in addition it matches the last login request (i.e. incall.transactionnumber=lastlogin.transactionnumber and incall.level=lastlogin.level and incall.sender=Authentication Manager)
4. IsLogoutReply(incall) iff incall is a reply to a logout request
5. LogoutAck(ucall) iff ucall is a tip generated reply to a logout request
6. IsLoginOk(incall) iff incall has the reply field set to ok and not incall.level=ready

The state transitions are:

Initially, state=ready and curlevel=low

```

if prevstate=loggingout then
  if not fromuser and IsLogoutReply(incall) then
    state=ready
  else
    state=loggingout
elif prevstate=active then
  if fromuser and IsLogout(ucall) then
    state=loggingout
  else
    state=active
elif prevstate=ready then
  if fromuser and Islogin(ucall) then
    state=loggingin
  else
    state=ready
elif prevstate=loggingin then
  if not fromuser and Isloginreply(incall)
    if IsLoginOk(incall) then
      state=active
    else
      state=ready
  elif fromuser and Islogout(ucall) then

```

```

    state=loggingout
  else
    state=loggingin
  end

```

```

LastLogin ne prevLastlogin -> (state=loggingin and prevstate=ready
    and Lastlogin.level=Requestedlevel(ucall))

```

II. Level specification.

For each state transition, let *curlevel* be the current security level and let *prevcurlevel* be the previous current security level. Then:

```

curlevel=prevcurlevel  or

state=active and prevstate=loggingin and curlevel=level of the
    last incoming call from the system(which is the loginreply)  or

state=ready and prevstate=loggingout  and curlevel=readylevel
    (using the fact that  state=ready  implies  curlevel=readylevel)

```

So there are two times when a level gets changed. After a logout the *curlevel* becomes the *readylevel*. After a correct login the *curlevel* becomes the level of the *loginreply* message (which is the level at which the TIP was set).

III. TIP output spec

The variables *senttouser* and *senttosys* are set to true iff there was a message sent to the user or system (respectively) during the last state transition. Let *prevstate* be the state at the time the message was sent.

```

senttouser iff ( (there was a message incall from the system and
    (prevstate=active and incall.level=curlevel)
  or (prevstate=loggingout and Islogoutreply(incall))
  or (prevstate=loggingin and Isloginreply(incall)) )

sentosys  iff
  (there was a message ucall from the user and
    (prevstate=active)
  or (prevstate=loggingin and Islogout(ucall))
  or (prevstate=ready and (Islogout(ucall) or Islogin(ucall))) )
  or (prevstate=loggingout and IsLogout(ucall))
    and state=loggingout and IsLoginreply(incall))

```

The output to the user is solely a function of the state (including *lastlogin* and

curlevel), and the incoming message from the system. The following description is based on the above mentioned cases.

1. It is the transformation solely of incall.
2. It is the transformed logoutreply message.
3. It is the transformed loginreply message.

The output to the system is solely a function of the state, and the incoming message from the system.

1. If the state is active then first the message is transformed from just ucall, and then label field is set.
2. It is a logout request (from active, loggingout or loggingin).
3. It is an AuthenticateAs operation being sent to the authentication manager with the message field solely a function of the users call.
4. It is a delayed logout request (which was waiting for the loginack from the system)

IV. LastLogin specifications.

Each login request coming from the user in the ready state will generate a unique transaction number for the outgoing call. (To ensure the probability of a mismatched login reply is essentially zero.)

V. Output assertion to User:

Any messages sent to a user after an Ok login acknowledgement but before the next logout acknowledgement will be at or below the level of the login acknowledgement (and hence at or below the level of the corresponding login).

4.3.5.2 Authentication Specifications

I. The content of messages.

For login and logout there are two kinds of messages sent, messages to the kernel and replies to the TIP.

Messages sent to the TIP have only 1 added bit of information sent back in reply (error=true or error=false) and the original message field is blank.

There are two kinds of messages sent to the kernel, set or reset process bindings.

II. In checking for correct login it is asserted that:

1. The request has been checked to make sure it is from an appropriate TIP.
2. The login data is correctly checked to see if it is legal.

If either check fails then the CI information will not be changed and the reply will be 'error'.

III. Every input to the manager will cause exactly one output, which is at the same level as the request. Also, level l messages only change and read level l information from the data structures.

IV. Only the owner of a password (and possibly the system manager) can change the password entry. (It is probably impractical to encode all the users as different levels and so this integrity condition is checked directly rather than just using security labels.)

V. Timing of Password Invalid Response.

Login error responses will happen at a rate no faster than some constant. This will require an auxilliary proof whose validity depends on details of the hardware.

4.3.5.3 Message Switch Assumptions

- I. Preservation of fields (The sender field is set correctly)
- II. Correctness of originator (including the level field and the sender field)

4.3.5.4 External Assumptions

I. Correctness of Filter

It is assumed that the filter will be implemented correctly. Information from a session is only visible to a user who acquired access to that session. Information passed to the system is only achieved by the user who acquired access to that session. A release line request will cause a logout.

II. Password Assignment

Passwords for users above minimal security will be sufficiently complicated. The probability of guessing a correct password should be near zero. (See, for example, the DoD Password Management guidelines [DoD Password 85].)

III. User to TIP assertion: After a login and before a logout, all user messages will be at the level of the login. Precisely,

for all input sequences s from the user to the TIP
 (for all n (ISLOGIN($s(n)$)) $\rightarrow$ (for all $m > n$ and $m \leq \text{length}(s)$ not ISLOGOUT(m))
 $\rightarrow$ for all $m > n$ and $m \leq \text{length}(s)$ restrictive level(m) $\leq$ restrictive level($s(n)$)))

This condition is really too stringent. A user really does not need to logout after an illegal login. However, the actual condition needs to be stated in terms of both inputs and outputs. It is,

for any trace of inputs and outputs from a user,
 $(\text{ISLOGIN}(s(n)) \rightarrow (\text{for all } m > n \ m \leq \text{length}(s) \text{ not ISLOGOUT}(m) \text{ and not ISLOGIN-REPLY}(m).\text{failed}) \rightarrow \text{for all } m > n \text{ and } m \leq \text{length}(s) \text{ restrictive level}(m) \leq \text{restrictive level}(s(n))))$

Directly proving restriction from this sort of limitation is harder than just showing input-limited restriction. So the proof will proceed in several steps. First we will assume the user abides by the first, stronger condition in order to show input-limited restriction of the TIP; then we will use the hookup theorem; and finally we will show that the modified version does not effect either restriction or security of the entire system.

IV. The Filter and the line release acknowledgement

The user should not time the releases of the terminal line so as to convey high level information. The system does not provide much help in aiding the user as to how to make this decision. On the other hand, failing to abide by this requirement would at worst be a very slow channel.

4.3.5.5 Authentication Manager Proof

The proof of restriction for the Authentication Manager is as detailed below:

Each input at level l causes an output at level l . The output is purely a function of the input, with level l information stored in the password table and pending operations list, and in constants of the specification.

It is likely that this proof could have been done using the program transformation technique of section 4.2.3.1. Instead, we present here a proof by cases.

Suppose the input is at level l .

Cases:

An AuthenticateAs input will use the message and the password table to construct the output. In fact only a level l entry of the password table is checked. Since these entries can only be changed by level l requests, the result only depends on the level l history. The actual message sent is only a function of that input and level l information from the data structure. The addition of this request to the waiting list for receiving a reply from the kernel does not change any other message on the list. (Hence no lower level requests will detect the difference)

A Logout request is low and will cause a message to be sent at that low level.

A ChangePassword request is made at level l and causes an output at level l . And it only makes changes to the password table at level l . (In fact the only field that may

be changed is the one belonging to that user)

A reply from the kernel at level l will be matched with a previous l message if any, and then sent out. Only a level l message can be removed from the data structure.

All other requests will be just be answered with a failed reply message at their level, (with the same label l , and same restrictive level).

So, in any case it is always possible to add or delete higher level inputs in a history so as to satisfy the restriction property.

4.3.5.6 Proof of restriction for the TIP (including the Filter).

In the following, the term "restriction" will always refer to input-limited restriction and the term "userlow" will mean the lowest level possible for any user.

Recall that there are two kinds of user messages, those to obtain and release access to the terminal, and requests to the system. The TIP will also receive system messages.

The proof of restriction will be conditioned upon assumptions on the inputs.

Note that in our model, if there is a consecutive group of inputs (no intervening outputs or internal events) it is possible that the next output will happen after all of them. (If they happen close enough together then they will be buffered.) Hence we may assume that in adjusting any $\alpha^{\wedge}\beta^{\wedge}\gamma'$, that all of the changes appear in γ' .

Proof:

Let $\alpha^{\wedge}\beta^{\wedge}\gamma$ be any TIP history (where β is just inputs). Assume the user input history satisfies the assumption. Let $\beta^{\wedge}\gamma'$ be a set of modifications to $\beta^{\wedge}\gamma$ produced by adding or deleting some messages with levels not less than l . Let m be any such message. Assume that the user input history from $\alpha^{\wedge}\beta^{\wedge}\gamma'$ satisfies the input limitation assumption. We look at the first change and then proceed by induction to the others. At each step we will make appropriate modifications to γ' .

We will assume that $l \geq \text{userlow}$ (since there is no message that a user can send to the TIP below this, and any message from the system to the user at this level can be forwarded).

Case 1: Requests to obtain the terminal will, at least in part, be visible to other users. Namely, being denied service means someone else got the terminal. So all such requests will be treated as userlow, and hence m cannot be such a request.

Case 2: Release the terminal. As above, this is a userlow request.

Case 3: m is a message from a user who does not have access. Such a message is rejected by the filter (without reply), and so let γ'' be γ' . (This case is artificial in the sense that any reasonable user will not attempt to send a message until he actually has access to the terminal. It is covered here for formal completeness.)

Case 4: m is a message from a user who does have access.

No matter what the state of the TIP, it will either discard the message or forward a message to the system at the same level (see TIP specifications). If needed, add or delete this as an output. If this message causes a state change in the TIP, it may be necessary by induction to change later outputs. All user messages to the system during this login session are at this level (except for logouts which are considered userlow) and all system messages except logout acknowledged are at this level.

Subcase 1: m appears after a valid login and before a logout. (So by the assumption on inputs it is the level of that login)

Subsubcase a: m appears while the TIP is in the active state. Then by induction m will be transformed into an output to the system and stamped with level l .

Subsubcase b: m appears while the TIP is in login or logout state, then make no modifications. (The message gets thrown away.)

Subsubcase c: m appears while the TIP is in the ready state. (This is not a possible case; see state transition description in section 4.3.5.1.)

Subcase 2: m does not appear after a valid login and before the logout.

Subsubcase a: If it is a login and the TIP is in the ready state forward the request at the requested level. (logins are always at the requested level, so restriction is not violated)

Subsubcase b: It is not a login. By induction, we know the TIP is not in the active state. In all the remaining possibilities for which m has level l , (i.e. is not a low logout), the message just gets thrown away, so no adjustments need to be made.

(Outputs are solely a function of the message and which of the four states the tip is in, and of the two state variables, lastlogin and curlevel.)

System cases:

Case 5: Logout acknowledged will be at userlow.

Case 6: Login acknowledged is at the level of the login by specification. In this instance, add or delete an output to the user informing him of failure.

Case 7: Messages above the level of the user get rejected. No outputs need be changed.

Case 8: Messages at (or below) the level of the user which are not loginacks or logoutacks, will get tossed if the TIP is not in the active state. If the TIP is in the active state then adjust the outputs by adding or deleting the message to the user.

As mentioned above we may need to fix outputs inductively.

So in all cases we may construct a γ'' so that $\alpha \wedge \beta' \wedge \gamma''$ is a history with the only differences between γ' and γ'' being the high level outputs.

User Outputs

We assume that the output behavior of the user meets the input limitations stated above.

4.3.5.7 Implementation Considerations

4.3.5.7.1 Implementing the filter

Recall that a filter is a formal construct which represents how different users share the terminal. In some cases, it may be possible for the user can handle the role of the filter. Whether there needs to be external security or whether a user can be trusted to do this depends on the environment. It may or may not be easy to block access to the terminal. We can often assume that the user is trusted to release the line before physically leaving a room in which the terminal resides.

It is surely the case that we must assume that a user with multiple accounts at different levels will behave securely (i.e. will not do any writedowns).

4.3.5.7.2 Constructing a Protocol

There are a number of possible protocols that can be used by the processes involved in authentication: the TIP, Authentication Manager and the Kernel. We have chosen a simple protocol which does not explicitly check that the order of the login is maintained when the kernel executes the set and reset process bindings operations. If the authentication manager and the TIP are on the same machine then the current specifications should be sufficient. A more complicated protocol may be needed if these messages must pass between machines. Proofs for this latter type of protocol are harder because the the limitations between the process connections are no longer just input-limitations. A generalization of input-limitation to trace-limitation is possible, but the theory has not been fully worked out. There is also a future possibility of using the theory of generalized restriction (section 4.2.4.3) to provide a satisfactory foundation for this type of protocol.

4.3.5.8 Composability of the User and the TIP

The input assumption in the proof of input limited restriction of the TIP is assumed to hold for each user. In return, the TIP will screen messages that the user should not see. This will be a sufficient limitation on inputs to the user (outputs from the TIP), so that the user can act as though he were a restrictive entity. Hence we could apply the input-limited hookup theorem between any set of users and their corresponding TIPs.

4.3.6 Concluding Remarks

So far we have shown:

The Authentication Manager plus the 'base system' of Kernel and MLS managers is restrictive. We also have shown that the combination of the TIP and the filter is input-limited restrictive, so we can apply the input-limited hookup theorem to include this component as well. Therefore, the entire system is input-limited restrictive.

4.3.6.1 Limitations on users

As mentioned previously, it is not just restrictiveness which makes this system secure. It is the fact that the only high level rights, other than login confirmation, that a user has are those obtained by a login at that level or higher. Other than logout acknowledgements and login acknowledgements the only other messages that get sent to the user are when the TIP is in the active state. This can only happen if the login request comes back with a matching login okay from the authentication manager. And the login okay can only happen if there was correct password verification in response to this login request.

If the message switch does not preserve the order of messages between the TIP and the Authentication Manager or between the Authentication Manager and the kernel, then a logout issued while logging in may not effectively clear the CI rights of the TIP. If this will be a problem, it is possible to buffer a logout request until the login acknowledgement has returned from the system.

A NewPassword command must also be designed correctly, so that password entries are not incorrectly changed.

Of course, one of the biggest problems is not the correct security of the system, but rather the correct security of the users. Authentication will serve little function if passwords are not properly protected outside of the system.

4.3.6.1.1 User assertion

We now see what happens if we weaken the constraint on the user. Suppose we impose no constraint between an illegal reply message and the next login or logout from the user. After an illegal reply message from the TIP (uniquely determinable by the absence of an intervening legal login reply message) the TIP will be in the ready state. Any message which is not a login or a logout will be discarded. Deleting or adding these messages causes no other change in the history because they are ignored by the TIP.

Here is an informal argument that this does not affect security. Let us choose any history h of the system (under the new weaker restraint on the user). Now add or delete high level inputs to h as in the input-limited restriction theorem, calling the new

sequence h' . Now delete from h' any user inputs after an illegal login acknowledgement and before the next login or logout. Call this sequence h'' . Let g be a modification to the original history with the extra inputs removed (so as to conform to the tighter restriction on users). Now adjust h'' to form h''' by deleting any low level input after a logout and before a login which was also deleted in g . (An event which used to follow an illegal login may, in h'' , follow a logout and so may need to be deleted to agree with g .) Now g and h''' agree except possibly on high level inputs. They also satisfy the tighter restrictions on the user. So by the input-limited restriction theorem adjust h''' to form h'''' . Now modify h'''' to form h''''' by inserting the superfluous inputs back into h'''' . This final history will be what we must produce to show that we have met the conditions for the input-limited restriction theorem with the weaker input restriction.

4.3.6.2 Conclusion

This system is input-limited restrictive. However, there are two small trouble spots. One is that a user may inadvertently do writedowns when they acquire and release access to the terminal. As pointed out, this is at worst a very slow channel. Secondly, we have granted unknown users the right to receive illegal login replies. This happens at a sufficiently slow rate, and the passwords are chosen to be sufficiently random so that it is highly unlikely that this will cause a more serious problem.

Hence, the system is essentially secure.

Chapter 5

Final Report

5.1 Project Goals and Accomplishments

The objective of this project has been to investigate multilevel security issues as they relate to distributed operating system design. The required deliverables include a security policy, formal model, and formal top level specification for an A1 class secure distributed operating system, and documentation of the issues and possible solutions that have been discovered in the course of this project. In support of these objectives, we devoted some effort to producing a high-level design for a secure distributed operating system (SDOS).

Working entirely in the abstract is difficult and often unproductive. Therefore we chose to investigate multilevel security issues in the context of an existing, operational distributed operating system. The existing system that we chose is Cronus. Cronus is an object-oriented distributed operating system, that can operate across a heterogeneous set of networks and host operating systems. It has a set of features that provide good support for the writing of distributed applications, and it has an internal architecture that allows those features to be implemented reliably and efficiently. By basing our SDOS design on Cronus, and preserving as much as possible of its feature set and internal architecture, we felt assured that the resulting design would be usable and implementable as well as secure.

A distributed operating system is built on top of a set of single hosts, which are connected by a network. There has been a great deal of research in the areas of single host security and network security. It is our conclusion that, while distributed system security is related to these other two areas, and in fact depends on them for its success, it is a distinct area with a set of problems unique to it. Exploring the distinctions between the three areas has helped increase our understanding of distributed system security.

Our research has uncovered a number of problems unique to distributed system security, and identified some possible solutions to them. They are mentioned briefly here, without any supporting arguments, in order to define the scope of the discussion. They are treated in more detail in the subsections that follow.

A distributed system security policy must be defined in terms of message passing between active entities, rather than the traditional (Bell and LaPadula) read and write operations of an active entity (process) on a passive entity (file). The concept of a distributed TCB, running in the higher layers (above the communications and host operating system layers), must be supported by the security services provided by the communications and host operating system layers; that is, the distributed TCB must be implementable within the security restrictions imposed by those lower layers. The distributed nature of the TCB, particularly the fact that parts of the TCB can drop out of and later rejoin the system, as hosts go down and come up, presents some security problems, especially in the area of object replication. The access class range (system low to system high) can, in general, be different for each host and each inter-host path; further, within each host it becomes useful to talk separately about the access class ranges for active entities (processes), passive entities (files), and messages sent to, and received by, the host. This has implications for the multilevel security policy. There are a number of covert channels that are brought into existence by the distributed operating system's attempts to make its distributed nature transparent to application programs and users, and its attempts to operate efficiently, minimizing delays due to inter-host communication. The desirable objective of having the SDOS operate across a heterogeneous set of networks and host operating systems presents some security problems, especially when the various networks and hosts vary in the degree of assurance of their security features.

In the area of formal specification and verification several significant results were achieved. Our basic goal was to write the Formal Top-Level Specification of the design of SDOS, and to prove formally that it satisfied multi-level security. To a large extent, this goal was met.

We based our definition of multi-level security on an emerging theory of information flow security being developed at ORA [Ulysses 87]. This theory defines information flow in terms of the deductions that can be made about unseen (higher security level) events in a system's history. A basic result of that theory is the discovery of a *composable* security property: two subsystems having the property can be hooked together to form a larger system also having the property.

Our work in this project has extended the theory of information flow security in two ways:

1. We have proved theorems that enable one to break the primary security property into simpler sub-properties;
2. We have developed a technique for demonstrating the simpler sub-properties using the Gypsy Verification Environment.

5.1.1 Distinction Between Network and DOS Security

In our view, a distributed system is different from a network, and thus distributed system security is different from network security. The difference is that a distributed system is built on top of a network. It is implemented in and above the higher layers of the OSI model. A distributed operating system provides support for the writing of distributed applications. It attempts to identify those functions that are common to most distributed applications, and implements them in the operating system, relieving application programmers of the task of implementing them in each application. Some of the services provided by Cronus (the DOS on which this project was focused) would be identified with the three highest layers of OSI (session, presentation, and application); others would be located in still higher layers. The DOS attempts to hide the distributed nature of the environment from application programs. For example, it provides the same interface for accessing local and remote data objects. A security policy for an SDOS must be stated in terms of the subjects, objects, and operations implemented by the higher layers in which the SDOS exists, and not in terms of the entities of lower layers.

It is useful to put this discussion into the context of the terminology and concepts in the TNI [NCSC TNI 87]. Section I.3.2 of the TNI describes two network views: the interconnected accredited AIS view and the single trusted system view. An SDOS is more closely related to the latter than to the former. However the collection of underlying hosts and networks upon which the SDOS is built may appear to conform more closely to the interconnected accredited AIS view. The nature of this hybrid view will be made clearer in the sections that follow.

The discussions of connection-oriented abstraction and subjects and objects in section I.3.2.2 are related to entities of layers lower than SDOS. (In the SDOS design, connections are owned by TCB partitions and are used for communication with their peers on other hosts.) Section I.4.3 mentions four types of security policies that may be supported by a network component: mandatory, discretionary, supportive, and application. It gives, as an example of an application security policy, a policy supported by a DBMS that is distinct from that supported by the underlying system. It goes on to say that application level policies will not be considered further in the TNI. We consider the relationship between the SDOS security policy and that of the underlying network to be similar to the relationship between the DBMS security policy and that of the underlying system: from the point of view of the network, the SDOS security policy is an application policy that is distinct from that supported by the underlying network.

Further, in some cases the SDOS security policy is in conflict with that of the underlying network (as outlined in the TNI), and therefore the SDOS layers must be privileged (i.e., they must be part of the TCB), so that they can enforce the SDOS security policy and not be hampered by the network security policy. One example of such a conflict is the set of restrictions in B.4.1 of the TNI, two of which are that a subject is confined to a single network component and that it may directly access only objects within its own component. The SDOS deliberately tries to mask all distinctions between compo-

nents (hosts). At the level of abstraction of the SDOS, a subject is logged in to, and authenticated for, the entire SDOS, and may access all objects in the SDOS, subject only to the restrictions of the SDOS security policy. This policy is one which prevents information flow from high secrecy to low secrecy entities and from low integrity to high integrity entities, where entities are the subjects and objects implemented by the SDOS. This discussion is continued in the section entitled Distributed TCB, below.

5.1.2 Contrasts Between Single-host and DOS Security

There are many differences between single-host and DOS security. This section will discuss only two of them. The two were chosen because the most effective way of explaining these two aspects of DOS security is contrasting them with their single-host counterparts. The two are the way in which TCB boundaries are defined and the way in which object references are implemented.

5.1.2.1 TCB Boundaries

In a traditional single-host secure operating system, the TCB begins at the top of some layer, and extends down through all intervening layers to (or into) the underlying firmware and hardware. Thus we think of the traditional TCB as having only an upper boundary. In contrast, the partitions of a distributed TCB each have both an upper and a lower boundary. In the general case, the partitions must exchange messages via some untrusted communications medium, and they must use some technique (most likely cryptographic) to protect the secrecy and integrity of the data that they transmit over the untrusted medium. Thus, by default, the lower boundary of each TCB partition is located at the beginning of the untrusted medium. This default TCB boundary location could be at a number of different places, depending on implementation details. For example, it could be at the beginning of an unprotected wire or an antenna. It could be at the host side of a hardware device driving the wire or antenna. If one or more of the lower layers of communication software run in a front end processor rather than in the host, the lower TCB boundary could be at the interface between the host and the front end (provided that no security-related functions are implemented in the lower layers).

It seems clear that the lower boundary of the TCB should not be allowed to be located by default, but rather that the best possible location should be chosen for it, taking into account a number of factors. These factors include the objective of minimizing TCB size, thereby maximizing assurance and minimizing implementation and verification cost; and choosing the architecturally best layer in which to locate each security feature. Finally, if the TCB boundary location is chosen such that some lower layers are outside the TCB but still within the host, some implementation technique (dependent on the host operating system) must be used that will protect the TCB from these lower layers even though the TCB calls them.

5.1.2.2 Object References

In a single-host system, object references (for example, file reads and writes) are typically handled by the TCB because the I/O system and file system are in the TCB. (Even if the full I/O and file systems are not in the TCB, some primitive object-management system, on which fully-functional I/O and file systems can be built, is in the TCB.) In a distributed system, on the other hand, object references are typically handled by untrusted processes on the hosts involved, which exchange messages with each other via the TCB. (In Cronus and SDOS, the process that issues the request on behalf of a user is called a client process, while the process that responds to the request, on the remote host where the object is located, is called a manager process.) The requirement for a two-way exchange of messages between the untrusted processes can sometimes cause security problems, because of mismatches between the access classes of the two processes and the object being referenced.

Consider, for example, the case of a high-secrecy process attempting to read a low-secrecy object. This is, on the face of it, a legal operation. In the single-host case, the read is implemented entirely within the TCB. In the distributed case, the read could involve the sending of a message from a client process on one host to a manager process on another host. Consider the constraints on the access class of the manager process. If it is of lower secrecy than the client, then the sending of the read request is in violation of the SDOS security policy (which forbids the flow of information from a high secrecy entity to a low secrecy entity). If the manager process is of higher secrecy than the client process, then the response to the read request would be in violation of the security policy. Clearly if there is to be a two-way exchange of messages between the single-level manager and client processes, they must have identical secrecy classes. But now consider the relationship between the secrecy classes of the manager process and the object(s) that it manages. If the manager has a higher secrecy class than the object, then it may read but not write the object. On the other hand if the manager has a lower secrecy class than the object, it may write but not read the object. Clearly if the manager is to be able to both read and write the object, the manager and object must have identical secrecy classes. (The analogous argument for integrity classes also applies, but it is omitted here for the sake of readability.) It seems that clients can communicate only with managers having access classes identical to their own, and that managers can read and write only those objects having access classes identical to their own. Thus, in the absence of some solution to this problem, clients can access only those objects that have access classes identical to their own. This is inconvenient.

This problem arises from the replacement of object references via the TCB, in the single host case, by object references via unprivileged manager processes, in the distributed case. We see two possible solutions to this problem: move the manager process into the TCB, or provide a set of manager processes (potentially multiple instantiations of the same executable code) at and above the access class of the object. The former has the drawback that an object manager is inherently an application program, and it is an objective of the SDOS design (and a feature of Cronus) to allow users (or at

least using organizations) to define new object types and implement managers for them. The latter has the drawback that it leads to an unmanageably-large number of manager processes: one for each of the possible client access classes. Our design allows a using organization to choose either of these solutions, according to their own requirements and resources. We neither require nor forbid the use of multilevel-secure (MLS) object managers to support multilevel object types. An organization that has the expertise and resources to implement MLS object managers may do so. The alternative of using single level managers, one for each client access class, is also available. The problem of host operating system limits on the number of active processes is addressed by providing for the dynamic activation of manager processes in response to requests from clients. The resulting performance problems are addressed by providing a least-recently-used manager deactivation algorithm similar to traditional page replacement algorithms, and by using various ad-hoc techniques to speed up the activation of manager processes.

5.1.3 Distributed TCB

We have identified a number of problems unique to distributed system security, and some possible solutions to them. We have chosen the term "Distributed TCB" (DTCB) to refer, collectively, to those solutions. The term was chosen deliberately to emphasize the difference between the DTCB and the Network TCB (NTCB) described in the TNI. The DTCB runs in higher layers than the NTCB. Conceptually, the DTCB could be implemented either on top of an NTCB, or on top of a collection of interconnected accredited AIS. (In practice, the implementation of the latter would be the more difficult.) The distributed operating system implements a number of abstract entities (subjects and objects); the DTCB enforces a mandatory security policy that controls the flow of information between those abstract entities.

Given that a partition of the DTCB resides in each host, it is useful to consider the meaning of the access class range of each host. If a host is assigned system-high and system-low access class limits, what do they mean? Do they delimit the range within which that host's DTCB partition is trusted? Or are they intended to place limits on the range within which unprivileged processes may run, or within which objects may be created? We will use an example to motivate the answer to this question.

Consider a very high speed computer that is dedicated to making weather predictions and continually updating a database containing world-wide weather information. Although accurate weather information could be of some value to an opponent, it is inherently of low secrecy. Further, it must be classified at least as low as the lowest clearance of any of its legitimate users. So in this example we will assign the weather information an access class containing a level of confidential, and an empty category set. The intention is that only the confidential weather information will reside on this host, but that clients of higher secrecy classes, on other hosts, will be able to request and receive weather information.

It is not possible to state this policy using a single access class range for each host.

In fact, we need three ranges for each host, plus one for the system as a whole. System-high and system-low will refer to the SDOS as a whole, and will limit the range of information that can exist in the system, and also limit the values that the per-host ranges may have. On each host, there will be message-high, message-low, process-high, process-low, object-high, and object-low. Message-high is an upper limit on the messages that may be sent to the host, while message-low is a lower limit on the labels that may be put on messages leaving the host. Process-high and process-low limit the range of unprivileged processes that may run on the host, while object-high and object-low limit the range of objects that may be created on the host.

In the case of the weather system, both object-high and object-low would be set to confidential. If the weather database has a MLS manager, then both process-high and process-low would also be confidential; however if there are single-level managers for the database, process-high would be set to the highest secrecy level from which client requests will be accepted. Message-high would be set to system-high of the SDOS, or to some lower value, if dictated by weak physical security at the weather host, or by "Yellow Book" [DoD Guidance 85] restrictions.¹ Message-low would be set to the lower of the two values of process-low and object-low. (Message-high and message-low actually correspond quite closely, in their effect, to the system-high and system-low parameters of a single-host secure system.)

The effect of having three ranges per host is to allow the placing of separate constraints on unprivileged code and the DTCB partition in each host. The DTCB partition is trusted to process information of higher secrecy than any unprivileged program on the host. This allows the expression of complex policies, taking into consideration the different physical security and need-to-know characteristics of each host in a distributed system.

5.1.4 SDOS Covert Channels

Any system will have covert channels in it as a side effect of its implementation. There are, however, a number of covert channels that occur in SDOS only as a result of its distributed nature. These channels, or similar ones, would probably be found in any distributed system. There are two broad classes of channels that are of interest here. The first class consists, in general, of the ability of a low secrecy process to detect the existence or recent use of an object, coupled with the ability of a high secrecy process to modify those pieces of information. The second class consists of ways in which an untrusted process could signal to a confederate who is listening to an insecure medium within the network underlying the SDOS. These covert channels, like most others, can be fully understood only in the context of a detailed description of the design in which they occur, and such a description is given elsewhere in this document. The nature of these channels will be briefly sketched here.

¹Note that this prevents very-high-secrecy clients from requesting weather information. This is not an unreasonable restriction, if the location for which weather is being requested could, if revealed to an opponent through weak physical security at the weather host, compromise a critical mission.

When a client invokes an operation on an object, the system must locate the object before carrying out the operation. This involves a broadcast to all hosts, requesting a response from the one on which the object resides. This causes a real time delay for the requesting client, and possibly a throughput problem for the system as a whole, if done with unnecessary frequency. To solve this performance problem each host maintains a cache containing the locations of recently-used objects. This cache introduces two covert timing channels. (It is suspected that under ideal conditions the existence of the cache would allow the operation of sequencing channels.)

A high secrecy client can read a low secrecy object, causing its location to be placed in the cache; then a low secrecy client on the same host can invoke an operation on the object and detect, by measuring the time delay, whether or not the object's location was in the cache. This allows a covert communications protocol to be implemented, using a previously agreed upon group of objects to transmit the 1's and 0's of a message.

A high secrecy client can create and delete high secrecy objects at will. If the write-up operation is allowed by the SDOS security policy, a low secrecy client could detect the existence or nonexistence of a high secrecy object by repeatedly attempting to write it. An existing object would have its location cached after the first attempt, resulting in lower time delays for subsequent attempts. A nonexistent object would never have a cache entry and would always cause a long time delay before the operation completes (even when success or failure of the operation is not reported to the low secrecy client).

Various solutions suggest themselves: disallow all write-up operations; introduce random delays into the completion of selected operations; eliminate the cache completely; classify cache entries and allow only those processes who are cleared to read them to benefit from their contents. These solutions all have obvious functionality and performance disadvantages.

The second class of channels involves the ability of untrusted software to signal by modulating information visible on an untrusted medium. A distributed application, having a global view of its goals, and possibly some foresight into its future behavior, could make good use of the ability to influence the behavior of lower layers in ways that would globally optimize the use of scarce resources such as communications bandwidth. (Lower layers can, at best, make local optimizations based on past history.) However, the ability of a higher layer to influence the behavior of a lower layer by passing detailed control information across the interface has the drawback that it allows untrusted software to deliberately modulate some of the information that is necessarily clearly visible on untrusted media—message destinations, routings, lengths—the sort of information that traffic flow analysis looks at. Current layered protocols provide some ways in which higher layers can direct the behavior of lower layers to achieve optimization. Current work in DOS research suggests that layer interface enhancements allowing the passing of even more control information to lower layers would be useful. However the closing of these covert channels would require that downward information flow be shut off by providing even less ability to pass control information than exists in current protocols. This conflict is unresolved, and it is one of the topics suggested below for future research.

Solutions to these covert channel problems all involve tradeoffs of functionality and performance on the one hand against security on the other hand. The parameters involved in the tradeoff (the value to application programmers of a particular item of functionality, the performance implications of any design change, and the bandwidth of a particular covert channel) can only be estimated during early design stages. The choice of solutions should be put off until the prototype implementation stage.

5.1.5 Problems Arising from Heterogeneity

5.1.5.1 Heterogeneous Networks

It is a design objective of SDOS that its hosts be able to communicate with each other over heterogeneous networks, rather than being restricted to one particular set of network hardware and software technologies. The main benefit of the use of heterogeneous networks is to allow the use of existing networks. Many existing networks are not secure. Thus the objective of network heterogeneity becomes the objective of being able to maintain security within SDOS while using insecure networks for communication between SDOS hosts. Encryption can protect higher layer data, but some lower layer information must travel the insecure networks in the clear, for the simple reason that header information of the lower three layers of the OSI model (or the equivalent thereof) must be interpreted in the course of message processing within each insecure node of an insecure network (for example, ARPANet IMPs). The information handled by those layers is subject to traffic flow analysis, and also provides a potential covert channel, as outlined in an earlier section. These problems are addressed, to some extent, by the SDOS design, but further research would be beneficial.

5.1.5.2 Heterogeneous Hosts

It is a design objective of SDOS that the host-resident software be portable to various processors and operating systems. The benefits of this heterogeneity of hosts are the ability to provide varying amounts of processing power to meet the needs of various applications, and to take advantage of new processors and operating systems as they become available from manufacturers. (Naturally, the hosts on which SDOS is built must have hardware and operating system features capable of supporting multilevel security.)

The problems arising from the heterogeneity objective fall into two areas: implementation costs and architectural questions. There are two kinds of implementation costs: the cost of actually porting SDOS to a new host, and the cost of designing an interface that allows it to be ported easily. It is our feeling that, while the former cost should be borne by the various using organizations that have a requirement to run SDOS on particular hosts, the latter cost (portable design) should be a part of the initial development of SDOS (in other words, we believe that portability to a heterogeneous set of hosts is an important requirement).

The architectural questions that are raised by heterogeneity could be viewed as merely higher level portability issues, although they do have a more fundamental impact on SDOS functionality. These questions have the common property that they involve decisions between using security features in the multilevel secure host operating system or ignoring those features and implementing analogous, but slightly different, features in the higher SDOS layers. The economic advantage to using existing features is obvious. The disadvantage is that various parts of the SDOS security feature set become constrained by the details of the host operating system security features. One example is the size of access class labels: the number of different levels and categories that can be expressed in labels. These difficult architecture/cost tradeoffs must be made, even if SDOS is designed to run on only one type of host. The requirement for heterogeneous hosts makes these tradeoffs even more difficult.

The cost tradeoff problems discussed in this section are far from fundamental topics for computer security research, but they are important nonetheless. Users of secure systems have as their fundamental objective the doing of the job that is their organization's reason for existence. Security is merely one of the constraints within which they must operate; cost is another constraint. Design decisions involving cost must be given serious attention if a secure systems project is to succeed.

5.1.6 Problems Related to Object Replication

Data replication, or the maintenance of several copies of data on different hosts, is the primary technique for achieving data availability in the event of host or communication failures. Data replication is achieved by coordinating reads and writes across all copies of the data. Much research has been directed toward developing algorithms for supporting data replication. The algorithms vary with respect to the degree of availability, consistency and performance that they offer. The consistency, availability and performance requirements of distributed, highly available applications also vary. The replication scheme that satisfies best an application's data requirements is dependent on the application. As a result, replication mechanisms are commonly placed in the resource management software component that understands the semantics of the data. In object-oriented systems such as SDOS, these mechanisms are placed in object managers, and the object is the granularity at which replication is supported. If replication mechanisms were placed in the Object Database instead of object managers then it would be impossible to tailor the management of replicated objects in an application-specific way.

In addition to the data stored in an object, there is a significant amount of administrative information that needs to be associated with an object. For example, all objects in SDOS have access control lists and security labels. Additionally, the set of hosts where copies of a replicated object reside needs to be maintained. Objects that are replicated must have this administrative and security-related information replicated as well, both to avoid bottlenecks and to provide high availability.

Security labels are stored separately from objects in the security database. Access

control lists are maintained with objects in the object database. The design decisions that led to this placement reflected the considerations of the desired functionality and criticality of the information. We have since come to understand that there are additional trade-offs to consider when the objects are replicated.

When the administrative information is maintained with the object by the object's manager (in the case of SDOS, the access control lists), the application developer may be given control over the policy that governs how copies of the information are maintained. In the case of access control lists, a variety of different policies are conceivable. For example, strict consistency would ensure that changes to an ACL are propagated to all copies atomically. However, this would prevent an administrators ability to remove a client from an access control list when some copies are unavailable. Regardless of the particular policy adopted, placing the information with the object provides the maximum amount of flexibility possible for setting the replication maintenance policy.

Security labels cannot be maintained by object managers because they are critical to the enforcement of the mandatory security policy and they are used by other kernel components. Placing them in the security database forces the security database to support their replication. This has several implications:

- The security database (i.e., the kernel) must implement algorithms to maintain consistency between the copies of an object's label. While it is necessary to implement a single policy for maintaining security label replicates, placing these algorithms in the kernel complicates it. It also requires the security database to maintain a list of where all copies of an object (or more specifically, its labels) reside.
- The policy setting the consistency/availability of objects may differ from the policy determining the consistency/availability of their labels.
- The replication of objects must be coordinated with the replication of their labels. As a result, both the object database and security database must support operations to replicate and dereplicate an object and its label, respectively. Furthermore, only the object database may invoke the security database replicate and dereplicate operations; otherwise, label and object copies could be created or removed in an uncoordinated fashion.

The policy on the replication of security labels is loose consistency: updates to security labels are propagated as permitted by the connectivity of hosts. Since only the System Manager may update security labels, we believe this policy is appropriate despite connectivity problems. This policy also simplifies the replication management performed by the Security Database.

Section 5.2.1.2 discusses the complications security features introduce for replication management based on our experience with the formal specification of the Catalog Manager.

5.2 Tasks and Lessons Learned

This section summarizes the work that has been done on the SDOS project, and discusses some results in more detail than was given in the previous section. The three subsections, Policy, Design, and Formal Methods, summarize the results that are presented in much more detail in Chapters 2, 3, and 4 of this report.

5.2.1 SDOS Security Policy

The SDOS security policy was formulated in response to perceived threats to security. The resulting policy rules can be divided roughly into three groups:

- A discretionary policy, designed to control the use of SDOS' abstract operations on the basis of client identities;
- A mandatory policy, controlling the flow of information on the basis of DoD security levels;
- A configuration policy, defining the system's security "configuration" in terms of a set of "policy parameters", and controlling the modification of those parameters, both by system users and by changes to the network connectivity.

The mandatory policy, in turn, is composed of two parts:

1. A policy on message passing. Each system component has a set of levels, called its *label*, which records the levels of data the component is authorized to handle. A component may only send a message with a level from its label, and it may only receive a message if the message's level is in its label or can be upgraded to be.

The system components are divided into two groups: those that are certified to handle multi-level data (MLS entities), and those that are not. Those that are not have singleton labels.

2. A composable policy controlling information flow through the system's MLS entities.

Several aspects of this policy are noteworthy. First, the mandatory and discretionary policies are cleanly separated. The discretionary policy is stated in terms of abstract operations of the object model that SDOS supports. The mandatory policy refers to the message passing operations which are used to implement the object model.

Second, it is a global policy, giving requirements for the entire system rather than for individual hosts.

In the following sections, we discuss some ways in which policy concerns in SDOS differ from those in a centralized system.

5.2.1.1 Read-down and Write-up

In a distributed system, a "read" operation will often not be considered a fundamental operation. A "read" may be composed of a pair of messages: a request to read, possibly followed by a response. If reading data from a lower level entity is to be considered secure, as it is in Bell-LaPadula, it is because the request to read send-message causes no harm. However, the fact that a request to read has been sent is in general as sensitive as the reader itself, and in general it will downgrade information.

There are two approaches to this problem. One may demand assurance from the message's sender that the request is overclassified, and can securely be sent at the lower level. On the other hand, one may demand assurance from the message's receiver that the information in a high-level request to read will not be used for any purpose other than initiating the read operation itself. Either approach can be developed. For SDOS, we have used only the former approach: a request to read can be downgraded in some cases if a human being decides that it is secure to do so.

In the SDOS design, we have taken the approach that the existence of an entity will in general be as sensitive as the entity itself. This was done to allow clients to delete entities at their own level. However, the approach has a drawback: for a "write-up" operation, in which the client's level has to be dominated by the object's, the location of the object cannot be found if the search is carried out at the client's level.

We have singled out such "write-up" operations. In the "write-up" mode, a client chooses to upgrade an operation so that it can be carried out completely at the object's level. Of course, the client does not know what level that is, and gives up any hope of seeing a meaningful acknowledgement. However, once the invocation is upgraded, the system can find the object's level and can carry out the operation. However, an acknowledgement indicating either that the invocation arrived at the manager or that the write-up cannot be returned to the client is insecure, because it would convey to the client that the higher level object exists. Since objects' existence can be used as a covert channel, no acknowledgement of any kind can be received by the invoking client. As a result, it is not possible to build a reliable write-up operation.

5.2.1.2 Object Replication

Section 5.1.6 considered the impact of object replication on the management of security labels. In this section we consider the impact of security on the management of replicated objects. Our formal specification of the design of the Catalog Manager, which maintains replicated directories, provided an opportunity to experiment with a particular replication management (i.e., concurrency control) mechanism. We first consider this experience, and then generalize these results to more general replication management mechanisms.

The catalog manager locks out all but one concurrent update to a replicated directory. A significant fact emerged from the verification exercise: concurrency control as used

in the catalog manager, will not interfere with multi-level security. The following two points explain this:

- When a client invokes an operation on a higher-level object, the SDOS message switch does not try to guarantee that the client can use manager services at the its own level. Instead, the client must choose to let the invocation be upgraded to the object's level, and surrenders any expectation of an acknowledgement from the object manager. Because such invocations either fail or are upgraded to their object's level, the catalog manager is guaranteed to receive only invocations at levels that dominate the level of their object.
- The only invocations that can succeed at levels strictly greater than their object's level are requests to read ("Lookup") a directory. In the catalog manager as specified, both read and modify operations are atomic when applied to a single directory replica. Therefore, a read (Lookup) will never be blocked because a distributed update is in progress.

The concurrency control mechanisms used by the Catalog Manager, then, only block operations which write directories at the same level as the concurrent invocations. Since the clients of these requests are blocked by the activity of clients at the same level, the concurrency control mechanism does not act as a covert channel.

In general, since accesses to an object may come from clients at many levels, a concurrency control mechanism that prevents access of one client because of accesses of other clients at higher levels will be insecure. The specific problem that arises is that clients reading down to objects may cause lower level (writing) clients to be blocked, and this execution delay is a covert channel. For example, read/write locking is insecure, because higher level clients can lock an object for read and cause lower level writing clients to be blocked. In contrast, voting algorithms do not block read-downs, and therefore are not insecure.

5.2.1.3 Restriction: Hook-up Security

The part of the mandatory policy that controls message passing eliminates direct downgrading of data. However, it is the other part, the policy for each multi-level secure entity, that prevents information compromise via covert channels. That policy must guarantee that the levels an MLS entity assigns to each message are not underestimates of the sensitivity of the message's content. We describe that policy now.

We have used the multi-level security policy of McCullough [McCullough 87]. That policy defines security in terms of information flow. Information flow is defined in terms of the deducibility of facts about the history of inputs received by a component. A system component is secure if it does not allow information to flow from high security levels to lower ones.

The McCullough policy goes beyond this, however. It has the additional property of *composability*, two MLS components, when hooked together, form a larger component that is also MLS. A component with this property is sometimes called "hookup secure", but in this report, we have called these components *restrictive*. (Other properties exist that limit deducibility and are also *composable* in the above sense.)

The SDOS policy requires that the collection of all MLS entities be restrictive. Since restriction is a composable property, it is sufficient to verify that each MLS component is restrictive. The fact that security verification can be decomposed in this fashion is a tremendous advantage when trying to verify security for a distributed system such as SDOS.

The fact that MLS components must be restrictive is also an advantage when a secure system is to be extended. Extensions may include either new hardware or new software. In SDOS, extensibility means adding new object managers to the system to define new classes of objects and new abstract operations on those objects. If a new component is added to SDOS, and if it is verified to be restrictive with the same degree of assurance as the original system, then adding the component will create a new system that is also restrictive. The information flow security of the new system can be guaranteed with the same degree of assurance, and without a re-verification.

Our work on SDOS is almost certainly the first attempt to verify the property of restriction for a secure operating system.

5.2.1.4 Configuration Policy

The need for a configuration policy follows naturally when considering security in a distributed system context. The system's security is configured by a set of values called here *policy parameters*. These parameters include the security labels of system components, discretionary access control lists, and the results of specific choices such as whether audit records are kept, and whether the system can be extended with new trusted software. The system's security configuration may change with time. The configuration policy constrains who may cause these changes, and what consistency is required between security configurations on different SDOS hosts.

5.2.2 Design

5.2.2.1 Overview of Design

SDOS, like Cronus, is an object-oriented system. Objects are instances of abstract data types. The definition of a type includes the set of operations that are possible for objects of that type. There is a hierarchy of types: types inherit operations from their parents. Clients (processes acting on behalf of users) access objects by invoking operations on them. The invocation of an operation is the only way to access an object. Operations are implemented by object managers. A manager hides the internal representation of an

object from a client, and provides a precisely defined high level interface to the object. All resources in the system are represented by objects, and all operations are carried out as described above.

The SDOS TCB consists of the kernel, and a set of trusted managers that provide system services. The trusted managers operate according to the object-oriented abstraction described above. Since the kernel is the entity that implements the object-oriented abstraction, that abstraction is not available for use within the kernel.

The kernel consists of the message switch, the locator, the process manager (so-named for historical reasons—it is an internal part of the kernel, and not an object manager), the security database, the object database, and the process table.

The trusted managers include the file manager, catalog manager, authentication manager, and trusted interface process.

The function of each of these TCB components is briefly described below. More detailed descriptions may be found elsewhere in this report.

- **Message Switch:** Routes messages between entities, both locally and remotely. Enforces the mandatory security policy governing the passing of messages between entities. Communicates with its peer message switches on other hosts, and cooperates with them in the passing of messages and the enforcement of the security policy.
- **Locator:** Locates objects that do not reside on the local host. Provides this service only to the local message switch. Maintains a cache containing the locations (remote host identifiers) of recently used remote objects. Remote objects are located by broadcasting a request for the object to all hosts. If no positive response is received after a suitable interval, failure is reported to the message switch.
- **Process Manager:** Creates and destroys processes, and maintains (sets and shows) process bindings. Process bindings is the term for the set of information that includes a user's identity, and mandatory and discretionary access control attributes.
- **Security Database:** The collection of data needed for the enforcement of the mandatory security policy. This information includes, for each entity on a host: an access class label; a switch indicating single-class or multilevel-secure; for a replicated object, the number of replicas in the system; and for an object type, information about its manager, including: whether local managers exist, whether they are active, and the location of their executable code.
- **Object Database:** Provides storage for all objects that reside on the local host. Used by object managers.
- **Process Table:** Contains information about all active processes on the local host, including the process bindings. Maintained by the Process Manager. Consulted also by the Message Switch, when making mandatory access control decisions.

- **File Manager:** A multilevel secure manager, allowing the write-up and read-down operations. Also implements create, delete, open, and close operations.
- **Catalog Manager:** Provides an abstract space of symbolic names for objects. Translates from an object's symbolic name into its UID. (The UID is used to reference an object when invoking an operation.)
- **Authentication Manager:** Implements login and logout requests from interactive users. Sets appropriate process bindings for users logging in.
- **Trusted Interface Process:** Implements a trusted path between the system and an interactive user at a terminal. Maintains the state of the terminal, with respect to whether or not a user is logged in. Relays login requests to the authentication manager. Relays the requests of a logged-in user to other parts of the system. Could be called the Trusted Terminal Manager.

5.2.2.2 Enforcing Security

5.2.2.2.1 Location of Security Mechanisms One of the fundamental decisions in the design of a secure system is the choice of locations for the implementation of security mechanisms. In the case of the mandatory controls in SDOS, the choice was rather clear. The mandatory security policy is based on message passing; the message switch is the entity which is responsible for the passing of messages; therefore the message switch is the natural place to implement the mandatory controls.

For discretionary controls the choice was more difficult. The SDOS discretionary control scheme, which is based on that of Cronus, allows, in general, for a different set of discretionary control rules for each object type. The SDOS discretionary control scheme is summarized below, and discussed in great detail in Section 3.6.4. Briefly, an ACL consists of a list of entries, each of which consists of some client identification information (details omitted here), and a list of the operations which the bearer of that identification is permitted to invoke on the object. The set of legal operations is different for each object type. Therefore the set of operations that can appear in an ACL entry is different for each object type, and the code that searches and interprets an ACL must be different for each object type. This argues for placing ACL interpretation in the manager of each object type. However, managers are assumed to have low assurance, being writable by users. The idea that it is acceptable for DAC to have lower assurance than mandatory controls has gained some acceptance recently, but to suggest that DAC be implemented in user programs of no assurance whatsoever would be going too far. In fact, we do not suggest this. Elsewhere in this report, we discuss the possibility that using organizations would in some cases have to develop the expertise required to write multilevel secure managers. We here suggest that all using organizations that wish to extend the system by defining new types and writing managers for them will need to have the capability of writing managers of at least C2 assurance, in order to provide acceptable discretionary controls.

The choice of location for the encryption mechanism was fairly straightforward. Encryption of data being sent out over an untrusted network is done in or near the IP sublayer of the Network layer. This is the lowest place in the OSI model where end-to-end encryption can be done across the internet. Encryption at a lower point would interfere with the operation of the network layer in untrusted network nodes. Encryption at a higher point would result in the passing of more unencrypted information in message headers, and require individualized encryption mechanisms for each of the higher layer protocols.

5.2.2.2.2 Discretionary Controls The SDOS discretionary access control mechanisms are based on access control lists. However, several aspects of their design distinguish them from conventional approaches, including their support of roles, nondiscretionary rights, direct operations, intermodule connection control, and proxies.

Discretionary controls are type specific, as each type defines the privileges clients may have to invoke operations. Clients are identified by a *principal* (user) and *project* (task or group). A client may be associated with several different projects (though always acting on behalf of exactly one), and in a different capacity in each project. For example, a client may be an operator for one application but a developer for another. The different capacities, or *roles*, in which a client acts determine the operations that are available to the client to access objects. Roles tend to have similar meaning across many different types, thus providing an aspect of uniformity to the type independence of SDOS access control.

The operations in an access control list are divided into discretionary and nondiscretionary categories. The degree to which an operation is discretionary reflects the extent to which its entries in access control lists may be modified. Discretionary operation entries in an access control list may be modified by a group of users for a type having the *Controlling Group* role. Nondiscretionary operation entries in an access control list may only be modified by the System Manager. Since intervention by the System Manager is expected to be rare, the extent to which modifications to nondiscretionary operation ACL entries may change is highly constrained.

Direct operations are operations which only may be invoked by a trusted Terminal Interface Process. Thus, operations which should only be invoked by humans can be protected from inadvertent or malicious invocation by application software.

Object-oriented systems invariably create instances of nested object invocations, where client A invokes an operation handled by manager B, which in turns invokes an operation handled by manager C. For example, a process authenticating itself causes the Authentication Manager to invoke a nested call to set its process bindings for discretionary access control. In many instances, the nested call should be made on behalf of (i.e., using the identity of) the original client. Clients may send proxies, or a highly constrained part of their identity, to managers, which can then act on behalf of that client's limited identity. Proxies constrain the behavior of malicious managers and assure that managers cannot take on illegal or forged identities.

Module interconnection controls refers to the control of which modules (clients) can call other modules (managers). For example, only the Authentication Manager should be able to invoke the **SetProcessBindings** operation for a process. SDOS discretionary controls allow a manager to determine the identity of the client, even when the client is acting on behalf of another client by using a proxy.

5.2.2.3 Host Operating System Security

SDOS, like Cronus, is a collection of higher layer software that is implemented on top of an existing operating system in each host. Unlike Cronus, which is implemented on top of systems such as UNIX and VMS which are rated below B, SDOS must be implemented on top of a secure host operating system with at least a B rating. This is necessary in order to provide the required assurance that the SDOS security features cannot be tampered with. The Criteria mandate at least B2 assurance for multilevel security. The design objective of SDOS is an A1 rating. Thus, the host operating system(s) on top of which SDOS is implemented must have a minimum of a B2 rating, and ratings of B3 or A1 are more desirable.

5.2.2.3.1 Advantages and Disadvantages of MLS Support Operating systems having B2 through A1 ratings will have multilevel security policies built into them. These policies will all be different from the one designed for SDOS. Some may be similar, while others may be completely incompatible. The reason for building on top of a secure system is to benefit from its assurance. The policy comes with it for free, and we must decide what to do with it—use it or ignore it. The temptation is strong to use it, for economic reasons. This may or may not be possible, depending on the policy of the particular system chosen. Any decision to use the policy of an existing system will involve some changes to the designed SDOS policy. These changes may range from changes so fundamental as to be unacceptable, to changes in unimportant details.

A decision to ignore the policy of an existing system also has its drawbacks. The security mechanisms in the existing system will continue to operate and contribute to overhead, even if SDOS is built on top of it in such a way that the existing policy does not restrict SDOS (for example, by labeling all SDOS entities with the same access class label in the existing system's label set). This decision will also tend to increase the SDOS implementation cost.

5.2.2.3.2 Desirable Host Operating System Properties A secure operating system that is a good candidate on which to implement SDOS will provide the following:

- assured process separation — the ability to prevent direct interprocess communication that is not controlled by the system;
- non-interference with process operation — SDOS processes responsible for security must not be tampered with;

- stable storage — data needed for enforcing security, such as user authentication data, must be stored in a fault-tolerant and protected manner.

In addition, the good candidate will have a multilevel security policy that lends itself to being used to implement the SDOS policy, rather than being ignored.

5.2.2.4 Network Security

5.2.2.4.1 Open vs Closed Networks A closed network is one whose nodes and inter-node communications media are under the physical control of the using organization, such that their security can be assured, making it practical to implement multilevel security in the nodes. An open network, on the other hand, uses public communications media (such as phone lines or radio signals) and public nodes, such as those in commercial packet switched networks or in the ARPANet. Physical security of the nodes and media of a public network can obviously not be assured, so multilevel security is impractical. It is a design objective of SDOS that the system be able to maintain its own security even when it is operating over an open network. The reason for this requirement is a practical one: open networks are prevalent, and becoming more so every day. Closed networks are relatively rare, usually unavailable, and costly and time-consuming to construct and maintain.

5.2.2.4.2 Encryption Encryption is the usual solution to the problem of maintaining security of communications across an open network. Two classes of encryption are of interest here: link encryption and end-to-end encryption.

Link encryption protects data on an insecure medium that is being used for communication between two secure network nodes. Messages being relayed through several nodes to an ultimate destination are decrypted and re-encrypted at each intermediate node.

End-to-end encryption is used by higher layer entities to protect data from untrusted lower layers or from untrusted nodes in an open network. Messages are encrypted by the sending higher layer entity, decrypted by the receiving higher layer entity, and remain encrypted while moving through lower layers and intermediate network nodes.

End-to-end encryption has the advantages that the encryption and decryption are only done once for each message, and that layers below the encrypting layer, and intermediate network nodes, do not have to be trusted. It has the disadvantage that message headers for the layers below the encrypting layer travel the network in the clear. The information in these headers is subject to traffic flow analysis. Further, as suggested earlier, there is a potential covert channel if untrusted software above the TCB is able to exercise control over lower layer operation in ways that would modulate the lower layer header information.

Link encryption has the advantage that all information carried on the insecure

medium is protected. It has the disadvantage that all layers down to the physical layer, and all intermediate network nodes, must be secure. In other words, link encryption implies a closed network.

Since the use of open networks is a requirement for SDOS, we have chosen to use end-to-end encryption, in the *IP* sublayer of the Network layer. This is the lowest point at which end-to-end encryption can be done without interfering with the activities of the network layer in intermediate untrusted nodes of an open network. This allows layers below the point of encryption in each SDOS host to be untrusted. Layers at and above the point of encryption, up through the TCB/application interface, must be in the TCB.

5.2.3 Formal Methods

One of the goals of this project has been to formally verify that the SDOS design meets the requirements of the SDOS security policy. This would give high assurance that the design is "secure". Much of our work toward this goal has focussed on verifying the part of the mandatory policy requiring constraints on information flow.

We have formalized the design of SDOS as a program in Gypsy [Good et al. 78]. To prove, using the Gypsy methodology, that a program meets its requirements, one must express those requirements as assertions that are true at particular times during the execution of the program. We found, however, that the information flow constraints of the SDOS policy cannot be directly expressed in this way. Other approaches were needed.

5.2.3.1 A New Security Methodology

Our emphasis on mandatory information flow security is a result of the emergence of a new methodology for security verification. The aim of this methodology is security verification through analysis. In other words, large designs can be decomposed into smaller ones, and the security of the larger can be inferred once the properties of the smaller are known. This approach has obvious merits, but it is only recently that it has been applied to formal specifications for information flow security.

The work of McCullough [McCullough 87] is a particular case of this new methodology. In his work, system components are defined in terms of their possible behaviours; the approach he used simplified and slightly modified the approach of CSP [Brookes et al. 84]. The hook-up, or composition, of two components is defined as in CSP. McCullough searched for, and found, a property that captures many desirable features of information flow security and is also a *composable* property: the hook-up of two components with the property is a new component with the property. We have called his security property *restrictiveness*, or *restriction*.

The verification work presented in this report ties into the new methodology. The SDOS security policy requires that the entire trusted part of the system be restrictive.

We have endeavored to show that the multi-level secure processes that comprise SDOS are each restrictive, so that the restrictiveness of the entire system can be inferred from composability. However, the problem of demonstrating the restrictiveness of each MLS process remains. One way in which we have handled this problem is to find other, simpler, properties, which when taken together, imply the restrictiveness property. These simpler properties are then proved, using Gypsy, for each component. We needed to develop special techniques for proving some of the simpler properties using Gypsy.

Other than restrictiveness, we formally defined several properties that are "security-like", in the sense that they also limit deducibility, and hence, information flow. Of primary importance is the property called *weak non-interference* (WNI). The WNI property limits deducibility in a way that is similar to the Goguen-Meseguer model [Goguen and Meseguer 82]. However, WNI is both weaker than restriction, and not composable. By conjuncting several other simple properties with WNI, however, we can infer restriction.

As stated earlier, Gypsy is ill-suited to direct verification of properties such as restriction and WNI. Each is a property of the form: "Given any history a , there must be a history b such that $P(a, b)$ holds." Essentially, one is required to show the existence of particular histories of a component. Gypsy embedded assertions, though, state requirements of individual histories, taken in isolation. They never directly imply the existence of any history. However, simply changing the specification language was unlikely to solve the problem: other popular specification methodologies used for proving invariant properties of state machines would fare no better.

We developed a technique for proving that WNI holds for a design expressed in Gypsy. The technique does not supply embedded assertions for the design itself, but rather, it first transforms the design into a new form, and then supplies assertions about the new form that imply WNI. The restrictiveness of a design is to be inferred from the fact that it satisfies WNI, plus other simpler properties. This "program transformation" technique essentially shows that an alternate history exists by constructing it from the actual history. The transformed Gypsy design contains the state variables and control structure of the original design, but replicated, so that the two histories can be constructed in parallel.

Decomposing restrictiveness into simpler properties can now be seen as an advantage, since these simpler properties turn out to be easier to handle in this program transformation technique.

In verifying the information flow security of various SDOS components, we found that the definition of security as restrictiveness may not always be appropriate. We needed generalizations of the property to permit the following:

- limited downgrading of information by way of covert channels;
- special protocols used in communication between components;
- assumptions about the boundary between the assured system components and

process and users with limited or no assurance.

The report contains some successes in these directions. However, the subject is far from closed.

5.3 Possible Future Directions

The work described in this report has only made a start toward the development of a secure distributed operating system. There is more work to be done, in a number of areas, ranging from the very theoretical to the very practical. Several of these areas are described below.

5.3.1 Prototype Implementation

The development of a prototype SDOS would have a number of useful results. It would allow the concepts described in this report to be tested and proved in a practical setting. It would uncover any ideas that look good on paper but prove to be impractical during implementation. It would allow measurements to be made of performance-critical operations and of covert channel bandwidths, and experiments to be made to speed up the former and slow down the latter. It would allow experimental applications to be developed using the facilities provided by SDOS, to determine their suitability for use in practical applications. It would be a first step toward the deployment of an operational secure distributed operating system.

5.3.2 Research Into Layering

Layering, abstraction, and data hiding are thought to be good software design methodologies, and their use in secure systems is mandated by the Criteria. In any layered software system that is undergoing evolution and change, there arise occasions where the addition of some function or the improvement of some existing function requires the passing of information and control across layer boundaries in ways that were not intended when the boundaries were originally defined. Often some compromise of the strict layering rules will be made to avoid the need to completely re-modularize the system and redefine the layers to accommodate the change being made. But it is usually thought that such a layer redefinition could always be done successfully if resources were available.

Layer violations in SDOS are especially troublesome, because we depend to some extent on the restriction of downward information flow between the layers to limit the bandwidth of some covert channels. For this reason, careful attention must be given to modularization and layering within SDOS. We have the layers of the SDOS kernel and the trusted managers, the layers of the multilevel secure operating system in the host,

and the layers of the OSI model. SDOS development involves the evolution of the Cronus kernel into the SDOS kernel, the addition of security features to the communications layers, and the integration of these two sets of layers with those of the TCB of the host operating system. It has been difficult to create a layer diagram that clearly and correctly represents the relationships between all of these components of the system.

Further work in this area might result in a redistribution of functions between the layers, that resolves these problems. On the other hand, it might result in the conclusion that the relationships between the entities in a system of this size and complexity are too complex to be described by the one-dimensional ordering represented by a traditional layer diagram, and that some multi-dimensional representation is more appropriate for describing their relationships.

5.3.3 Research into formal methods

Several points should be noted concerning the limitations of the formal methods of verification discussed here.

- In practice, the Gypsy program transformation technique is clumsy. The practitioner not only needs to combat the difficulties of using Gypsy, but must also carry out many tedious transformation steps manually. When an error is discovered in the transformed program, not only that program, but the original source must be corrected. Complicated assertions must be given at many places in the text of the transformed program. Many of the verification conditions have a repetitively similar form.

None of this is intrinsic to the method. Many of these difficulties could be relieved by automated support.

- In at least one place, the design of SDOS depends on non-determinism for its security. (The security database uses random numbers to generate identifiers securely.) However, the methods we developed for proving non-interference using Gypsy are not applicable to non-deterministic designs. Not only are Gypsy procedures intended to model just deterministic algorithms, but showing the existence of traces in a non-deterministic system is a harder problem in general than in the deterministic case.
- The method of decomposition called "input-limited restriction" is only a simple example of a larger search: if two components agree to communicate using some protocol, and the security of each is made dependent on whether the other obeys the protocol, what component properties and interesting protocols can be used to hook together a restrictive system?
- In cases that a covert channel could not be eliminated completely from an SDOS component, the component obviously could not be proved restrictive. But leaving such a component unverified is not satisfactory. Generalized versions of restriction

need to be found, such that limited violations of security are possible. The severity of the violation can be controlled and quantified. The work appearing at the end of chapter 4 is an attempt at such a generalization. However, more powerful extensions can undoubtedly be found.

Each of these points represents an avenue of possible future research.

Appendix A

The Gypsy Specifications

A.1 Notes on the Gypsy Specification

The Gypsy specification that follows was developed and parsed under the GVE, Version 2.05, running on the Symbolics 3600. The specification is not entirely straightforward, and therefore a few explanatory comments are in order.

The specification is divided into several Symbolics files. This has the advantage that proving a particular file may not require that all other files be loaded at once. Also, when file A is altered, in addition to A only those files which reference symbols in file A must be reloaded. A slight disadvantage is that the structure of symbol references between files will force files to be loaded in order. The files of the specification include: a file of globally-visible types declarations; a file of globally-visible function declarations; several MLS type managers; the MLS terminal interface processes (TIP). The type managers and the TIP may each be loaded separately, but only after the types and functions have been loaded.

The types defined in the type-declaration file apply throughout the spec. Type managers will have their own particular types, such as the names of particular operations they implement, and these are declared separately in each type manager file.

The types 'abst_op' and 'abst_type' must occur wherever the object model is used. There are a number of constants of each type which represent abstract types and operations known to the kernel. These include all operations which may be invoked on the host itself. They are declared in the global types files.

Gypsy does not allow dynamic creation and destruction of concurrent processes. Because all SDOS processes, hosts, and users will be represented by concurrent Gypsy processes, all such entities which could conceivably be created must be declared beforehand. Because the current version of Gypsy requires that cobegins of processes be indexed by integer types, we have given these potential entities 'names' which are integers. These names may be thought of as external names, or, in the case of SDOS

processes, as the names which identify each process to the underlying COS. They are not to be confused with SDOS internal names, such as UIDs. Part of the task of the system will be to generate the internal names, and possibly to remember various relations between internal and external names. The ranges of external names for users, hosts, and processes are declared in the global types file.

In order to specify communication between each instance of the kernel and type managers running on the same machine, we must give particular external names to the type managers we specify. The kernel itself is in a unique situation since all buffers lead either to or from it (representing COS-enforced process separation), and therefore no external name need be given to it. In contrast, each type manager we specify is assigned a specific integer; without loss of generality, these are assigned counting up from the minimum 'minproc'. The set of external names for all other processes is then the integer range [mingenericproc..maxproc].

Communication between kernel and local processes, and kernel and network, is represented by Gypsy buffers. There is only one type of communication event in the spec: 'sendmessage'. All buffer types are therefore equivalent. The specification shows each sendmessage as a record with fixed fields, the same for every message. Each sendmessage has a 'level' field which indicates the level of the message being sent.

In an actual system, messages may undergo packing and unpacking appropriate to the particular kind of message being sent. These transformations could be represented by functions which convert between an abstract (packed) data type and concrete (unpacked) types. However, displaying these functions certainly detracts from clarity of the specification, and instead we have included enough specific message fields that very few operations defined on the host type will require packing. Some of the fields may remain unused in specific messages, though. Operations defined by other type managers may require either more than this number of fields, or fields of different (Gypsy) type. In every such case, the fields 'sendmessage.invoke.param' and 'sendmessage.reply.param' hold an object of type 'data', which may be unpacked into the specific (Gypsy) types needed by particular type managers.

The COS is not represented explicitly. It could have been included as a separate, concurrently running process, sharing buffers with each local process and the network. Its functions would then be:

- handle network protocol.
- correctly identify the source of messages, and pass them through to the kernel.
- maintain the host's stable storage (represented as local Gypsy variables).

These three functions have been combined (implicitly) into the kernel specification. Interactions between kernel and COS have been modeled as updates to local variables, and the fact that operations may be left pending as a result of the kernel-COS interaction has been ignored.

MLS managers outside the kernel must control the level they affix to outgoing messages based on the levels of messages they receive. Demonstrating that this is done correctly is the central problem of multi-level security.

A.1.1 Conventions used in the Gypsy FTLs

This section aims to clarify some conventions used in the Gypsy specification.

The basic function of the system as specified is to support the invocation of abstract operations on abstract objects. The message switch supports the routing of such invocations and their replies, while the various managers support the processing of the abstract operations themselves. The managers and the message switch must agree on various matters of protocol at their interfaces. Our purpose in this section is to describe their protocol.

Every invocation, reply, or direct inter-process communication (IPC) is packaged in the same form: as a value of Gypsy type 'sendmessage'. The sendmessage type has these fields:

- The field '.control' is used to distinguish invocations ('InvokeOp') from replies ('ReplyOp') from IPC ('SendOp'). Different values for '.control' cause different methods for message routing in the message switch.
- The field '.invoke' is used to record parameters of the invocation, including its object, its abstract operation, and any operation-specific parameters. The client who initiates the invocation fills this field, and may expect that if and when a reply is received, the value of this field will be unchanged. A process sending IPC may also fill the '.invoke' fields with message-specific information, and may also expect that any reply to the send will leave the information unchanged.
- The field '.reply' is used to record responses to invocations and sends, including whether there was an error, the appropriate error code, and any operation-specific data to be returned. A manager which processes the invocation will normally fill this field, change the '.control' field from 'SendOp' to 'ReplyOp' and expect that the message switch will route the reply back to its client.
- At the time a message is delivered, the field '.sender' indicates the UID of the message's sender. For sends and invokes, the message switch simply fills in the UID of the sending process. For replies, the sender of the reply will normally copy the '.sender' field of a previous send or invoke into the '.sender' field of the reply to indicate its destination. In this case, the message switch then uses the '.sender' field to determine routing, but before delivering the message it overwrites the field with the UID of the sender of the reply. (In this way, an invoke can be returned to its sender merely by changing '.control' to 'ReplyOp'.)
- At the time a message is delivered, the field '.SenderHost' indicates the location of the message's sender. It is filled by the message switch. During processing of

a 'ReplyOp', the sender of the reply will normally copy the 'senderhost' field of a previous send or invoke into the 'senderhost' field of the reply to indicate the location of the destination. In this case, the message switch then uses the 'senderhost' field to determine routing, but before delivering the message it overwrites the field with the UID of its own host.

- The field '.receiver' is supplied by the client when the sendmessage is a 'SendOp'. It is expected that the message switch will route the message to this UID. In other cases, this field is not filled by the client, but rather by the message switch once the receiver is determined. In the case of an invoke, '.receiver' is filled once it is determined to which type manager the invoke will be routed. In the case of a reply, '.receiver' is copied by the message switch from 'sender'.
- The field '.transaclbl' is used to assign a transaction number to an invocation and any secondary invocations and replies which may follow from it. This will normally be needed by a manager to distinguish replies to invokes which may coincidentally have the same value for the 'invoke' field. It will also be useful for relating nested transactions. Normally a manager receiving an invoke or a send will generate a new transaction number to be used for all ensuing processing. However, when replying to an invoke or send, a manager is expected to return its client's transaction number unchanged.
- The field '.level' is used to indicate the level of the sendmessage.

A number of special sub-fields have been supplied as components of 'invoke' and 'reply'. These are normally filled with information needed by particular, frequently-occurring invocations. When an invoke is made on the host, to read the ODB or the SDB, for example, the 'invoke.object' subfield is the UID of the host, while the 'invoke.objpar' subfield may be filled with the UID of an object to which the invocation is related.

The subfields 'invoke.param' and 'reply.param' are catch-all fields which can be filled with data which is peculiar to each particular abstract type.

Processes in the Gypsy specification are not expected to fill all these fields for every message. In most cases, there are many fields which are not relevant to the particular message being sent.

A.2 Global Type Declarations

```
scope sdos =
begin
```

```
{ some general SDOS types }
```

```

type data = pending;
type uid = pending;
type uidset = set of uid;
type uidseq = sequence of uid;

{ types relating to multi-level security }
type level = pending;
type label = set of level;
const l:label := pending;
const sys_hi: level := pending;
const host_hi: level := pending;
const host_lo: level := pending;
const sys_lo: level := pending;
type mls_attrib = boolean;

{ abstract types of the object model }
type abst_type = pending;
const ProcessType: abst_type := pending;
const HostType: abst_type := pending;

type abst_messages = (InUse, UndefOp, NotOpen, AlreadyOpen, NoMsg,
                      NoPermission, DoesNotExist, InvalidOp, IncorrectLevel,
                      AlreadyExists, Waiting,
                      ObjectOnly, ManagerOnly, ObjectandManager);

{ abstract operations of the object model }
type abst_op = pending;

const LocateUid: abst_op := pending;
const CreateODBEntry: abst_op := pending;
const ReadODBEntry: abst_op := pending;
const WriteODBEntry: abst_op := pending;
const ModifyODBEntry: abst_op := pending;
const RemoveODBEntry: abst_op := pending;
const CopyODBEntry: abst_op := pending;
const ReplicateODBEntry: abst_op := pending;
const DereplicateODBEntry: abst_op := pending;
const CreateSDBEntry: abst_op := pending;
const ReadSDBEntry: abst_op := pending;
const ModifySDBEntry: abst_op := pending;
const RemoveSDBEntry: abst_op := pending;
const ReplicateSDBEntry: abst_op := pending;
const DereplicateSDBEntry: abst_op := pending;
const IncrementReplicaNo: abst_op := pending;
const DecrementReplicaNo: abst_op := pending;

const SetProcessBindings: abst_op := pending;
const ShowProcessBindings: abst_op := pending;

```



```

const ResetProcessBindings: abst_op := pending;
const ChangeActiveCCI: abst_op := pending;
const DetermineClientId: abst_op := pending;
const ObtainProxy: abst_op := pending;
const CreateProc: abst_op := pending;
const RemoveProc: abst_op := pending;

```

```

{ names for all possible users }
const minuser: integer := pending;
const maxuser: integer := pending;
type username = integer[minuser..maxuser];
type userboolarr = array (username) of boolean;

```

```

{explicit names for special users}
const system_manager: integer:= minuser;

```

```

{ names for all possible hosts }
const minhost: integer := pending;
const maxhost: integer := pending;
type hostname = integer[minhost..maxhost];

```

```

type hostboolarr = array (hostname) of boolean;
type hostnamemap = mapping from uid to hostname;
type hosttblmap = mapping from uid to label;

```

```

{ names for all possible processes,
  including type managers, but excluding kernel and tips }
const minproc:integer := pending;
const maxproc:integer := pending;
type procname = integer[minproc..maxproc];
type procname_3 = integer[minproc+3..maxproc];
type procboolarr = array (procname) of boolean;
type procnamemap = mapping from uid to procname;

```

```

{ names for all possible tip processes }
const mintip:integer := pending;
const maxtip:integer := pending;
type tipname = integer[minproc..maxproc];
type tipnamemap = mapping from uid to tipname;
type tiplist = sequence of uid;

```

```

{ explicit names for special type manager processes }
const catalog_manager: procname := minproc;
const authmgr: procname := minproc +1;

```

```

const file_manager: procname := minproc +2;
const dacmgr: procname := minproc +3;
const configmgr: procname := minproc +4;
const pkgmgr: procname := minproc +5;
const mingenericproc: procname := minproc +6;

{ events of the object model }
type invocation = record(operation: abst_op;
                        object: uid;
                        objpar: uid;
                        objpar1: uid;
                        objlbl: label;
                        param: data;
                        flag: boolean);

type response = record(error: boolean;
                      objpar: uid;
                      objlbl: label;
                      param: data;
                      errcode: abst_messages;
                      flag: boolean);

{ types for communication events and event histories }
type controlop = (InvokeOp, SendOp, ReplyOp);
type transacno = pending;
type transacno set = set of transacno;
type sendmessage = record(sender: uid;
                          senderhost: hostname;
                          receiver: uid;
                          control: controlop;
                          operateup_enabled: boolean;
                          transaclbl: transacno;
                          invoke: invocation;
                          reply: response;
                          level: level);

type hostbuf = buffer of sendmessage;
type hostbufarr = array (hostname) of hostbuf;
type procbuf = buffer of sendmessage;
type procbufarr = array (procname) of procbuf;
type eventseq = sequence of sendmessage;
type eventarray = array(level) of eventseq;

type object = uid;
type client = uid;

{ discretionary access control (DAC) types }
type principal = pending;
type accessrightset = set of abst_op;
type ACL = mapping from principal to accessrightset;

```

```

type DAC = mapping from uid to ACL;
const fixed_auth: DAC := pending;

```

```

end; { scope }

```

A.3 Global Function Declarations

```

scope sdos = {extending sdos scope}
begin

{ auxiliary functions used in kernel }

function dominates(l1,l2:level) : boolean = pending;
{ return true if security label l1 dominates l2 }

function uidhost(id: uid) : uid =
begin
    exit (assume uidtype(result) = HostType);
end;
{ return the uid of the host named in the host field of the uid }

function uidtype(id: uid) : abst_type = pending;
{ return the abstract type field of the uid }

function generic_object(at: abst_type) : uid =
begin
    exit (assume uidtype(result) = at);
end;
{ return the uid of the generic object of this type }

function type_object(id: uid) : uid =
begin
    exit (assume result = generic_object(uidtype(id)));
end;
{ a shorthand for composition of the previous two functions }

function procuid(pn: procname; table: procnamemap) : uid =
begin
    exit (assume pn in range(table) -> table(result) = pn);
end;
{ compute the inverse of a process table }

function hostuid(hn: hostname; table: hostnamemap) : uid =
begin

```

```

    exit (assume hn in range(table) -> table(result) = hn);
end;
{ compute the inverse of a host table }

function host_label(arg1:uid; table:hostlblmap) : label =
begin
    exit (assume arg1 in domain(table) -> result = table(arg1));
end;

end; {scope kernel_scope for auxil}

```

A.4 The File Manager Specification

A.4.1 Local Function, Procedure and Type Declarations

```

scope sdos =
begin

const OpenFile: abst_op := pending;
const ReadFile: abst_op := pending;
const WriteFile: abst_op := pending;
const CloseFile: abst_op := pending;
const CreateFile: abst_op := pending;
const DeleteFile: abst_op := pending;
const marker1: abst_op := pending;
const marker2: abst_op := pending;

type accessmode = (Blank, Read, Write, ReadWrite);
type objectmap = mapping from object to object;
type cliobjmap = mapping from client to objectmap;
type ghostmap = mapping from level to cliobjmap;
type access_level_list = mapping from client to level;
type access_level_table = mapping from object to access_level_list;
type access_mode_list = mapping from client to accessmode;
type access_mode_table = mapping from object to access_mode_list;

```

{EXPLANATION OF KEY FUNCTIONS/PROCEDURES:

update_generic : makes an entry into the table generic

purge_generic : removes an entry from the table generic

already_open : determines if the current client already has access to the current object at the current level.

[note: this does check level not accessmode.

This is a sufficient check because we will not allow a client to open the same object to read and write at the same level. If the client does desire to read and write to the object then he must open it for readwrite.]

has_access : checks if the current client already has access to the current object in the current mode and that the access is permissible. [note: this does not check level of access]

write_locked_by_another : determines whether

the object is being accessed by another

client at the same level. [The level check is sufficient to determine that this is a write because read requests are always mapped to ghost files. This has the added benefit that it would disallow writes to the ghost files that have been created for the read operation.]

fill_call, fill_reply, fill_call_temp : updates necessary fields in a variable of type sendmessage. This would then be the invocation on the kernel or a reply to the client/kernel.

fill_call_temp was necessiated by a bug in handling of record types in Gypsy.

}

```
function already_open(call:sendmessage;TABLE:access_level_table): boolean =
begin
```

```
    exit (assume result =
```

```
        (call.invoke.object in domain(TABLE) and
```

```
        call.sender in domain(TABLE(call.invoke.object)) and
```

```
        call.level = TABLE(call.invoke.object)(call.sender)));
```

```
end;
```

```
function write_locked_by_another(call:sendmessage;TABLE:access_level_table):
```

```
boolean =
```

```
begin
```

```
    exit (assume result =
```

```
    (some c1:client,
```

```
        call.invoke.object in domain(TABLE) and
```

```
        c1 in domain(TABLE(call.invoke.object)) and
```

```
        c1 ne call.sender and
```

```
call.level = TABLE(call.invoke.object)(c1));
end;
```

```
function has_access(call: sendmessage;
                    table: access_mode_table): boolean =
begin
  exit (assume result =
        (call.invoke.object in domain(table) and
         call.sender in domain(table(call.invoke.object)) and
         allowed(call.invoke.operation, table(call.invoke.object)(call.sender))));
end;
```

```
procedure update_OPENFOR(call1: sendmessage; var TABLE: access_mode_table) =
begin
  exit ((all o1:object, all c1:client, o1 in domain(TABLE') and
        c1 in domain(TABLE'(o1)) -> o1 in domain(TABLE) and
        c1 in domain(TABLE(o1)) and
        TABLE(o1)(c1) = TABLE'(o1)(c1)) and

        [all c:sendmessage,
         (c ne call1) and
         c.invoke.object in domain(TABLE) and
         c.sender in domain(TABLE(c.invoke.object)) ->
         c.invoke.object in domain(TABLE') and
         c.sender in domain(TABLE'(c.invoke.object)) and
         TABLE(c.invoke.object)(c.sender) =
         TABLE'(c.invoke.object)(c.sender)] and
        (call1.invoke.object in domain(TABLE) and
         call1.sender in domain(TABLE(call1.invoke.object)) and
         TABLE(call1.invoke.object)(call1.sender) =
         mode_param(call1.invoke.param)));
end;
```

```
procedure update_OPENAT(call1: sendmessage; var TABLE: access_level_table) =
begin
  exit ((all o1:object, all c1:client, o1 in domain(TABLE') and
        c1 in domain(TABLE'(o1)) -> o1 in domain(TABLE) and
        c1 in domain(TABLE(o1)) and
        TABLE(o1)(c1) = TABLE'(o1)(c1)) and

        [all c:sendmessage,
         (c ne call1) and
         c.invoke.object in domain(TABLE) and
         c.sender in domain(TABLE(c.invoke.object)) ->
         c.invoke.object in domain(TABLE') and
```

```

                                c.sender in domain(TABLE'(c.invoke.object)) and
                                TABLE(c.invoke.object)(c.sender) =
TABLE'(c.invoke.object)(c.sender)] and
    (call1.invoke.object in domain(TABLE) and
    call1.sender in domain(TABLE(call1.invoke.object)) and
    TABLE(call1.invoke.object)(call1.sender) = call1.level));
end;

```

```

procedure update_GT(call1: sendmessage; var TABLE: ghostmap) =
begin
    exit ((all o1:object, all c1:client, all l1:level,
l1 in domain(TABLE') and
    c1 in domain(TABLE'(l1)) and
    o1 in domain(TABLE'(l1)(c1)) ->
l1 in domain(TABLE) and
c1 in domain(TABLE(l1)) and
                                o1 in domain(TABLE(l1)(c1)) and
                                TABLE(l1)(c1)(o1) =
TABLE'(l1)(c1)(o1)) and

    (all c:sendmessage,
    c ne call1 and
    c.level in domain(TABLE) and
    c.sender in domain(TABLE(c.level)) and
    c.invoke.object in domain(TABLE(c.level)(c.sender)) ->
    c.level in domain(TABLE') and
    c.sender in domain(TABLE'(c.level)) and
                                c.invoke.object in domain(TABLE'(c.level)(c.sender)) and
                                TABLE(c.level)(c.sender)(c.invoke.object) =
TABLE'(c.level)(c.sender)(c.invoke.object)) and

    (call1.level in domain(TABLE) and
    call1.sender in domain(TABLE(call1.level)) and
    call1.invoke.object in domain(TABLE(call1.level)(call1.sender)) and
    TABLE(call1.level)(call1.sender)(call1.invoke.object) =
    call1.reply.objpar)
);
end;

```

```

procedure purge_OPENFOR(call1: sendmessage; var TABLE: access_mode_table) =
begin
    exit ((all c1:client, all o1:object,
    o1 in domain(TABLE) and

```

```

        c1 in domain(TABLE(o1)) -> o1 in domain(TABLE') and
                                c1 in domain(TABLE'(o1)) and
                                TABLE(o1)(c1) = TABLE'(o1)(c1)) and
    (all c:sendmessage,
    c ne call1 and
        c.invoke.object in domain(TABLE') and
        c.sender in domain(TABLE'(c.invoke.object)) ->
c.invoke.object in domain(TABLE) and
        c.sender in domain(TABLE(c.invoke.object)) and
        TABLE(c.invoke.object)(c.sender) =
TABLE'(c.invoke.object)(c.sender)) and
        not (call1.sender in domain(TABLE(call1.invoke.object))));
end;

```

```

procedure purge_OPENAT(call1: sendmessage; var TABLE: access_level_table) =
begin
    exit ((all c1:client, all o1:object,
        o1 in domain(TABLE) and
        c1 in domain(TABLE(o1)) -> o1 in domain(TABLE') and
                                c1 in domain(TABLE'(o1)) and
                                TABLE(o1)(c1) = TABLE'(o1)(c1)) and

    (all c:sendmessage,
    c ne call1 and
        c.invoke.object in domain(TABLE') and
        c.sender in domain(TABLE'(c.invoke.object)) ->
c.invoke.object in domain(TABLE) and
        c.sender in domain(TABLE(c.invoke.object)) and
        TABLE(c.invoke.object)(c.sender) =
TABLE'(c.invoke.object)(c.sender)) and
        not (call1.sender in domain(TABLE(call1.invoke.object))));

end;

```

```

procedure purge_GT(call1: sendmessage; var TABLE: ghostmap) =
begin
    exit ((all c1:client, all o1:object, all l1:level,
l1 in domain(TABLE) and
        c1 in domain(TABLE(l1)) and
        o1 in domain(TABLE(l1)(c1)) ->
l1 in domain(TABLE') and
c1 in domain(TABLE'(l1)) and
                                o1 in domain(TABLE'(l1)(c1)) and
                                TABLE(l1)(c1)(o1) = TABLE'(l1)(c1)(o1)) and

    (all c:sendmessage,

```



```

        c ne call1 and
        c.level in domain(TABLE') and
        c.sender in domain(TABLE'(c.level)) and
        c.invoke.object in domain(TABLE'(c.level)(c.sender)) ->
c.level in domain(TABLE) and
c.sender in domain(TABLE(c.level)) and
        c.invoke.object in domain(TABLE(c.level)(c.sender)) and
        TABLE(c.level)(c.sender)(c.invoke.object) =
        TABLE'(c.level)(c.sender)(c.invoke.object)) and

        not (call1.invoke.object in domain(TABLE(call1.level)(call1.sender))) );
end;

```

```

function has_disc_access(call: sendmessage;
                        TABLE: DAC) : boolean = pending;

{ do discretionary access controls defined in 'table'
  authorize the file request defined by 'call' ? }

```

```

function allowed(op: abst_op; acc: accessmode) : boolean =
begin
    exit (assume result =
        ((op = ReadFile and acc in [set: ReadWrite, Read]) or
         (op = WriteFile and acc in [set: Write, ReadWrite]) or
         (op = CloseFile and acc in [set: ReadWrite, Write, Read])));
end;

```

```

function mode_param(par: data) : accessmode = pending;
{ decode a parameter into an accessmode argument }

```

```

function purge(trace: eventseq) : eventseq =
begin
    exit (assume result =
        if dominates(1, last(trace).level)
        then purge(nonlast(trace)) <: last(trace)
        else purge(nonlast(trace)) fi);
end;
{ remove from trace all events not dominated by 1}

```

```
function dominates(l1: level; l2: level) : boolean = pending;
```

```
function unset(l:label):level = pending;
{the label returned by ReadSDBEntry is actually going to be a one element
set because multilevel files are not allowed.
This function just draws the element out of the set. The need is
to ensure that the parameters of the right type are fed into dominates}
```

```
function has_ghost(call:sendmessage;TABLE:ghostmap): boolean =
begin
  exit (assume result =
    (call.level in domain(TABLE) and
    call.sender in domain(TABLE(call.level)) and
    call.invoke.object in domain(TABLE(call.level)(call.sender)) )
  );
end;
```

```
function map_onto_ghost(call:sendmessage;TABLE:ghostmap) :object =
begin
  exit (assume result =
    if call.level in domain(TABLE) and
    call.sender in domain(TABLE(call.level)) and
    call.invoke.object in domain(TABLE(call.level)(call.sender))
    then TABLE(call.level)(call.sender)(call.invoke.object)
    else call.invoke.object
  fi);
end;
```

```
function fill_call(call:sendmessage;op:abst_op;
                  thisproc:uid;thishost:uid):sendmessage =
{note CopyODB copies from objpar to objpar1}
begin
  exit (assume result =
    call with (.control := InvokeOp;
              .sender := thisproc;
              .invoke := call.invoke with
                (.operation := op;
                 .object := thishost;
                 .param := call.invoke.param;
                 .objpar1 := call.reply.objpar)) );
end;
```

```
function fill_call_temp(call:sendmessage;op:abst_op;object1:object;
                      thisproc:uid;thishost:uid):sendmessage =
{note CopyODB copies from objpar to objpar1}
```

```

begin
  exit (assume result =
    call with (.control := InvokeOp;
              .sender := thisproc;
              .invoke := call.invoke with
                (.operation := op;
                 .object := thishost;
                 .param := call.invoke.param;
                 .objpar1 := call.reply.objpar)) );
end;

function fill_reply(call:sendmessage;thisproc:uid;thishost:uid;data1:data;
  flag:boolean;code:abst_messages):sendmessage =
begin
  exit (assume result =
    call with (.control := ReplyOp;
              .sender := thisproc;
              .receiver := call.sender;
              .senderhost := thishost;
              .reply := call.reply with
                (.error := flag;
                 .errcode := code;
                 .param := data1) ));
end;

end; {sdos scope for functions}

```

A.4.2 Main Procedure

```

scope sdos =
begin

  procedure file_manager(  thishost    : uid;

```

```

                                thisproc      : uid;
                                var port       : procbuf <input>;
                                var kport    : procbuf <output>) =

begin

{EXPLANATION OF KEY VARIABLES :

OPENFOR : table indexed by client and object, to store the current mode of
access.

OPENAT : table indexed by client and object to store the current level of
access.

ghost_table : table index by level, client and object to store the uid of
the ghost object.

pending_ops : calls queued when making invocations on the kernel (replies to
which need further processing) so as to be able to match replies with
invocations.
}
{Note: at any particular level, there is only one channel per client per
object.}


var call, reply, oldcall, tmpcall, host_call : sendmessage;
var i: integer;

var pending_ops: eventarray;
var OPENAT: access_level_table;
var GhostTable: ghostmap;
var OPENFOR: access_mode_table;

const DD:data := null(data);

loop

if call.control = InvokeOp
then
    {mapping the actual object to ghost object is necessary because:
     read accesses are handled by creating ghosts. Therefore, if
     a particular access already has a ghost, then we must deal
     w.th the ghost object instead of the true object
     Note: if no ghost exists for that access, then map_onto_ghost
     will return the actual object uid}

```

```

tmpcall := call;
tmpcall.invoke.object := map_onto_ghost(call,GhostTable);

case call.invoke.operation

is Openfile:

    {does the client have the object already open?
     If so return error}

    if already_open(tmpcall,OPENAT) or already_open(call,OPENAT)
    then
        reply := fill_reply(call,thisproc,thishost,DD,true,AlreadyOpen);
        send reply to kport;

        {is another client writing to the same object?
         (readwrite will be considered same as write)
         If so return error}

    elif write_locked_by_another(call,OPENAT) and
          (mode_param(call.invoke.param) = write or
           mode_param(call.invoke.param) = ReadWrite)
    then
        reply :=
            fill_reply(call,thisproc,thishost,DD,true,InUse);
        send reply to kport;

        {file is free, so commence check for legitimacy of request
         before granting access}

    else
        host_call := fill_call(call,ReadSDBEntry,thisproc,thishost);
        pending_ops(call.level) := pending_ops(call.level) <: call;
        send host_call to kport;

    end; {if already_open}

is ReadFile:

    {if the client has the object open in read or readwrite mode then
     invoke the necessary operation on the ODB
     else Error - note if object was opened for read then we
     would be dealing with the ghost, otherwise we would be dealing
     with the actual object.}

```

```

if not has_access(tmpcall, OPENFOR) or
    not already_open(tmpcall, OPENAT)
then
    reply := fill_reply(call, thisproc, thishost, DD, true, NotOpen);
    send reply to kport;
else
    host_call := fill_call_temp(call, map_onto_ghost(call,
        GHOSTTABLE), ReadODBEntry, thisproc, thishost);
    pending_ops(call.level) :=
        pending_ops(call.level) <: call;
    send host_call to kport;
end; {if not has_access}

```

is WriteFile:

```

{if the client has the object open in write or readwrite mode then
    invoke the necessary action on the ODB
    else Error - note in both cases we will be dealing with
    the actual object.}

```

```

if not has_access(call, OPENFOR) or
    not already_open(call, OPENAT)
then
    reply := fill_reply(call, thisproc, thishost, DD, true, NotOpen);
    send reply to kport;

else
    host_call := fill_call(call, WriteODBEntry, thisproc, thishost);
    pending_ops(call.level) :=
        pending_ops(call.level) <: call;
    send host_call to kport;

end; {if not has_access}

```

is CloseFile:

```

{if there was no access to begin with the return Error.
    Otherwise update the necessary data structures to
    indicate that client no longer has the object open.
    Also if the access had resulted in creating a ghost then
    invoke a RemoveODBEntry operation on the ODB to delete
    the ghostfile.}

```

```

if not has_access(tmpcall, OPENFOR) or
    not already_open(tmpcall, OPENAT)
then

```

```

    reply := fill_reply(call, thisproc, thishost, DD, true, NotOpen);
    send reply to kport;

else
    purge_OPENFOR(tmpcall, OPENFOR);
    purge_OPENAT(tmpcall, OPENAT);
    purge_GT(tmpcall, GhostTable);
end; {if not has_access}

if tmpcall.invoke.object ne call.invoke.object
then
    host_call := fill_call_temp(call, map_onto_ghost(call,
        GHOSTTABLE), RemoveODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;
end;

is CreateFile:

{pass on the request to the ODB as a CreateODBEntry invocation.
 If that succeeds then this invocation will.}

    host_call := fill_call(call, CreateODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;

is DeleteFile:

{pass on the request to the ODB as a RemoveODBEntry invocation.
 If that succeeds then this invocation will.
 Anybody using the object will be bumped off and so the
 data structures that reflect accesss to objects will
 be updated on successful return from the ODB}

    host_call := fill_call(call, RemoveODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;

else: {of case}

{since the invocation is not an operation supported by the
 file-manager then return error}

    reply := fill_reply(call, thisproc, thishost, DD, true, UndefOp);
    send reply to kport;

```

```

    end; {case}

else {if call.control = ReplyOp}

{REPLIES FROM THE KERNEL -
in cases where an entry exists in pending_ops for the reply, additional
action is taken, otherwise, the response is relayed to the client}

i := 0;

loop

    {have we walked through pending_ops?}

    if i > size(pending_ops(call.level)) then leave;

    {is the reply from the kernel in response to the
    current element in pending_ops}

    elif call.level = pending_ops(call.level)(i).level and
        call.transac1bl = pending_ops(call.level)(i).transac1bl
    then

        oldcall := pending_ops(call.level)(i);

        pending_ops(call.level) :=
            pending_ops(call.level) with (seqomit(i));

        {additional action can be avoided if the reply from
        the kernel is an error}

        if not call.reply.error
        then

            case oldcall.invoke.operation

            is OpenFile:

                {this must be in reply to the ReadSDBEntry invoked
                by the file-manger to check the level of the
                object.}

                {if the level of the invoke = level of the object and
                the request is to write or readwrite then grant the

```



```

    request and update necessary data structures to
    reflect the client's access to the object}
{the additional check as whether another client is
accessing the file in write/readwrite mode is to
close the crack on cases where seperate write
requests come in and are being considered because
neither request has succeeded yet. In other words, if
the check is not made, it is possible that two
clients could end up with write access to the same
file. Even if that happened, it would be no security
violation}

if unset(call.reply.objlbl) = oldcall.level and
    (mode_param(oldcall.invoke.param) = write or
     mode_param(oldcall.invoke.param) = ReadWrite) and
    not write_locked_by_another(oldcall,OPENAT)
then
    update_OPENFOR(oldcall, OPENFOR);
    update_OPENAT(oldcall, OPENAT);
    reply := fill_reply(oldcall,thisproc,thishost,
                        DD,false,NoMesg);

    send reply to kport;

{if the level of the invoke dominates the level of
the object and the request is to read then
invoke a CreateODBEntry operation on the ODB to
create a ghost file}

elif (dominates(oldcall.level,
                unset(call.reply.objlbl)) and
     mode_param(oldcall.invoke.param) = read)
then
    host_call :=
        fill_call(oldcall,CreateODBEntry,thisproc,
                  thishost);

    pending_ops(oldcall.level) :=
        pending_ops(oldcall.level) <: oldcall
        with (.invoke.operation := marker2);
    send host_call to kport;

else
    reply := fill_reply(oldcall,thisproc,thishost,
                        DD,true,NoPermission);

    send reply to kport;

end;

is marker2:

```

```

{now that the ghost file has been successfully created,
 copy contents of actual object onto the ghostfile}

host_call :=
    fill_call(oldcall, CopyODBEntry, thisproc, thishost);
pending_ops(oldcall.level) :=
    pending_ops(oldcall.level) <: oldcall with
        (.invoke.operation := marker1);
send host_call to kport;

is marker1:
{since the ghost is now a mirror of the actual object
 grant the open to read request and update data
 structures to map actual to ghost and to reflect
 client's access to the object}

update_OPENFOR(oldcall, OPENFOR);
update_OPENAT(oldcall, OPENAT);
update_GT(oldcall, GhostTable);
reply := fill_reply(oldcall, thisproc, thishost,
                    DD, false, NoMsg);
send reply to kport;

is DeleteFile:

{since RemoveODBEntry was successful, no client can
 have access to the file, therefore, update the
 relevant data structures}

purge_OPENFOR(call, OPENFOR);
purge_OPENAT(call, OPENAT);
purge_GT(call, GhostTable);
reply := fill_reply(oldcall, thisproc, thishost,
                    DD, false, NoMsg);
send reply to kport;

else: {case}

{no special action needs to be taken. Just relay
 the response from the kernel to the client}

reply := fill_reply(oldcall, thisproc, thishost,
                    call.reply.param, call.reply.error, call.reply.errcode);
send reply to kport;
end; {case}

leave;

```

```
    else {if not call.reply.error}

        {since the reply from the kernel was an error - relay
         the error to the client}

        reply := fill_reply(oldcall,thisproc,thishost,DD,
                           call.reply.error,call.reply.errcode);
        send reply to kport;

        leave;

    end; {if not call.reply.error}

    else {if i = ...}
        i := i + 1;
    end;{if i > size(pending_ops)}

end; {loop}

end; {if InvokeOp}

end; {outermost loop}
end; {file_manager}

end; {sdos scope}
```

A.5 The Catalog Manager Specification

A.5.1 Local Function, Procedure and Type Declarations

```

scope sdos =    { extending scope sdos }
begin
{ The catalog manager.
  Types relating to the catalog are declared first, followed by various
  functions auxiliary to the catalog manager, followed by the procedures
  which define the operations on directories. }

{ The abstract type of directories,
  i.e., of objects managed by the catalog manager. }
const DirectoryType : abst_type := pending;

{ The specific abstract operations defined by the catalog manager. }
const Lookup : abst_op := pending;
const CreateAlias : abst_op := pending;
const RemoveAlias : abst_op := pending;
const CreateDirectory : abst_op := pending;
const RemoveDirectory : abst_op := pending;
const ReplicateDirectory : abst_op := pending;
const DereplicateDirectory : abst_op := pending;

{ The components of data stored in each directory. }
type identifier = pending;           {symbolic name identifiers}
type path = sequence of identifier;  {directory path}
type alias_list = mapping from identifier to uid; {collection of aliases}
type directory = record(replica: boolean;        {whether replicated}
                        primary: uid;            {primary catmgr}
                        list: alias_list);       {uids of subdirs and objects}

{ Relations between pending operations, indexed by transaction number.
  A variable of type 'pending_state' will be declared for each transaction
  in progress. }
type pending_state = record(orig: sendmessage;    { original request }
                           curr: sendmessage;    { current secondary req }
                           replica: boolean;      { is object replicated ? }
                           primary: uid;          { object's primary catmgr }
                           waiting: boolean;      { transaction blocked ? }
                           count: integer);       { number of responses from
                                                    catmgr replicas }

type pending_list = mapping from transacno to pending_state;

{ The list of catalog managers running on other hosts,
  and their associated host uids.
  These lists are specified as static, but in general, operations
  could be provided to modify them when catalog manager replicas
  are created or destroyed. }

```

```

const catmgr_replicas : uidseq := pending;
const catmgr_hosts : uidseq := pending;
lemma catmgr_locate(i: integer) =
  (size(catmgr_replicas) = size(catmgr_hosts) and
   i <= size(catmgr_hosts) -> catmgr_hosts[i] = uidhost(catmgr_replicas[i]));

{ The three types of multicasts to other hosts }
type sendmode = (every, notlocal, single);

{-----}
{ Catalog Manager functions }

{ function to recognize any defined catalog manager operation. }

function is_catmgr_op(sm: sendmessage) : boolean =
begin
  exit (assume result =
        (sm.invoke.operation = Lookup or
         sm.invoke.operation = CreateAlias or
         sm.invoke.operation = RemoveAlias or
         sm.invoke.operation = CreateDirectory or
         sm.invoke.operation = RemoveDirectory or
         sm.invoke.operation = ReplicateDirectory or
         sm.invoke.operation = DereplicateDirectory));
end; { is_catmgr_op }

{ function to generate a secondary invocation on a kernel,
  stemming from an original invocation on the catalog.
  The object of the secondary operation is a host, and the object of
  the original catalog invocation becomes a parameter in '.objpar'. }

function kernel_request(sm: sendmessage; hostuid: uid; op: abst_op) :
  sendmessage =
begin
  exit (assume (result.invoke.object = hostuid and
                result.invoke.operation = op and
                result.invoke.objpar = sm.invoke.object and
                result.level = sm.level));
end; { kernel_request }

{ function to determine whether a directory is locked by its primary
  catalog manager because a distributed update is in progress.
  'is_locked' will be true if the local catmgr has already begun servicing
  some transaction on this object 'obj', and after reading the local ODB
  has found that 'obj' is replicated, that the primary manager for 'obj'
  is itself, and that the operation in progress is not a Lookup.
  It will mark this transaction as possessing the lock by making
  .waiting = false. }

```

```

function is_locked(obj: object; pending_ops: pending_list; myuid: uid) :
    boolean =
begin
    exit (assume (result =
        some trno: transacno,
            trno in domain(pending_ops) and
            pending_ops(trno).orig.invoke.object = obj and
            pending_ops(trno).orig.invoke.operation ne Lookup and
            pending_ops(trno).replica and
            pending_ops(trno).primary = myuid and
            not pending_ops(trno).waiting));
end; { locked }

```

{ function to determine whether any transactions in 'pending_ops' are waiting for object 'obj' to be unlocked }

```

function is_waiting(obj: object; pending_ops: pending_list) : boolean =
begin
    exit (assume (result =
        some trno: transacno,
            trno in domain(pending_ops) and
            pending_ops(trno).orig.invoke.object = obj and
            pending_ops(trno).waiting));
end;

```

{ function to return the transaction number of a waiting transaction, assuming one exists }

```

function find_waiting(obj: object; pending_ops: pending_list) : transacno =
begin
    entry is_waiting(obj, pending_ops);
    exit (assume (result in domain(pending_ops) and
        pending_ops(result).orig.invoke.object = obj and
        pending_ops(result).waiting));
end;

```

{ function to return the effective level of a transaction number. Making transaction numbers dependent on levels is necessary for security in a deterministic specification. If transaction numbers could be picked randomly from an infinite pool, then indexing by security levels could be avoided. }

```

function transacno_level(trno: transacno) : level = pending;

```

{ function to generate a new unique transaction number of a given level }

```

function generate_new_transacno(used: transacno;
                                lev: level) : transacno =
begin
    exit (assume (not result in used and
                  transacno_level(result) = lev));
end;

```

```

{ function to unpack directory data }
function data_to_directory(d: data) : directory = pending;

```

```

{ function to pack directory data }
function directory_to_data(d: directory) : data = pending;

```

```

{ function to unpack directory path }
function data_to_path(d: data) : path = pending;

```

```

{ function to pack directory path }
function path_to_data(p: path) : data = pending;

```

```

{ function to unpack symbolic identifier }
function data_to_identifier(d: data) : identifier = pending;

```

```

{ function to pack symbolic identifier }
function identifier_to_data(i: identifier) : data = pending;

```

{ This procedure handles replies to previous sends and invokes.
 If the reply is an error, abort the transaction no matter what stage it has reached. If the reply is for a previous SendOp, then the primary manager for the object is reporting completion; its reply is then passed back to the client.
 Otherwise, the reply is for a previous InvokeOp, handled by some host. If the invoke was a Read-, Create-, or RemoveODBEntry, then it was a single (not multicast) invoke on the local host (these invokes are never multicast by the catalog manager). ReadODB is handled separately; Create- and RemoveODB are intermediate operations from Create- and RemoveDirectory, and they proceed to invoke ModifyODB.

A reply to any other invoke is the last step in processing of its transaction. If the directory is not replicated, the reply is passed back to the client. Otherwise, the local manager is the primary manager for this object and will wait for successful replies from every catalog manager host before replying to the client. If not every catmgr host replies successfully, then the reply to the client will report failure and future messages from other catmgr hosts will be ignored. }

```

procedure reply_handler(   rcv: sendmessage;
                           var snd: sendmessage;
                           var out: boolean;
                           var multicast: sendmode;
                           myuid: uid;
                           var pending_ops: pending_list) =
begin
  var trstate: pending_state;
  var origmesg, currmesg: sendmessage;
  var dir: directory;
  var ident: identifier;

  trstate := pending_ops(rcv.transaclbl);
  origmesg := trstate.orig;
  currmesg := trstate.curr;
  snd := origmesg;                                { default message }
  out := true;                                    { default value }
  multicast := single;                            { default value }

  if rcv.reply.error
  then                                             { abort transaction }
    snd.control := ReplyOp;
    snd.reply.error := true;
    remove_pending_ops(rcv.transaclbl);

  elif rcv.sender in catmgr_replicas
  then                                             { this is reply to a SendOp }
    snd.control := ReplyOp;                        { or secondary InvokeOp on }
    snd.reply := rcv.reply;                        { a remote catalog mgr }
    remove_pending_ops(rcv.transaclbl);

  elif rcv.invoke.operation = ReadODBEntry        { sender is local host }
  then
    read_ODB_handler(rcv, snd, out, multicast, myuid, pending_ops);

  elif rcv.invoke.operation = CreateODBEntry or
       rcv.invoke.operation = RemoveODBEntry      { Create- or Remove- }
  then                                             { Directory in progress }
    ident :=
      data_to_identifier(origmesg.invoke.param);  { unpack new ident }
    snd := currmesg;                               { an invoke on host }
    snd.invoke.operation := ModifyODBEntry;
    snd.invoke.objpar := origmesg.invoke.object;

    if rcv.invoke.operation = CreateODBEntry
    then
      new rcv.reply.objpar into dir.list(ident);  { add alias }
      snd.invoke.param := directory_to_data(dir); { pack modified dir }
    else
      remove dir.list(ident);                      { delete alias from dir }

```



```

        snd.invoke.param := directory_to_data(dir);      { pack modified dir }
    end;
    if trstate.replica
    then
        multicast := every;                            { multicast if necessary }
    end;
    trstate.curr := snd;
    pending_ops := pending_ops with
        (into (rcv.transaclbl) := trstate);

    elif trstate.replica                                { reply to final tr invoke }
    then                                                { distributed update }
        trstate.count := trstate.count + 1;
        if trstate.count = size(catmgr_replicas)
        then                                           { got last reply to multicast }
            snd.control := ReplyOp;
            snd.reply := rcv.reply;
            remove pending_ops(rcv.transaclbl);
        else
            out := false;                            { still waiting for more replies; no output }
        end;
    else                                              { local update }
        snd.control := ReplyOp;
        snd.reply := rcv.reply;
        remove pending_ops(rcv.transaclbl);
    end;
end; { reply_handler }

```

{ This procedure handles replies to previous "ReadODBEntry" invocations. Each reply is non-error, and was sent from the local ODB. Lookup transactions are passed on to the lookup_handler. Otherwise, the information in the directory, including whether the directory is replicated and the uid of its primary manager, is unpacked. Every non-lookup transaction is either passed on to the primary manager, or if the local manager is primary, handled in a manner appropriate to each operation. }

```

procedure read_ODB_handler(    rcv: sendmessage;
                               var snd: sendmessage;
                               var out: boolean;
                               var multicast: sendmode;
                               myuid: uid;
                               var pending_ops: pending_list) =

begin
    var trstate: pending_state;
    var origmesg, currmesg: sendmessage;
    var dir: directory;
    var ident: identifier;
    var replicated, i_am_primary: boolean;

```

```

trstate := pending_ops(rcv.transaclbl);
origmesg := trstate.orig;
currmesg := trstate.curr;
snd := origmesg;                                     { default message }

if origmesg.invoke.operation = Lookup
then
    lookup_handler(rcv, snd, out, multicast, pending_ops);
else
    dir := data_to_directory(rcv.reply.param);      { unpack directory info }
    trstate.replica := dir.replica;
    replicated := dir.replica;
    trstate.primary := dir.primary;
    i_am_primary := (dir.primary = myuid) or
                    not replicated;

    if is_locked(origmesg.invoke.object, pending_ops, myuid)
    then                                           { make this tr wait }
        trstate.waiting := true;
        pending_ops := pending_ops with
            (into (rcv.transaclbl) := trstate);

    elif replicated and not i_am_primary
    then                                           { forward to primary mgr }
        snd.control := SendOp;
        snd.receiver := dir.primary;
        pending_ops := pending_ops with
            (into (rcv.transaclbl) := trstate);

    else                                           { not locked and either not }
                                           { replicated or local is primary }

        case origmesg.invoke.operation
        is CreateAlias:
            ident :=
                data_to_identifier(origmesg.invoke.param); { unpack new ident }
            if ident in domain(dir.list)
            then
                snd.control := ReplyOp;                { alias already exists }
                snd.reply.error := true;                { error }
                remove_pending_ops(rcv.transaclbl);
            else
                if replicated
                then
                    multicast := every;                { broadcast }
                end;
                snd := currmesg;
                snd.invoke.operation := ModifyODBEntry;
                new origmesg.invoke.objpar into dir.list(ident); { add alias }

```

```

    snd.invoke.param := directory_to_data(dir); { pack modif dir }
    trstate.curr := snd;
    pending_ops := pending_ops with
        (into (rcv.transaclbl) := trstate);
end;

is RemoveAlias:
    ident :=
        data_to_identifier(origmesg.invoke.param); { unpack new ident }
    if not ident in domain(dir.list)
    then
        snd.control := ReplyOp;    { alias doesnt exist to be removed }
        snd.reply.error := true;    { error }
        remove_pending_ops(rcv.transaclbl);
    else
        if replicated
        then
            multicast := every;    { broadcast }
        end;
        snd := currmesg;
        snd.invoke.operation := ModifyODBEntry;
        remove_dir_list(ident);    { delete alias from dir }
        snd.invoke.param := directory_to_data(dir); { pack modif dir }
        trstate.curr := snd;
        pending_ops := pending_ops with
            (into (rcv.transaclbl) := trstate);
    end;

is CreateDirectory:
    ident :=
        data_to_identifier(origmesg.invoke.param); { unpack new ident }
    if ident in domain(dir.list)
    then
        snd.control := ReplyOp;    { new dir identifier already }
        snd.reply.error := true;    { exists in parent dir }
        remove_pending_ops(rcv.transaclbl);
    else
        snd := currmesg;
        snd.invoke.operation := CreateODBEntry;
        snd.invoke.objlbl := origmesg.invoke.objlbl;
        dir.list := null(alias_list);    { initialize }
        dir.replica := false;    { new directory }
        dir.primary := myuid;    { local catmgr is primary }
        snd.invoke.param := directory_to_data(dir);
        trstate.curr := snd;
        pending_ops := pending_ops with
            (into (rcv.transaclbl) := trstate);
    end;

is RemoveDirectory:

```

```

    ident :=
      data_to_identifier(origmsg.invoke.param); { unpack new ident }
    if not ident in domain(dir.list)
    then
      snd.control := ReplyOp;           { dir identifier doesn't }
      snd.reply.error := true;          { exist to be removed }
      remove_pending_ops(rcv.transaclbl);
    else
      snd := currmsg;
      snd.invoke.operation := RemoveODBEntry;
      trstate.curr := snd;
      pending_ops := pending_ops with
        (into (rcv.transaclbl) := trstate);
    end;

is ReplicateDirectory:
  if dir.replica
  then
    snd.control := ReplyOp;           { dir already replicated }
    snd.reply.error := true;          { error }
    remove_pending_ops(rcv.transaclbl);
  else
    multicast := notlocal;    { request of all other catmgr hosts }
    snd := currmsg;
    snd.invoke.operation := ReplicateODBEntry;
    snd.invoke.param := rcv.reply.param;    { directory content }
    trstate.curr := snd;
    pending_ops := pending_ops with
      (into (rcv.transaclbl) := trstate);
  end;

is DereplicateDirectory:
  if dir.replica
  then
    multicast := notlocal;    { request of all other catmgr hosts }
    snd := currmsg;
    snd.invoke.operation := DereplicateODBEntry;
    trstate.curr := snd;
    pending_ops := pending_ops with
      (into (rcv.transaclbl) := trstate);
  else
    snd.control := ReplyOp;           { dir not replicated }
    snd.reply.error := true;          { error }
    remove_pending_ops(rcv.transaclbl);
  end;
end;
end;
end;
end; { read_odb_handler }

```

```

{ This procedure handles replies to ReadODB invocations
  which result from lookups.
  If the first identifier in the path cannot be translated, abort.
  Otherwise, try to translate remaining path starting from subdirectory. }

procedure lookup_handler(   rcv: sendmessage;
                           var snd: sendmessage;
                           var out: boolean;
                           var multicast: sendmode;
                           var pending_ops: pending_list) =
begin
  var trstate: pending_state;
  var origmesg, currmesg: sendmessage;
  var dir: directory;
  var alist: alias_list;
  var symbolic: path;

  trstate := pending_ops(rcv.transaclbl);
  origmesg := trstate.orig;
  currmesg := trstate.curr;

  symbolic := data_to_path(origmesg.invoke.param);
  { unpack path to be translated }
  dir := data_to_directory(rcv.reply.param); { unpack directory info }
  alist := dir.list;
  if first(symbolic) in domain(alist)
  then
    { found first identifier }
    if nonfirst(symbolic) = null(path)
    then
      { translation complete }
      snd.control := ReplyOp;
      snd.reply.objpar := alist(first(symbolic)); { return uid }
      remove pending_ops(rcv.transaclbl);
    else
      { lookup remaining path }
      snd.control := InvokeOp;
      snd.invoke.object := alist(first(symbolic)); { invoke on subdir }
      snd.invoke.param :=
        path_to_data(nonfirst(symbolic)); { remaining path }
      pending_ops := pending_ops with
        (into (rcv.transaclbl) := trstate);
    end;
  else
    { lookup fails }
    snd.control := ReplyOp;
    snd.reply.error := true;
    remove pending_ops(rcv.transaclbl);
  end;
end; { lookup_handler }

end; { scope }

```

A.5.2 Main Procedure

```
{ Catalog Manager procedures }
```

```
scope sdos = { extending scope sdos }
begin
```

```
{ The main procedure of the catalog manager.
Handles simultaneous, multi-level invocations of
catalog operations.
Its own uid and its host's uid are supplied as arguments.
Inbuf and outbuf are buffers shared with kernel.
```

Each input message, in 'rcvmesg', is examined in conjunction with information in 'pending_ops', to decide values for 'out', which indicates whether any message will be sent out in response, 'sndmesg', the message to be sent, and 'multicast', which indicates whether the response message will be destined for a single receiver, for each host running a replica of the catalog manager, or for each host excepting the local one. This processing of each input is performed by procedure 'handler'.

Once the input message has been given an appropriate response, the catalog manager examines 'pending_ops' to determine if there is any transaction whose progress has been blocked ('waiting') because its object has been 'locked', but which has just been unblocked during the handling of the previous message. Each such transaction will be resumed from the point at which it was suspended.}

```
procedure catalog_manager(  hostuid:  uid;
                           myuid:    uid;
                           var inbuf:  procbuf<input>;
                           var outbuf: procbuf<output>) =
begin
  var pending_ops: pending_list;      { relate pending catalog invocations
                                       to nested invokes }
  var used: transacnoset;              { transaction numbers already used }
  var rcvmesg, sndmesg: sendmessage;
  var trstate: pending_state;
  var trno: transacno;
  var out: boolean;                   { whether current msg will get reply }
  var multicast: sendmode;            { whether reply will be multicast }

  pending_ops := null(pending_list);
  used := null(transacnoset);

  loop
    receive rcvmesg from inbuf;        { get next input }

    out := true;                       { default: will send reply }
```



```

mode: sendmode) =
begin
  var snd: sendmessage := sndmesg;
  var i: integer := 1;
  loop
    if i gt size(catmgr_hosts) then leave; end;
    snd.invoke.object := catmgr_hosts[i];
    if (mode = every or snd.invoke.object ne myuid)
    then
      send sndmesg to outbuf;
    end;
    i := i + 1;
  end;
end;

```

{ Requests of the catalog manager are initiated as 'invokes' or 'sends', which are then identified by a unique transaction number until completed. After receiving an invoke or send, the catalog manager first decides whether the message refers to a new or already-existing transaction. If new, it issues a new transaction number and requests information from the ODB for the object directory. If already-existing, it must take appropriate action, either performing secondary operations on the ODB representations of directories, or in the case of a replicated directory, sending IPC to its primary manager. A primary manager for a replicated directory must synchronize concurrent updates to the directory.

Other procedures which handle messages sent by this catalog manager may issue 'replies'.

The processing of replies depends on the nature of the original invoke or send as well as on the reply's content.

```

Procedure 'reply_handler' does this processing. }

```

[illegible]


```

        init_transaction(rcvmesg, sndmesg, out, multicast,
                        hostuid, myuid, pending_ops, used);
    else
        sndmesg := rcvmesg;                { improper invoke: reply error }
        sndmesg.control := ReplyOp;
        sndmesg.reply.error := true;
    end;

is ReplyOp:                                { got reply }

    if rcvmesg.transaclbl in domain(pending_ops) and
        rcvmesg.level = pending_ops(rcvmesg.transaclbl).orig.level and
        (rcvmesg.sender in catmgr_replicas or
         uidtype(rcvmesg.sender) = HostType)
    then
        reply_handler(rcvmesg, sndmesg,    { reply to pending op }
                      out, multicast, myuid, pending_ops); { from known source }
    else
        out := false;                      { irrelevant reply: ignore }
    end;

is SendOp:                                { got IPC }

    if is_catmgr_op(rcvmesg.invoke.operation) and
        uidtype(rcvmesg.invoke.object) = DirectoryType and
        rcvmesg.sender in catmgr_replicas
    then
        init_transaction(rcvmesg, sndmesg, out, multicast,
                        hostuid, myuid, pending_ops, used); { start new transaction }
    else
        out := false;                      { improper IPC: ignore }
    end;
end; { case }
end; { handler }

{ This procedure initializes an entry in 'pending_ops' for a new transaction. }

procedure init_transaction(    rcv : sendmessage;
                              var snd: sendmessage;
                              var out: boolean;
                              var multicast: sendmode;
                              hostuid: uid;
                              myuid: uid;
                              var pending_ops: pending_list;
                              var used: transacnoset) =
begin
    var trno: transacno;
    var trstate: pending_state;

```

```

snd := kernel_request(rcv, hostuid, ReadODBEntry);
trno := generate_new_transacno(used, rcv.level);
new trno into set used;

snd.transaclbl := trno;
trstate.orig := rcv;
trstate.curr := snd;
trstate.replica := false;
trstate.count := 0;
if is_locked(rcv.invoke.object, pending_ops, myuid)
then
    trstate.waiting := true;
    out := false;
else
    trstate.waiting := false;
end;
new trstate into pending_ops(trno);
end; { init_transaction }

{ This procedure reinitializes an entry in 'pending_ops'
  for a transaction that was previously held waiting.
  No new transaction number need be generated. }

procedure re_init_transaction(    rcv : sendmessage;
                                var snd: sendmessage;
                                var out: boolean;
                                var multicast: sendmode;
                                hostuid: uid;
                                myuid: uid;
                                var pending_ops: pending_list;
                                trno: transacno) =
begin
    var trstate: pending_state;

    snd := kernel_request(rcv, hostuid, ReadODBEntry);
    snd.transaclbl := trno;
    trstate.orig := rcv;
    trstate.curr := snd;
    trstate.replica := false;
    trstate.count := 0;
    if is_locked(rcv.invoke.object, pending_ops, myuid)
    then
        trstate.waiting := true;
        out := false;
    else
        trstate.waiting := false;
    end;
    new trstate into pending_ops(trno);

```

```
end; { re_init_transaction }

end; { scope }
```

A.6 Authentication

A.6.1 The Authentication Manager Specification

```
scope sdos=

{authenticate module}

begin

  type passstr=pending; { type for password strings}
  type passwordtabletype=pending; {all of the password-user information}

  type state=(isop,login2,logout2,unknownreply);

  type todotype=record (oldmess:sendmessage;
                        lasttrans:transacno);

  type todoseqtype=sequence of todotype;
  const readylevel:level :=pending;

  type data=pending;
  const nodata:data :=pending;
  const blankrec:invocation :=pending;
  type CIdatatype = data;
  const noprivelege:CIdatatype :=pending;

  const AuthenticateAs: abst_op := pending;
  const Logout: abst_op := pending;
  const NewPassword: abst_op := pending;

  procedure infologin(thedata:data; var user:principal;
                      var realpassword:passstr) :=pending;
  {unpacks login information from message}

  procedure Checkusersecurity(username:uid; password:passstr;
                              lev:level; passwordtable:passwordtabletype;
```

```

        CIdata:CIdatatype; userlevelisok:boolean)=pending;
{*** This procedure must be CORRECT ***}
{*** This procedure must not give out negative answers too rapidly. ***}
{ Userlevelisok only if password,securitylevel is ok for this user.
  If it is ok also return CIdata}

function extractCIinfo(CIdata:data):CIdatatype =pending;
{returns only that part of the CIdata needed for the kernel.}

function IsTip(principal:uid; mytips:tiplist):boolean=pending;
{check if its from one of this authenticators tips}

Function SetCIpack(bindings:data; origsender:uid; discrlevel:level):data=
                                                    pending;
{combine the data fields into one}

Function Logoutlevel(mess:data):level=pending;
{level of logout -- should be userlow }

function SendertoTrans(origsender:uid):transacno=pending;
{convert senders id to use as a transaction number, for matching response}

procedure SetCImessage( var call:sendmessage;
                        me:uid; Op:abst_op; origsender:uid; bindings:data;
                        lev:level; flag:boolean; discrlevel:level)=
.

begin
  exit call.control=invokeop and call.reply.param=nodata and call.level=lev
    and call.invoke.operation=op;

  call.invoke:=call.invoke with( .operation:=op;
    .param:=SetCIpack(bindings,origsender,discrlevel); .flag:=false);
  call.level:=lev;
  call.reply.param:=nodata;
  call.control:=invokeop;
  call.transaclbl:=SendertoTrans(origsender)
end;

procedure sendreply(var call:sendmessage; lev:level; okreply:boolean) =

begin

exit call.control=replyop and call.invoke.param=nodata and
  call.reply.param=nodata and call.level=lev and
  call.reply.error=okreply and
  call.invoke.operation=call'.invoke.operation;

```

```

call.control:=replyop;
call.invoke.param:=nodata;
call.reply.error:=okreply;
call.reply.param:=nodata;
call.level:=lev;
end;

procedure changepass(user :principal; password: data; lev:level;
    var passwordtable:passwordtabletype; err:boolean) = pending;
{Change the password for user at lev security level. Might want to
    modify this to allow some confirmation}

function purgetodo(todoseq:todoseqtype; l:level):todoseqtype =
begin
    exit result = if todoseq=null(todoseqtype) then
        null(todoseqtype)
    elif l=last(todoseq).oldmess.level then
        purgetodo(nonlast(todoseq),l)
    else purgetodo(nonlast(todoseq),l)<:last(todoseq) fi ;
end;

lemma purgesimplify=
    all l:level, all todoseq:todoseqtype, all count:integer,
    count gt 0 and count le size(todoseq) and
    todoseq[count].oldmess.level=l ->
    purgetodo( todoseq with (seqomit [count]),l)=purgetodo(todoseq,l) ;

procedure getnextmess(var awaitreply:state; call:sendmessage;
    var savedcall:sendmessage; var todoseq:todoseqtype)=

begin

exit ((call.control=invokeop and awaitreply=isop)
    or (call.control ne invokeop and( awaitreply=unknownreply or
        savedcall.level=call.level))) and
    (purgetodo(todoseq',call.level)=purgetodo(todoseq,call.level));
{and the only todoseq changes are at the level of the call}

var count:integer;
var todo:todo:typ;

if call.control=invokeop then
    awaitreply:=isop
elif call.control=replyop then
    awaitreply:=unknownreply; {assume error if not found}
    count:=1;
    loop
        assert call.control=replyop and

```

```

        (purgetodo(todoseq',call.level)=purgetodo(todoseq,call.level)) and
        (awaitreply=unknownreply);
    if count>size(todoseq) then leave end;
    todo:=todoseq(count);
    if call.transac1bl=todo.lasttrans and call.level=todo.oldmess.level
    then
        if call.invoke.operation=resetprocessbindings and
            todo.oldmess.invoke.operation=logout then
            awaitreply:=logout2
        elif call.invoke.operation=setprocessbindings and
            todo.oldmess.invoke.operation=authenticateas then
            awaitreply:=login2
        else {impossible case}
            awaitreply:=unknownreply
        end;
        if awaitreply ne unknownreply then
            savedcall:=todo.oldmess;
            remove todoseq(count);
            leave
        end
    else
        count:=count+1
    end
end
else {a sendop -- an invalid request}
    awaitreply:=unknownreply
end
end;

procedure auth (var port:procbuf <input>; var kport:procbuf <output>;
                me:hostname;
                thepasswordtable:passwordtabletype; mytips:tiplist)=

begin

    var todo:todo:typ;
    var todoseq:todo:seqtype;
    var realpassword,password:passstr;
    var isok,userflag,tiplevelisok,userlevelisok,err,init:boolean;
    var awaitreply:state;
    var count,i:integer;
    var call,savedcall,incall,prfcall:sendmessage;
    var CIdata:data;
    var username:principal;
    var passwordtable:passwordtabletype;
    var tipsecurityleveldata:data;

    passwordtable:=thepasswordtable;

```

```

todoseq:=null(todoseqtype);

loop

  {   Process next request   }

  receive call from port;
  incall:=call; {for proof}
  getnextmess(awaitreply,call,savedcall,todoseq);

  { assertions to help theorem prover. Theorem about output
    is at bottom of loop}

  assert ((call.control=invokeop and awaitreply=isop)
    or (call.control ne invokeop and
      ( awaitreply=unknownreply or savedcall.level=call.level)))
    and (incall.level=call.level);

  case awaitreply
  is isop:
    { * * * * * }
    { *           LOGIN part 1           * }
    { * * * * * }
    if call.invoke.operation=AuthenticateAs then
      {convert message into proper parts}
      infologin(call.invoke.param,username,password);
      {Check user for proper password and retrieve CIdata}
      Checkusersecurity(username,password,call.level,passwordtable,CIdata,
        userlevelisok);
      If userlevelisok and IsTip(call.sender,mytips) then
        savedcall:=call;
        {set up message to change CI bindings}
        SetCImessage(call,me,SetProcessBindings,savedcall.sender,
          extractCIinfo(CIdata),call.level,userflag,call.level);
        send call to kport;
        todo.lasttrans:=call.transaclbl;
        todo.oldmess:=call;
        todoseq:=todoseq <: todo
      else
        {illegal login toss request}
        sendreply(call,call.level,true);
        send call to kport;
      end
    { * * * * * }
    { *           LOGOUT part 1           * }
    { * * * * * }
    elif call.invoke.operation=Logout then
      savedcall:=call;
      todo.oldmess:=call;

```

```

    {send out request to change CI status}
    SetCImessage(call,me,ResetProcessBindings,savedcall.sender,
    noprivelege,call.level,userflag,Logoutlevel(savedcall.invoke.param));
    send call to kport;
    todo.lasttrans:=call.transaclbl;
    todoseq:=todoseq <: todo;
{ * * * * * }
{ *          CHANGE PASSWORD          * }
{ * * * * * }
    elif call.invoke.operation=NewPassword then
        ChangePass(call.sender,call.invoke.param,call.level,
        passwordtable,err);
        sendreply(call,call.level,err);
        send call to kport;
{ * * * * * }
{ *          ILLEGAL REQUEST          * }
{ * * * * * }
    else
        {request is not of the right kind}
        sendreply(call,call.level,true);
        send call to kport;
    end;
{ * * * * * }
{ *          LOGIN part 2              * }
{ * * * * * }
    is login2:
        if call.reply.error then
            { failure in setting discretionay access. This should not
              occur. }
            sendreply(savedcall,savedcall.level,true);
            call:=savedcall; {for proof}
            send call to kport;
        else
            {login succesful .. tell user }
            sendreply(savedcall,savedcall.level,false);
            call:=savedcall; {for proof}
            send call to kport;
        end
{ * * * * * }
{ *          LOGOUT part 2              * }
{ * * * * * }
    is logout2:
        if call.reply.error then
            {unable to reset priveleges. This case should not occur.
              Let the user know logout failed.}
            sendreply(savedcall,savedcall.level,true);
            call:=savedcall; {for proof}
            send call to kport;
        else
            {logout succesful .. tell user }

```



```

        sendreply(savedcall,call.level,false);
        call:=savedcall; {for proof}
        send call to kport;
    end;
{ * * * * * }
{ *          ILLEGAL REQUEST          * }
{ * * * * * }
    else: {it is an unknown reply, or impossible state for
           awaitreply, this shouldn't occur}
           {possibly do nothing or else the following:}
           sendreply(call,call.level,true);
           send call to kport;

end; {case}
assert ((call.control=replyop and call.invoke.param=nodata and
        call.reply.param=nodata ) or
        (call.control=sendop and call.invoke.param=nodata and
        call.reply.param=nodata ) or
        (call.control=invokeop and
        call.reply.param=nodata and
        (call.invoke.operation=Setprocessbindings or
        call.invoke.operation= Resetprocessbindings)) ) and

        (incall.level=call.level);

{ also show that the error field for replies is set correctly}
{ also show that a level 1 call only reads and writes level information
  from the parameters todoseq and passwordtable}
end {outer loop}
end; {procedure}

end; {scope}

```

A.6.2 The Terminal Interface Process Specification

```

scope sdos=

{tip module}

begin
    const readylevel:level :=pending;    {below user levels}

    type statetype=(active,loginin,logout,ready);
    type usermess=pending; {for sending and receiving to user}
    type userbuf= buffer of usermess;
    type lastlogintype = record(transaclbl:transacno; level:level);

```

```

{ for global types see global types description}

procedure UserToSys(ucall:usermess; var call:sendmessage)=pending;
{transform a usermessage into a system message}
{ generate unique transaction number for any login}

procedure SysToUser(call:sendmessage; var ucall:usermess)=pending;
{transform a system message to a user message}

function RequestedLevel(ucall:usermess):level=pending;
{Returns the requested login level}

function Islogout(ucall:usermess):boolean =

begin
  exit assume IsLogout(ucall) -> not Islogin(ucall);
end;

function Islogin(ucall:usermess):boolean =

begin
  exit assume Islogin(ucall) -> not Islogout(ucall);
end;

function Isloginreply(incall:sendmessage; lastlogin:lastlogintype):boolean=
begin
  exit assume Isloginreply(incall,lastlogin) ->
    incall.transaclbl=lastlogin.transaclbl and
    incall.level=lastlogin.level
    and incall.sender=authmgr;
end;

function Islogoutreply(call:sendmessage):boolean=pending;

function IsLoginOk(call:sendmessage):boolean=

begin
  Result:=call.reply.error=false;
end;

function Settransactionlbl:transacno=pending;
{Current proof is based on this just returning a random number. }

procedure tip(var port:procbuf <input>;
              var kport:procbuf <output>;

```

```

        var userport:userbuf<input>;
        var userkport:userbuf <output>;
        me:hostname)=
begin

    var incall,outcall:sendmessage;
    var ucall,pendingcall:usermess;
    var fromuser,senttouser,senttosys:boolean;
    var state,prevstate:statetype;
    var curlevel,prevcurlevel:level;
    var lastlogin,prevlastlogin:lastlogintype;

    state:=ready;
    prevcurlevel:=readylevel;
    prevlastlogin:=lastlogin;
    curlevel:=readylevel;
    prevstate:=ready;
    senttouser:=false;
    senttosys:=false;
    fromuser:=false;

loop

    assert

    If prevstate=loggingout then
        (if not fromuser and IsLogoutReply(incall) then
            state=ready
        else
            state=loggingout fi)
    elif prevstate=active then
        (if fromuser and IsLogout(ucall) then
            state=loggingout
        else
            state=active fi)
    elif prevstate=ready then
        (if fromuser and Islogin(ucall) then
            state=loggingin
        elif fromuser and Islogout(ucall) then
            state=loggingout
        else
            state=ready fi)
    else (prevstate=loggingin and
        if not fromuser and IsLoginreply(incall,lastlogin) then
            (if Isloginok(incall) then
                state=active
            else
                state=ready fi)
        elif fromuser and Islogout(ucall) then
            state=loggingout

```

```

        else
            state=loggingin fi
    fi

    and (lastlogin ne prevlastlogin ->
        (prevstate=ready and state=loggingin and
         lastlogin.level=Requestedlevel(ucall) and
         lastlogin.transaclbl= outcall.transaclbl))

    and
        (state=ready->curlevel=readylevel) and
        (state=loggingin -> curlevel=readylevel) and
        (state=loggingout ->curlevel=readylevel)

    and
        ((curlevel=prevcurlevel) or (state=active and prevstate=loggingin) or
         (state=loggingout and prevstate=active))
    and ((prevstate=loggingin and state=active) ->curlevel=incall.level)

    and ((state=loggingin and prevstate=ready) -> outcall.control=invokeop
        and outcall.invoke.operation=AuthenticateAs
        and outcall.invoke.object=authmgr
        and outcall.level=lastlogin.level
        and lastlogin.level=Requestedlevel(ucall) )

    and (senttouser iff (not fromuser and
        ((prevstate=active and incall.level=curlevel)
         or (prevstate=loggingout and Islogoutreply(incall) )
         or (prevstate=loggingin and Isloginreply(incall,lastlogin)) )))

    and (senttosys iff (fromuser and
        ( (prevstate=active and not Islogin(ucall))
         or (prevstate=ready and Islogin(ucall))
         or (Islogout(ucall)) ) ))

    and [set:state] sub [set: ready,loggingin,loggingout,active];

{remove unneeded hypotheses to help theorem prover}

assert
    ((curlevel=prevcurlevel) or (state=active and prevstate=loggingin) or
     (state=loggingout and prevstate=active))
    and
    (state=ready->curlevel=readylevel) and
    (state=loggingin->curlevel=readylevel)

```

```

    and (state=loggingout -> curlevel=readylevel) and
    [set:state] sub [set: ready,login,logout,active];

await
    on receive incall from port then fromuser:=false;
    on receive ucall from userport then fromuser:=true;
end;

prevcurlevel:=curlevel;
prevlastlogin:=lastlogin;
prevstate:=state;
senttouser:=false;
senttosys:=false;

case state
is active:
    if fromuser and not Islogin(ucall) then
        if IsLogout(ucall) then
            state=loggingout;
            curlevel:=readylevel
        end;
        UserToSys(ucall,outcall);
        outcall.level:=curlevel;
        senttosys:=true;
        send outcall to kport
    elif not fromuser and incall.level=curlevel then
        {possibly change to incall.level <=curlevel}
        SysToUser(incall,ucall);
        senttouser:=true;
        send ucall to userkport
    end
is loggingout:
    if not fromuser and Islogoutreply(incall) then
        SysToUser(incall,ucall);
        senttouser:=true;
        send ucall to userkport;
        state:=ready;
    elif fromuser and IsLogout(ucall) then
        state:=Loggingout;
        UserToSys(ucall,outcall);
        curlevel:=readylevel;
        outcall.level:=readylevel;
        senttosys:=true;
        send outcall to kport
    end
is loginin:
    if not fromuser and Isloginreply(incall,lastlogin) then
        if IsLoginOk(incall) then

```

```

        state:=active;
        curlevel:=incall.level;
        SysToUser(incall,ucall);
        senttouser:=true;
        send ucall to userkport
    else
        state:=ready;
        SysToUser(incall,ucall);
        senttouser:=true;
        send ucall to userkport
    end
elif fromuser and IsLogout(ucall) then
    state:=Loggingout;
    UserToSys(ucall,outcall);
    curlevel:=readylevel;
    outcall.level:=readylevel;
    senttosys:=true;
    send outcall to kport
end
is ready:
if fromuser and Islogin(ucall) then
    state:=loginin;
    UserToSys(ucall,outcall);

{When UsertoSys is built, the following assignment can be included.
 Include the assertion that it is done as the following: }

    outcall:=outcall with (.level:=Requestedlevel(ucall);
        .control:=invokeop;
        .invoke:=outcall.invoke with( .operation:=AuthenticateAs;
        .object:=authmgr));
    outcall.transaclbl:=SetTransactionlbl;
    Lastlogin.transaclbl:=outcall.transaclbl;
    Lastlogin.level:=outcall.level;
    senttosys:=true;
    send outcall to kport
elif fromuser and IsLogout(ucall) then
    state:=Loggingout;
    UserToSys(ucall,outcall);
    outcall.level:=readylevel;
    senttosys:=true;
    send outcall to kport
end
end; {case}
end; {loop}
end; {procedure}
end; {scope}

```

A.7 The Kernel Specification

The kernel specifications will be given as specification of the individual kernel components.

A.7.1 Local Function, Procedure and Type Declarations

```

scope sdos = {extending sdos scope}
begin

{ auxiliary functions used in kernel }

function dominates(l1,l2:level) : boolean = pending;
{ return true if security label l1 dominates l2 }

function uidhost(id: uid) : uid =
begin
    exit (assume uidtype(result) = HostType);
end;
{ return the uid of the host named in the host field of the uid }

function uidtype(id: uid) : abst_type = pending;
{ return the abstract type field of the uid }

function generic_object(at: abst_type) : uid =
begin
    exit (assume uidtype(result) = at);
end;
{ return the uid of the generic object of this type }

function type_object(id: uid) : uid =
begin
    exit (assume result = generic_object(uidtype(id)));
end;
{ a shorthand for composition of the previous two functions }

function procuid(pn: procname; table: procnamemap) : uid =
begin
    exit (assume pn in range(table) -> table(result) = pn);
end;
{ compute the inverse of a process table }

function hostuid(hn: hostname; table: hostnamemap) : uid =
begin
    exit (assume hn in range(table) -> table(result) = hn);
end;
{ compute the inverse of a host table }

function host_label(arg1:uid; table:hostlblmap) : label =
begin

```

```

    exit (assume arg1 in domain(table) -> result = table(arg1));
end;

```

```

end; {scope kernel_scope for auxil}

```

A.7.2 The Message Switch Specification

```

scope sdos = { extending sdos scope }
begin

```

```

procedure message_switch(    thishost    : hostname;
                             var port     : hostbufarr;
                             var procport : procbufarr;
                             var procreqport : procbufarr) =

```

```

{ this procedure receives and routes messages both from
  the rest of the network, and from local processes. }
{messages can be invokes, sends or replies}

```

```

begin
    var SDB: securitydb;
    var ODB: objectdb;
    var ATL,AT: activemap;
    var cache: hostnamemap;
    var kn1: hostlblmap;
    var kn: hostnamemap; { locally-known map from host uid to name }
    var PPT: procnamemap; {locally-known map from process uid to name }
    var PT: proctableentry; {process bindings}
    var p_ops: eventseq; { local invocations not yet completed }
    var msg, msg1: sendmessage;
    var ent: SDBentry;
    var proc: procname;

```

```

    loop

```

```

        await
        on receive msg from port[thishost]      { got message from net }
        then
            case msg.control

```



```

is ReplyOp:                                {Reply to operation invoked by kernel}
if km(msg.receiver) = thishost
then
    { it is a reply to an op invoked by this kernel }
    { that op has to be a locateuid since that is the
      only operation soliciting a reply issued by the
      kernel across the network.
      All other ops are
      issued by some other manager processes}
if msg.invoke.operation = LocateUid
then
    Locator(msg,ATL, kn1, kn, thishost, cache, port);
    {let the locator update the cache and send back info;
      if the msg has the requiried info}

    if not msg.reply.error
    then
        do_pending_ops(msg, p_ops, port);
    elif (msg.reply.errcode = DoesNotExist)
    then
        Reply(thishost,msg,kn1,kn,PPT,port,procport);
    else { ignore the reply because Locator has not
          found the location yet or it is
          duplicate information arriving from different
          host}

    end;
elif msg.invoke.operation = ReadSDBEntry
    {must be replies to the SDB as part of
      replicate operation}
then
    SDBPROC(AT,SDB, msg,kn1,kn,thishost,PPT,port,procport);
    if not (msg.reply.errcode = Waiting)
    then
        Reply(thishost,msg,kn1,kn,PPT,port,procport);
    else { ignore the reply}
    end;
elif msg.invoke.operation = ReadODBEntry
    {must be replies to the ODB as part of
      replicate operation}
then
    ODBProc(AT,ODB,SDB,msg,kn1,kn,thishost,PPT,port,procport);
    if not (msg.reply.errcode = Waiting)
    then
        Reply(thishost,msg,kn1,kn,PPT,port,procport);
    else { ignore the reply}
    end;
end;

else

```

```

    {route the message to the destination process on this host}
    RouteMsg(SDB,AT,thishost,msg,kn1,kn,PPT,port,procport);
end;

```

```

is InvokeOp:
    msg1 := msg;
    msg1.invoke.objpar := type_object(msg.invoke.object);
    msg1.invoke.operation := ReadSDBEntry;
    SDBPROC(AT,SDB, msg1,kn1,kn,thishost, PPT, port, procport);
    msg.level := msg1.level; {very important : resetting the
                                level of the invocation to
                                the reply from the SDB for
                                handling cases of up-operations}
    if msg1.reply.error then { error: no such type locally }
        msg.reply.error := true;
        Reply(thishost,msg,kn1,kn,PPT,port,procport);
    else
        Invoke(p_ops,AT,SDB,ODB,msg,kn1,kn,thishost,PPT,PT,port,
procport);
    end;

```

```

is SendOp:
    RouteMsg(SDB,AT,thishost,msg,kn1,kn,PPT, port, procport);

```

```

end; {case}

```

```

on receive msg from procreqport[proc] { got message from local proc }
then { stamp client id by checking port
      info with the process manager}

```

```

    msg1 := msg;
    msg1.invoke.objpar := transform(proc);
    msg1.invoke.operation := DetermineClientId;
    processmanager(SDB,AT,msg1,PPT,PT, kn1,kn,thishost,port,procport);
    {stamping the client's id}
    msg.sender := msg1.reply.objpar;
    { verifying/rectifying the client's level }
    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;
    msg1.invoke.objpar := msg.sender;

    SDBPROC(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);

```

```

if not msg1.reply.error then
  ent := unpack(msg.reply.param);
  if not ent.mls then           { stamp label if single-level }
    msg.level := ent.label;

    {if mls entity then the level of the message must be in the
     set of permissible levels..else error }
  elif not msg.level in [ent.label]
  then
    msg.reply.error := true;
    msg.reply.errcode := IncorrectLevel;
    Reply(thishost,msg, kn1,kn, PPT, port, procport);
  end;

  case msg.control

  is ReplyOp:
    RouteMsg(SDB,AT,thishost,msg,kn1,kn,PPT, port, procport);

  is SendOp:
    RouteMsg(SDB,AT,thishost,msg,kn1,kn,PPT, port, procport);

  is InvokeOp:
    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;

    SDBPROC(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
    if msg1.reply.error
    then
      { error: no such object locally
       hence need to locate across net}
      Locator(msg,ATL, kn1, kn, thishost, cache, port);
      start_Active_Table(msg,thishost,ATL);
      if msg1.reply.error {no information in local cache}
      then
        p_ops := p_ops <: msg;
      else
        Invoke(p_ops,AT,SDB, ODB, msg, kn1, kn,
              thishost, PPT,PT,port,procport);
      end;
    else
      Invoke(p_ops,AT, SDB, ODB, msg, kn1, kn,
            thishost, PPT, PT,port, procport);
    end;
  end; {case}
else
  msg1.reply := msg1.reply;
  Reply(thishost, msg, kn1, kn, PPT, port, procport);
end; {if}
end; {await}
end; {loop}

```

```
end; {message switch}
```

```
procedure Invoke(var p_ops: eventseq;
                 var AT: activemap;
                 var SDB: securitydb;
                 var ODB: objectdb;
                 var msg: sendmessage;
                 kn1: hosttblmap;
                 kn: hostnamemap;
                 thishost: hostname;
                 var PPT: proctablemap;
                 var PT: proctableentry;
                 var port: hostbufarr;
                 var procport: procbufarr) =
```

```
{ this procedure carries out an invoke once it has been
  determined that the object type is defined locally.
  a manager of the object's type is started, if necessary.
  fill in any msg.reply fields appropriate, and reply if possible. }
```

```
begin
```

```
  var ent: SDBentry;
  var manager: uid;
  var msg1,msg2: sendmessage;
```

```
  if msg.operateup_enabled
```

```
  then
```

```
    msg1 := msg;
    msg1.reply.error := false;
    msg1.operateup_enabled := false;
    Reply(thishost,msg1, kn1, kn, PPT, port, procport);
```

```
  end;
```

```
  case uidtype(msg.invoke.object)
```

```
  is HostType: { route to "host manager" }
```

```
    if kn(msg.invoke.object) = thishost {Op invoked on kernel}
```

```
    then
```

```
      case msg.invoke.operation
```

```
      is LocateUid:
```

```
        msg1 := msg;
        msg1.invoke.operation := ReadSDBEntry;
        SDBPROC(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
        if not msg1.reply.error
        then
          Reply(thishost,msg, kn1, kn, PPT, port, procport);
        end;
```

```

is CreateODBEntry, WriteODBEntry, ReadODBEntry, RemoveODBEntry,
  ReplicateODBEntry, DereplicateODBEntry, CopyODBEntry:

  ODBProc(AT, ODB, SDB, msg, kn1, kn, thishost, PPT, port, procport);
  if (msg.reply.errcode eq Waiting)
  then
    p_ops := p_ops <: msg;
  else
    Reply(thishost, msg, kn1, kn, PPT, port, procport);
  end;

is CreateSDBEntry, ModifySDBEntry, ReadSDBEntry, RemoveSDBEntry,
  ReplicateSDBEntry, DereplicateSDBEntry, IncrementReplicaNo,
  DecrementReplicaNo:

  SDBPROC(AT, SDB, msg, kn1, kn, thishost, PPT, port, procport);
  if (msg.reply.errcode eq Waiting)
  then
    p_ops := p_ops <: msg;
  else
    Reply(thishost, msg, kn1, kn, PPT, port, procport);
  end;

else: {case}
  msg.reply.error := true;
  msg.reply.errcode := InvalidOp;
  Reply(thishost, msg, kn1, kn, PPT, port, procport);
end;
else
  {msg should not have come to this kernel ...so ignore}
end;

is ProcessType:                                     { route to process manager }

  processmanager(SDB, AT, msg, PPT, PT, kn1, kn, thishost, port, procport);
  Reply(thishost, msg, kn1, kn, PPT, port, procport);

else:                                               { route to manager outside kernel }
  msg1 := msg;
  msg1.invoke.objpar := type_object(msg.invoke.object);
  msg1.invoke.operation := ReadSDBEntry;
  SDBPROC(AT, SDB, msg1, kn1, kn, thishost, PPT, port, procport);
  if not msg1.reply.error
    {there should not be an error because there
     has been an kernel check to determine
     that the type is supported on this machine}

```

```

then
  ent := unpack(msg.reply.param);
  if ent.mls and
    ent.active_mgrs ne null(labeltable)
  then
    manager := mgrchoose(ent.active_mgrs);
  else
    if not ent.mls and
      msg.level in range(ent.active_mgrs)
    then
      manager := mgruid(msg.level, ent.active_mgrs);
    else
      msg1.invoke.objpar := ent.exe_file;
      msg1.invoke.objlbl := msg.level;
      msg1.level := msg.level;
      msg1.invoke.operation := CreateProc;
      processmanager(SDB,AT,msg1,PPT,PT,kni,kn,
                    thishost,port,procport);

      if not msg1.reply.error
      then
        manager := msg1.reply.objpar;
        ent.active_mgrs := ent.active_mgrs
          with (into (manager) := msg1.invoke.objlbl);
        msg1.invoke.objpar := type_object(msg.invoke.objpar);
        msg1.invoke.operation := ModifySDBEntry;
        SDBPROC(AT,SDB,msg1,kni,kn,thishost,PPT,port,procport);
      end;
    end;
  end;
end;
send msg to procport[PPT(manager)];
end;
end; {case}
end; {procedure}

```

```

procedure RouteMsg(var SDB: securitydb;
  var AT: activemap;
  thishost: hostname;
  var msg: sendmessage;
  kni: hostlblmap;
  kn: hostnamemap;
  PPT: procnamemap;
  var port: hostbufarr;
  var procport: procbufarr) =

```

```

{route msg as a reply, with destination determined by 'client'.
 client is remote if host uid field is foreign. }

```

```

begin
  var msg1: sendmessage;

  if msg.receiver in domain(PPT)
  then
    msg1 := msg;
    msg1.invoke.objpar := msg.receiver;
    msg1.invoke.operation := ReadSDBEntry;
    SDBPROC(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
    {route msg to destination process}
    msg.level := msg1.level;
    if equals(msg1.reply.objlbl, msg.level)
    then
      send msg to procport[PPT(msg.receiver)];
    else
      msg.reply.error := true;
      Reply(thishost,msg,kn1,kn,PPT,port,procport);
    end;
  else
    msg.reply.error := true;
    Reply(thishost,msg,kn1,kn,PPT,port,procport);
  end;
end; {procedure}

```

```

procedure Locator(var msg: sendmessage;
  var ATL: activemap;
    kn1: hostlblmap;
    kn: hostnamemap;
    thishost: hostname;
    var cache: hostnamemap;
    var port: hostbufarr) =
begin
  var msg1: sendmessage;

  if msg.control = InvokeOp and
    msg.invoke.objpar in domain(cache)
  then
    msg.invoke.object := cache(msg.invoke.objpar);
    msg.reply.error := false;
  else
    msg.invoke.operation := LocateUid;
    Broadcast(msg, thishost, kn1, kn, port);
    msg.reply.error := true;
  end;
end;

```

```

    msg.reply.errcode := Waiting;
end;

if msg.control = ReplyOp
then
    update_Active_Table(msg, thishost, ATL);
    if msg.reply.error = false and
        not msg.invoke.objpar in domain(cache)
        {this handling of the cache harbours on the
         assumption that clients do not migrate.}
    then
        cache := cache with (into (msg.invoke.objpar) := msg.senderhost);
        msg.reply.error := false;
    elif not check_sufficient(msg, thishost, ATL)
    then
        msg.reply.error := true;
        msg.reply.errcode := Waiting;
    else
        msg.reply.error := true;
        msg.reply.errcode := DoesNotExist;
    end;
end;

end; {procedure}

procedure do_pending_ops(msg: sendmessage;
                        var p_ops: eventseq;
                        var port: hostbufarr) =

{ complete any operation which is pending, waiting for the
  location info contained in 'msg'. }

begin
    var count: integer;
    count := size(p_ops);
    loop
        if count le 0 then leave; end;
        if p_ops[count].invoke.object = msg.invoke.objpar
        then
            send p_ops[count] to port[msg.sender];
            if count ne size(p_ops)
            then p_ops[count] := last(p_ops); end;
            p_ops := nonlast(p_ops);      { remove op from list }
        end;
    end;
end;

```



```

procedure Reply(thishost: hostname;
                var msg: sendmessage;
                kn1: hosttblmap;
                kn: hostnamemap;
                PPT: procnamemap;
                var port: hostbufarr;
                var procport: procbufarr) =

{route msg as a reply, with destination determined by 'client'.
 client is remote if host uid field is foreign. }

begin
var msg1: sendmessage;

msg.control := ReplyOp;

if not msg.operateup_enabled
then
  if msg.senderhost eq thishost    {replying to a local process}
  then
    if msg.sender in domain(PPT)
    then
      msg1 := msg;
      msg1.sender := msg.receiver;
      msg1.receiver := msg.sender;
      send msg1 to procport[PPT(msg.sender)];
    end;
  else                                {replying to another host}
    msg1 := msg;
    {check levels to see if reomote host can handle the message}
    {under current scheme the level of the reply is the level of
     the invoke, therefore check not mandatory}
    if dominates(kn1(msg.senderhost),msg.level)
    then
      msg1.senderhost := thishost;
      msg1.sender := msg.receiver;
      msg1.receiver := msg.sender;
      send msg1 to port[kn(msg.senderhost)];
    end;
  end;
else {should not reply to client..the kernel has already ack. that action

```

```

                                is being initiated on the client's behalf}
end;

end; {procedure}

procedure Broadcast(var msg: sendmessage;
                    thishost: hostname;
                    kn1: hosttblmap;
                    kn: hostnamemap;
                    var port: hostbufarr) =

begin
{ invoke the operation on all other known hosts that can handle the requested
  level of invocation}

    var h: hostname;

    h := minhost;

    loop
        if h ne thishost and
        h in range(kn)
        then
            msg.invoke.object := hostuid(h, kn);
            {determine the label of host h...public information}
            if dominates(meet(host_label(msg.invoke.object,kn1)),msg.level)
            then
                msg.invoke.object := hostuid(h,kn);
                msg.sender := thishost;
                msg.receiver := h;
                send msg to port[h];
            end;
        end;
        if h = maxhost then leave; end;
        h := h + 1;
    end; {loop}
end; {procedure}

end; {sdos scope for message switch}

```

A.7.3 The Security Database Specification

```
scope sdos =      { extending sdos scope }
```



```

    msg.reply.objpar := msg.invoke.objpar;
    SDB := SDB with (into (msg.invoke.objpar) :=
                                unpack(msg.invoke.param));
  else
    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
  end;
else
  msg.reply.error := true;
  msg.reply.errcode := NoPermission;
end;

is ModifySDBEntry: {if operateup bit is set then :
                    level of the object dominates level of the message
                    else level of object equals level of the message}
                    {also enforces the configuration policy that the system
                    manager and authentication manager are the only entities
                    that can successfully invoke this operation}

if not msg.operateup_enabled
then
  if msg.invoke.objpar in domain(SDB) and
    equals(SDB(msg.invoke.objpar).label, msg.level) and
    msg.sender in [set: system_manager, authmgr]
  then
    msg.reply.error := false;
    SDB := SDB with (into (msg.invoke.objpar) :=
                                unpack(msg.invoke.param));

    Broadcast(msg, thishost, kn1, kn, port);
  elif not msg.sender in [set: system_manager, authmgr] and
    dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
  then
    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
  else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
  end;

else {if operateup_enabled}
  if msg.invoke.objpar in domain(SDB) and
    dominates(meet(SDB(msg.invoke.objpar).label), msg.level) and
    msg.sender in [set: system_manager, authmgr]
  then
    msg.reply.error := false;
    SDB := SDB with (into (msg.invoke.objpar) :=
                                unpack(msg.invoke.param));

    Broadcast(msg, thishost, kn1, kn, port);
    msg.level := join(msg.level, SDB(msg.invoke.objpar).label);
  elif not msg.sender in [set: system_manager, authmgr] and

```

```

        dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
    then
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
        msg.level := join(msg.level, SDB(msg.invoke.objpar).label);
    else
        msg.reply.error := true;
        msg.level := sys_hi;
        msg.reply.errcode := DoesNotExist;
    end;
end;

```

is ReadSDBEntry:

```

        {if operateup bit is set then
            level of the message dominates the level of the object
        else level of the message should equal the level of the object.}

    if not msg.operateup_enabled
    then
        if msg.invoke.objpar in domain(SDB) and
            dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
        then
            msg.reply.error := false;
            msg.reply.objlbl := SDB(msg.invoke.objpar).label;
            msg.reply.param := pack(SDB(msg.invoke.objpar));
        else
            msg.reply.error := true;
            msg.reply.errcode := DoesNotExist;
        end;
    else {if operateup_enabled}
        if msg.invoke.objpar in domain(SDB)
        then
            msg.reply.error := false;
            msg.reply.objlbl := SDB(msg.invoke.objpar).label;
            msg.reply.param := pack(SDB(msg.invoke.objpar));
            msg.level := join(msg.level, SDB(msg.invoke.objpar).label);
        else
            msg.reply.error := true;
            msg.level := sys_hi;
            msg.reply.errcode := DoesNotExist;
        end;
    end;
end;

```

is LocateUid: {will determine existence of the object and/or the
manager.

```

        if the operateup switch is set then
            level of the message dominates level of the object
        else
            level of the object equals level of the message.}

```

```

if not msg.operateup_enabled
then
  if msg.invoke.objpar in domain(SDB) and
    dominates(msg.level,meet(SDB(msg.invoke.objpar).label)) and
    can_handle(SDB(type_object(msg.invoke.objpar)),msg)
  then
    msg.reply.error := false;
    msg.reply.param := code_param(ObjectandManager);
  elif msg.invoke.objpar in domain(SDB) and
    dominates(msg.level,meet(SDB(msg.invoke.objpar).label))
  then
    msg.reply.error := false;
    msg.reply.param := code_param(ObjectOnly);
  elif can_handle(SDB(type_object(msg.invoke.objpar)),msg)
  then
    msg.reply.error := false;
    msg.reply.param := code_param(ManagerOnly);
  else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
  end;
else {if operateup_enabled}

  if msg.invoke.objpar in domain(SDB) and
    can_handle(SDB(type_object(msg.invoke.objpar)),msg)
  then
    msg.reply.error := false;
    msg.level := join(msg.level,SDB(msg.invoke.objpar).label);
    msg.reply.param := code_param(ObjectandManager);
  elif msg.invoke.objpar in domain(SDB)
  then
    msg.reply.error := false;
    msg.level := join(msg.level,SDB(msg.invoke.objpar).label);
    msg.reply.param := code_param(ObjectOnly);
  elif can_handle(SDB(type_object(msg.invoke.objpar)),msg)
  then
    msg.reply.error := false;
    msg.level := join(msg.level,SDB(msg.invoke.objpar).label);
    msg.reply.param := code_param(ManagerOnly);
  else
    msg.reply.error := true;
    msg.level := sys_hi;
    msg.reply.errcode := DoesNotExist;
  end;
end;
end;

```

is RemoveSDBEntry: {if operateup switch is set then the level of the

```

        object dominates the level of the call.
    else
        level of the object equals the level of the call}
    {note this will result in removal of this
     entry from all ODB's}

if not msg.operateup_enabled
then
    if msg.invoke.objpar in domain(SDB) and
        equals(SDB(msg.invoke.objpar).label,msg.level)
    then
        msg.reply.error := false;
        SDB := SDB with (mapomit (msg.invoke.objpar));
        msg1 := msg;
        msg1.invoke.operation := RemoveSDBEntry;
        Broadcast(msg1,thishost,kn1, kn,port);
    elif (msg.invoke.objpar in domain(SDB)) and
        dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
    then
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    else
        msg.reply.error := true;
        msg.reply.errcode := DoesNotExist;
    end;
else {if operateup_enabled}
    if msg.invoke.objpar in domain(SDB) and
        dominates(meet(SDB(msg.invoke.objpar).label),msg.level)
    then
        msg.reply.error := false;
        SDB := SDB with (mapomit (msg.invoke.objpar));
    elif (msg.invoke.objpar in domain(SDB)) and
        dominates(msg.level,meet(SDB(msg.invoke.objpar).label))
    then
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    else
        msg.reply.error := true;
        msg.level := sys_hi;
        msg.reply.errcode := DoesNotExist;
    end;
end;

is ReplicateSDBEntry: {level of the object must
                       equal the level of the request}

if not msg.invoke.objpar in domain(SDB)
    {object does not exist locally}
then

```

```

    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;
    Broadcast(msg1, thishost, kn1, kn, port);
    msg.reply.errcode := Waiting;
    start_Active_Table(msg, thishost, AT); {make an entry to look
                                         out for replies to this broadcast}
  elif msg.invoke.objpar in domain(SDB) and
    dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
  then
    msg.reply.error := true;
    msg.reply.errcode := AlreadyExists;
  else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
  end;
end;

```

is DeReplicateSDBEntry: {level of the object must equal
level of the request}

```

  if msg.invoke.objpar in domain(SDB) and {object exists locally}
    equals(SDB(msg.invoke.objpar).label, msg.level) and
    SDB(msg.invoke.objpar).replicas ge 1
  then
    msg1 := msg;
    msg1.invoke.operation := DecrementReplicaNo;
    Broadcast(msg1, thishost, kn1, kn, port);
    msg.reply.error := false;
  elif (msg.invoke.objpar in domain(SDB)) and
    dominates(msg.level, meet(SDB(msg.invoke.objpar).label))
  then
    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
  else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
  end;
end;

```

is IncrementReplicaNo:

```

  if msg.invoke.param in domain(SDB) and
    equals(ent.label, msg.level)
  then
    ent := unpack(msg.invoke.param);
    ent.replicas := ent.replicas + 1;
    SDB := SDB with (into (msg.invoke.param) := ent);
    msg.reply.error := false;
  else

```



```

    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
end;

is DecrementReplicaNo:

    if msg.invoke.param in domain(SDB) and
       equals(ent.label, msg.level)
    then
        ent := unpack(msg.invoke.param);
        ent.replicas := ent.replicas - 1;
        SDB := SDB with (into (msg.invoke.param) := ent);
        msg.reply.error := false;
    else
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    end;

else: {case}
    msg.reply.error := true;
    msg.reply.errcode := InvalidOp;

end: {case}

is ReplyOp: {valid replies are from ReadSDBEntry operations
              invoked as part of ReplicateSDBEntry request}

case msg.invoke.operation

is ReadSDBEntry:
    {make an entry to reflect that a host
     has replied to a particular broadcast msg}
    update_Active_Table(msg, thishost, AT);

    if not msg.reply.error and
       not msg.invoke.objpar in domain(SDB)
    then
        ent := unpack(msg.reply.param);
        if equals(ent.label, msg.level)
        then
            msg.reply.error := false;
            msg1 := msg;
            msg1.invoke.operation := IncrementReplicaNo;
            Broadcast(msg1, thishost, kn1, kn, port);
            ent.replicas := ent.replicas + 1;
            SDB := SDB with (into (msg.invoke.objpar) := ent);

```

```

        elif dominates(msg.level,meet(ent.label))
        then
            msg.reply.error := true;
            msg.reply.errcode := NoPermission;
        end;
    elif not check_sufficient(msg,thishost,AT) {have all hosts
                                                responded}
    then
        msg.reply.errcode := Waiting;
    else
        msg.reply.error := true;
        msg.reply.errcode := DoesNotExist;
    end;

else : {ignore the reply}

end; {case ReplyOP}

end; {case control}

end; {procedure}

end; {scope sdos for sdb }

```

A.7.4 The Object Database Specification

```
scope sdos =      { extending sdos scope }
```

{note : The client has the option to set the operateup bit to invoke actions on objects above his level. The ODB relies on the SDB to supply the level of the object. In this process of ReadSDBEntry, the level of the reply from the SDB may be higher than the level of the request to the SDB, because the operateup bit was set. The ODB then resets the level of the request it got from the object-manager to this new level. So when a reference is made to level of invoke in the documentation, we mean the this new modified level.}

{note : .invoke.objpar has the uid of the object on which ODB and SDB ops are to be carried out}

```

begin

{ Object database }

procedure ODBProc(var AT: activemap;

```

```

        var ODB: objectdb;
        var SDB: securitydb;
        var msg: sendmessage;
        kn1: hostlblmap;
        kn: hostnamemap;
        thishost: hostname;
        PPT: procnamemap;
        var port: hostbufarr;
        var procport: procbufarr) =
begin
var type_error, object_error: boolean;
var type_entry, object_entry: sdbentry;
var msg1: sendmessage;
var info: ODBObject;
var ent: SDBentry;

case msg.control

is InvokeOp:

    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;

    msg1.invoke.objpar := type_object(msg.invoke.objpar);
    SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
    type_entry := unpack(msg1.reply.param);

    if msg.sender in domain(type_entry.active_mgrs)
        {is sender actually a manager for that type}
    then
        {check to see if the particular instance is
        supported on the local host}
        msg1.invoke.objpar := msg.invoke.objpar;
        SDBProc(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
        object_error := msg1.reply.error;
        object_entry := unpack(msg1.reply.param);
        msg.level := msg1.level;    {very important ...the level of the
                                     invoke is now being reset to the
                                     level of the reply from ReadSDB :
                                     for up operations this will equal
                                     the level of the object}

        case msg.invoke.operation

        is CreateODBEntry: {if client is a manager of the specified type, and
                             label of new object dominates level of invoke then
                             create a new object of the manager's type and
                             return its uid. }

            if dominates(meet(msg.invoke.objlbl), msg.level)

```

```

then
  object_entry := initial(SDBentry);
  object_entry.label := msg.invoke.objlbl;
  msg1.level := msg.level;
  msg1.invoke.operation := CreateSDBEntry;
  msg1.invoke.objpar := type_object(msg.invoke.objpar);
  msg1.invoke.param := pack(object_entry);
  SDBProc(AT,SDB, msg1,kn1, kn,thishost,PPT,port,procport);
  if not msg1.reply.error
  then
    msg.reply := msg1.reply;
    ODB := ODB with (into (msg.reply.objpar) :=
                                                                initial(ODBobject));
  else
    msg.reply := msg1.reply;
  end;
else
  msg.reply.error := true;
  msg.reply.errcode := NoPermission;
end;

is WriteODBEntry:
  {if client is a manager for the type of this object, and
   level of object equals level of invoke deposit the data.}

  if not object_error
  then
    if equals(object_entry.label, msg.level)
    then
      msg.reply.error := false;
      ODB := ODB with (into (msg.invoke.objpar) :=
                                                                msg.invoke.param);
    else
      msg.reply.error := true;
      msg.reply.errcode := NoPermission;
    end;
  else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
  end;

is ReadODBEntry:
  {if client is a manager for the type of this object, and
   level of invoke dominates level of object, return the data.}

  if not object_error

```

```

then
  msg.reply.error := false;
  msg.reply.param := ODB(msg.invoke.objpar);
else
  msg.reply.error := true;
  msg.reply.errcode := DoesNotExist;
end;

```

is RemoveODBEntry:

{if client is a manager for the type of this object and level
of object equals level of the invoke, remove the object.}

{Note: this will remove all occurrences}

```

if not object_error
then
  if equals(object_entry.label, msg.level)
  then
    msg.reply.error := false;
    ODB := ODB with (mapomit (msg.invoke.objpar));
    msg1 := msg;
    msg1.invoke.operation := RemoveSDBEntry;
    SDBProc(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
    Broadcast(msg1,thishost,kn1, kn,port);
    msg1.invoke.operation := RemoveODBEntry;
    Broadcast(msg1,thishost,kn1, kn,port);
  else
    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
  end;
else
  msg.reply.error := true;
  msg.reply.errcode := DoesNotExist;
end;

```

is CopyODBEntry: {client can copy object A to B if level of B
dominates level of A}

```

if not object_error
then
  msg1 := msg;
  msg1.invoke.operation := ReadSDBEntry;

  msg1.invoke.objpar := msg.invoke.objpar1;
  SDBProc(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
  ent := unpack(msg1.reply.param);

```

```

        if not msg1.reply.error and
            type_object(msg.invoke.objpar) =
                type_object(msg.invoke.objpar1)
        then
            msg1.invoke.objpar := msg.invoke.objpar1;
            SDBProc(AT,SDB,msg1.kn1,kn,thishost,PPT,port,procport);
            ent := unpack(msg1.reply.param);
        if msg.operateup_enabled
        then
            if dominates(meet(ent.label), msg.level)
            then
                ODB := ODB with (into (msg.invoke.objpar1) :=
                    ODB(msg.invoke.objpar));
            msg.reply.error := false;
            msg.level := join(msg.level,ent.label);
            else
                msg.reply.error := true;
                msg.reply.errcode := NoPermission;
                msg.level := join(msg.level,ent.label);
            end;
        else {if msg.operateup_enabled}
            if equals(ent.label, msg.level)
            then
                ODB := ODB with (into (msg.invoke.objpar1) :=
                    ODB(msg.invoke.objpar));
                msg.reply.error := false;
            elif dominates(meet(ent.label), msg.level)
            then
                msg.reply.error := true;
                msg.reply.errcode := NoPermission;
            else
                msg.reply.error := true;
                msg.reply.errcode := DoesNotExist;
            end;
        end;
    else
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    end;
else
    msg.reply.error := true;
    msg.reply.errcode := DoesNotExist;
end;

is ReplicateODBEntry: {create a local copy of an object}
{level of the invoke must equal level of
the object}

```

```

if not object_error
then
  msg1 := msg;
  msg1.invoke.operation := ReplicateSDBEntry;
  SDBProc(AT,SDB,msg1, kn1, kn, thishost, PPT, port, procport);
  msg1.reply := msg1.reply;
  if not msg1.reply.error
  then
    msg1 := msg;
    msg1.invoke.operation := ReadODBEntry;
    Broadcast(msg1, thishost, kn1, kn, port);
    msg1.reply.errcode := Waiting;
  end;
else
  msg1.reply.error := true;
  msg1.reply.errcode := DoesNotExist;
end;

is DeReplicateODBEntry:
  {force a DeReplicateSDBEntry..if that succeeds then
   delete local ODBEntry. Else echo the error from
   DeReplicateSDBEntry operation}

if not object_error
then
  msg1 := msg;
  msg1.invoke.operation := DeReplicateSDBEntry;
  SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
  msg1.reply := msg1.reply;
  if not msg1.reply.error
  then
    ODB := ODB with (mapomit (info));
  end;
else
  msg1.reply.error := true;
  msg1.reply.errcode := DoesNotExist;
end;

end; {case}

else {if not sender in type_managers}
  msg1.reply.error := true;
  msg1.reply.errcode := NoPermission;
end; {if}

```

```

is ReplyOp:      {valid replies are only for the ReadODBEntry operation
                  invoked as part of the ReplicateODBEntry operation}

case msg.invoke.operation

is ReadODBEntry:

    {instead of maintaing elaborate queues of
     requests sent and recieved, we have elected
     to do a ReadSDBEntry operation again to
     validate the operation of writing into the
     object}

    if not msg.reply.error and uidtype(msg.sender) = HostType and
       not msg.invoke.objpar in domain(ODB)
    then
        msg1 := msg;
        msg1.invoke.operation := ReadSDBEntry;
        SDBProc(AT,SDB,msg1,kn1,kn,thishost,PPT,port,procport);
        if not msg1.reply.error and
           equals(msg1.reply.objlbl,msg1.level)
        then
            info := msg.reply.param;
            ODB := ODB with (into (msg.invoke.objpar) := info);
            msg.reply.error := false;
        end;
    else
        msg.reply.errcode := Waiting; {informing message switch
                                       that the Replicate Operation
                                       is still not complete}
    end;

end; {case under ReplyOp}

end; {case}

end; {procedure}

end; {scope sdos for odb }

```


A.7.5 The Process Manager Specification

```

scope sdos =      { extending sdos scope }

begin

procedure ProcessManager(var SDB: securitydb;
                        var AT: activemap;
                        var msg: sendmessage;
                        var PPT: proctnamemap;
                        var PT: proctableentry;
                        kn1: hosttblmap;
                        kn: hostnamemap;
                        thishost : hostname;
                        var port : hostbufarr;
                        var procport : procbufarr) =

begin
var ent,ent1: SDBentry;
var msg1: sendmessage;
var name1: procname;

case msg.invoke.operation

    is CreateProc: {level at which process is to be created dominates the
                    level of the msg and the level of the file that
                    contains the process code.}
                    {the level of the new process is in .invoke.objlbl and
                    the uid of the file that contains process code is in
                    .invoke.objpar1 and the uid of the gereric object of
                    the type is in msg.invoke.objpar}

                    msg1 := msg;
                    msg1.invoke.operation := ReadSDBEntry;
                    msg1.invoke.objpar := msg.invoke.objpar1;
                    SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
                                {checking the level of the source file}
                    ent := unpack(msg1.reply.param);

                    if not msg1.reply.error and
                        dominates(meet(msg.invoke.objlbl),msg.level) and
                        dominates(meet(msg.invoke.objlbl),ent.label)
                    then
                        msg1 := msg;
                        msg1.invoke.operation := CreateSDBEntry;
                        ent.label := msg.invoke.objlbl;
                        ent.mls := false;

```

```

msg1.invoke.param := pack(ent);
SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
if not msg1.reply.error
then
    msg1.invoke.objpar := msg1.invoke.objpar; {get hold of the new
                                                uid}

    msg1 := msg;
    msg1.level := host_lo;
    update_PPT(add, msg1, PPT);
                                {make an entry into the process port table.
                                This information is Host_lo}

    if not msg1.reply.error
    then
        msg1 := msg;
        msg1.level := meet(msg.invoke.objlbl);
        update_PT(add, msg, PT);
                                {make an entry about process bindings. This
                                information is at the level of the new
                                process}

    end;
end;

elif (msg1.reply.error or
      not dominates(meet(msg.invoke.objlbl),ent.label)) and
      dominates(meet(msg.invoke.objlbl),msg.level)
then
    msg.reply.error := false;
else
    msg.reply.error := true;
    msg.reply.errcode := NoPermission;
end;

is RemoveProc: {level of the process dominates level of the msg}

msg1 := msg;
msg1.invoke.operation := ReadSDBEntry;
SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
ent := unpack(msg1.reply.param);
msg1.level := msg1.level;
                                {important: resetting the level of the invoke
                                to the return from ReadSDB..good way of
                                handling operateups}

if (not msg1.reply.error) and equals(ent.label, msg1.level)
then
    msg.reply.error := false;
    update_PPT(delete,msg,PPT);
    update_PT(delete,msg,PT);
else
    msg.reply.error := true;

```

```

        msg.reply.errcode := DoesNotExist;
    end;

is ShowProcessBindings:

    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;
    msg1.invoke.objpar := type_object(msg.invoke.objpar);
    SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
    ent1 := unpack(msg1.reply.param);

    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;
    SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
    ent := unpack(msg1.reply.param);

    msg.level := msg1.level; {important : restting the level of the
                               invoke to the return from ReadSDB..
                               good way of handling operateups}

    if (not msg1.reply.error) and equals(ent.label, msg.level) and
        msg.sender in domain(ent1.active_mgrs)
    then
        msg.reply.error := false;
    else
        msg.reply.error := true;
        msg.reply.errcode := DoesNotExist;
    end;

is SetProcessBindings: {System Manager and Authenication Manager can
                        set bindings. Level of the invoke has to be
                        dominated by the level of the object}

    msg1 := msg;
    msg1.invoke.operation := ReadSDBEntry;
    SDBProc(AT,SDB, msg1, kn1, kn, thishost, PPT, port, procport);
    ent1 := unpack(msg1.reply.param);
    if dominates(msg.level, meet(ent.label)) and
        msg.sender in [set: authmgr, system_manager]
    then
        update_PT(add,msg,PT);
    elif dominates(msg.level, meet(ent.label))
    then
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    else
        msg.reply.error := true;
        msg.reply.errcode := DoesNotExist;

```

```

    end;

    is ChangeActiveCCI: {client can change his own CCI}

        if msg.sender = msg.invoke.objpar
        then
            update_PT(add,msg,PT)
        else
            msg.reply.error := true;
            msg.reply.errcode := NoPermission;
        end;

    is DetermineClientId: {no security checks needed because this is
                           public info}

        name1 := un_transform(msg.invoke.objpar);
        msg.reply.objpar := procuid(name1,PPT);
        msg.reply.error := false;

end; {case}
end; {procedure}

procedure update_PPT(op: abst_op;
                    var msg: sendmessage; var PPT:procnamemap) =
    {updating the process-port-table . Since the port info is
     host_lo the update has to be at host_lo}
begin
    var temp: procname;

    if msg.level = host_lo
    then
        msg.reply.error := false;
        if op = add
        then
            temp := generate_unique_process_name(PPT);
            PPT := PPT with (into (msg.invoke.objpar) := temp);
        elif op = delete
        then
            PPT := PPT with (mapomit (msg.invoke.objpar));
        end;
    else
        msg.reply.error := true;
        msg.reply.errcode := NoPermission;
    end;
end;

```

```

end;
end; {procedure}

procedure update_PT(op: abst_op; var msg: sendmessage;
                   var PT: proctableentry) = pending;

end; {scope sdos for sdb }

```

A.8 The System Specification

```

scope sdos = {extending sdos scope}
begin

procedure system(var host_exists: hostboolarr;
                 var user_exists: userboolarr) =
begin
    var port: hostbufarr;
    cobegin
        each hn: hostname, host(hn, port);
        each un: username, user(un);
    end;
end; { system }

procedure host(thishost: hostname; var port: hostbufarr) =
begin
    var procport, procreqport: procbufarr;
    var userport, userreqport: procbufarr;
    var table: procnamemap;
    var table1: passwordtabletype;
    var localtips: tiplist;
    cobegin
        message_switch(thishost, port, procport, procreqport);
        catalog_manager(thishost, table(minproc), procport(minproc),
                        procreqport(minproc));
        file_manager(thishost, table(minproc+2), procport(minproc+2),
                        procreqport(minproc+2));
        auth(procport(minproc+1), procreqport(minproc+1), table1, thishost,
            localtips);
        each pn: procname_3,
            process(thishost, table(pn), procport(pn), procreqport(pn));
        each tp: tipname,
            tip(procport(minproc+1), procreqport(minproc+1), userport(tp),
                userreqport(tp), thishost);
    end;
end;

```

```
    end;  
end; { host }  
  
procedure process(thishost:hostname; thisproc:procname;  
                  var inport:procbuf; var outport:procbuf) = pending;  
  
procedure user(thisuser: username) = pending;  
  
end; { scope }
```

Appendix B

Transformed Specifications for the File Manager

```
scope sdos =  
begin
```

```
procedure file_manager(    thishost    : uid;  
                           thisproc    : uid;  
                           var port     : procbuf <input>;  
                           var kport   : procbuf <output>;  
                           var kport_sh: procbuf <output>) =
```

```
begin
```

```
  var call, reply, oldcall, tmpcall, host_call : sendmessage;  
  var i: integer;
```

```
  var pending_ops: eventarray;  
  var OPENAT: access_level_table;  
  var GhostTable: ghostmap;  
  var OPENFOR: access_mode_table;
```

```
  var call_sh, reply_sh, oldcall_sh, tmpcall_sh, host_call_sh : sendmessage;  
  var pending_ops_sh: eventarray;  
  var OPENAT_sh: access_level_table;  
  var GhostTable_sh: ghostmap;  
  var OPENFOR_sh: access_mode_table;
```

```
  const DD:data := null(data);
```

loop

```

assert (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
[pending_ops(call.level) = pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
[already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
[purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

```

receive call from port;

```

if dominates(l,call.level)
then

```

```

    call_sh := call;

```

```

assert
    dominates(l,call.level) and
    (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
[pending_ops(call.level) = pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
[already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
[call = call_sh] and
[purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

```

```

if call.control = InvokeOp
then

```

```

    tmpcall := call;
    tmpcall.invoke.object := map_onto_ghost(call,GhostTable);
    tmpcall_sh := call_sh;
    tmpcall_sh.invoke.object := map_onto_ghost(call_sh,GhostTable_sh);

```

```

    case call.invoke.operation

```


is Openfile:

```

assert
    dominates(l,call.level) and
    (all call:sendmessage, dominates(l,call.level) ->
    [has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
    [map_onto_ghost(call,GhostTable) =
    map_onto_ghost(call,GhostTable_sh)] and
    [pending_ops(call.level) = pending_ops_sh(call.level)] and
    [has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
    [write_locked_by_another(call,openat) iff
    write_locked_by_another(call,openat_sh)] and
    [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
    [call = call_sh] and
    [tmpcall = tmpcall_sh] and
    [tmpcall.level = call.level] and
    [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)] and
    [already_open(tmpcall,OPENAT) iff already_open(tmpcall,OPENAT_sh)] and
    [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))]);

    if already_open(tmpcall,OPENAT) or already_open(call,OPENAT)
    then
        reply :=
        fill_reply(call,thisproc,thishost,DD,true,AlreadyOpen);
        send reply to kport;
        reply_sh :=
        fill_reply(call_sh,thisproc,thishost,DD,true,AlreadyOpen);
        send reply_sh to kport_sh;

        elif write_locked_by_another(call,OPENAT) and
        (mode_param(call.invoke.param) = write or
        mode_param(call.invoke.param) = ReadWrite)
        then
            reply :=
            fill_reply(call,thisproc,thishost,DD,true,InUse);
            send reply to kport;
            reply_sh :=
            fill_reply(call_sh,thisproc,thishost,DD,true,InUse);
            send reply_sh to kport_sh;

    else
        host_call := fill_call(call,ReadSDBEntry,thisproc,thishost);
        pending_ops(call.level) := pending_ops(call.level) <: call;
        send host_call to kport;

        host_call_sh :=
        fill_call(call_sh,ReadSDBEntry,thisproc,thishost);
        pending_ops_sh(call_sh.level) :=
        pending_ops_sh(call_sh.level) <: call_sh;

```

```

    send host_call_sh to kport_sh;
end; {if already_open}

```

```

is ReadFile:

```

```

assert
  dominates(l,call.level) and
    (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
    [pending_ops(call.level) pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
    [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
[call = call_sh] and
[tmpcall = tmpcall_sh] and
[tmpcall.level = call.level] and
[has_access(tmpcall,OPENFOR) iff has_access(tmpcall,OPENFOR_sh)] and
[already_open(tmpcall,OPENAT) iff already_open(tmpcall,OPENAT_sh)] and
    [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

    if not has_access(tmpcall, OPENFOR) or
        not already_open(tmpcall,OPENAT)
    then
        reply := fill_reply(call,thisproc,thishost,DD,true,NotOpen);
        send reply to kport;
        reply_sh :=
        fill_reply(call_sh,thisproc,thishost,DD,true,NotOpen);
        send reply_sh to kport_sh;
    else
        host_call :=
fill_call(tmpcall,ReadODBEntry,thisproc,thishost);
        pending_ops(call.level) :=
            pending_ops(call.level) <: call;
        send host_call to kport;
        host_call_sh :=
fill_call(tmpcall_sh,ReadODBEntry,thisproc,thishost);
        pending_ops_sh(call_sh.level) :=
            pending_ops_sh(call_sh.level) <: call_sh ;
        send host_call_sh to kport_sh;
    end; {if not has_access}

```

is WriteFile:

```

assert
dominates(l,call.level) and
  (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
  [pending_ops(call.level) = pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
  [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
[call = call_sh] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh)] and
[already_open(call,OPENAT) iff already_open(call,OPENAT_sh)] and
  [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

  if not has_access(call, OPENFOR) or
    not already_open(call,OPENAT)
  then
    reply := fill_reply(call,thisproc,thishost,DD,true,NotOpen);
    send reply to kport;
    reply_sh :=
fill_reply(call_sh,thisproc,thishost,DD,true,NotOpen);
    send reply_sh to kport_sh;
  else
    host_call := fill_call(call,WriteODBEntry,thisproc,thishost);
    pending_ops(call.level) :=
      pending_ops(call.level) <: call;
    send host_call to kport;
    host_call_sh :=
      fill_call(call_sh,WriteODBEntry,thisproc,thishost);
    pending_ops_sh(call_sh.level) :=
      pending_ops_sh(call_sh.level) <: call_sh;
    send host_call_sh to kport_sh;
  end; {if not has_access}

```

is CloseFile:

```

assert
dominates(l,call.level) and
  (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
  [pending_ops(call.level) = pending_ops_sh(call.level)] and

```

```

[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
    [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
[call = call_sh] and
[tmpcall = tmpcall_sh] and
[tmpcall.level = call.level] and
[has_access(tmpcall,OPENFOR) iff has_access(tmpcall,OPENFOR_sh)] and
[already_open(tmpcall,OPENAT) iff already_open(tmpcall,OPENAT_sh)] and
    [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

```

```

        if not has_access(tmpcall, OPENFOR) or
            not already_open(tmpcall,OPENAT)
        then
            reply := fill_reply(call,thisproc,thishost,DD,true,NotOpen);
            send reply to kport;
            reply_sh :=
fill_reply(call_sh,thisproc,thishost,DD,true,NotOpen);
            send reply_sh to kport_sh;
        else
            purge_OPENFOR(tmpcall,OPENFOR);
            purge_OPENAT(tmpcall,OPENAT);
            purge_GT(tmpcall,GhostTable);
            purge_OPENFOR(tmpcall_sh,OPENFOR_sh);
            purge_OPENAT(tmpcall_sh,OPENAT_sh);
            purge_GT(tmpcall_sh,GhostTable_sh);
            end; {if not has_access}

        if tmpcall.invoke.object ne call.invoke.object
        then
            host_call :=
fill_call(tmpcall,RemoveODBEntry,thisproc,thishost);
            send host_call to kport;
            host_call_sh :=
fill_call(tmpcall_sh,RemoveODBEntry,thisproc,thishost);
            send host_call_sh to kport_sh;
        end;

```

is CreateFile:

```

            host_call :=
fill_call(call,CreateODBEntry,thisproc,thishost);
            pending_ops(call.level) := pending_ops(call.level) <: call;
            send host_call to kport;

            host_call_sh :=
fill_call(call_sh,CreateODBEntry,thisproc,thishost);

```

```

        pending_ops_sh(call_sh.level) :=
pending_ops_sh(call_sh.level) <: call_sh;
        send host_call_sh to kport_sh;

```

```

is DeleteFile:

```

```

        host_call :=
fill_call(call, RemoveODBEntry, thisproc, thishost);
        pending_ops(call.level) := pending_ops(call.level) <: call;
        send host_call to kport;

```

```

        host_call_sh :=
fill_call(call_sh, RemoveODBEntry, thisproc, thishost);
        pending_ops_sh(call_sh.level) :=
pending_ops_sh(call_sh.level) <: call_sh;
        send host_call_sh to kport_sh;

```

```

        else: {of case}
        reply := fill_reply(call, thisproc, thishost, DD, true, UndefOp);
        send reply to kport;
        reply_sh :=
fill_reply(call_sh, thisproc, thishost, DD, true, UndefOp);
        send reply_sh to kport_sh;
        end; {case}

```

```

else {if call.control = ReplyOp}

```

```

{HANDLING REPLIES FROM THE KERNEL -

```

```

  in cases where an entry exists in pending_ops for the reply, additional
  action is taken,
  otherwise, the response is relayed to the client}

```

```

i := 0;

```

```

loop

```

```

assert dominates(1, call.level) and
  (all call:sendmessage, dominates(1, call.level) ->
[has_ghost(call, GhostTable) iff has_ghost(call, GhostTable_sh)] and
[map_onto_ghost(call, GhostTable) =

```

```

map_onto_ghost(call,GhostTable_sh)] and
  [pending_ops(call.level) = pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
  [already_open(call,OPENAT) iff already_open(call,OPENAT_sh))] and
[call = call_sh] and
  [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

  if i>size(pending_ops(call.level)) then leave;
  elif call.level = pending_ops(call.level)(i).level and
    call.transac1bl = pending_ops(call.level)(i).transac1bl
  then

    oldcall := pending_ops(call.level)(i);
    oldcall_sh := pending_ops_sh(call.level)(i);

    pending_ops(call.level) :=
pending_ops(call.level) with (seqomit(i));
    pending_ops_sh(call_sh.level) :=
pending_ops_sh(call_sh.level) with (seqomit(i));

assert
dominates(l,call.level) and
  (all call:sendmessage, dominates(l,call.level) ->
[has_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
[map_onto_ghost(call,GhostTable) =
map_onto_ghost(call,GhostTable_sh)] and
  [pending_ops(call.level) = pending_ops_sh(call.level)] and
[has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
[write_locked_by_another(call,openat) iff
write_locked_by_another(call,openat_sh)] and
  [already_open(call,OPENAT) iff already_open(call,OPENAT_sh))] and
[call = call_sh] and
[oldcall = oldcall_sh] and
  [oldcall.level = call.level] and
  [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))];

  if not call.reply.error
  then

    case oldcall.invoke.operation

    is OpenFile:

      if unset(call.reply.obj1bl) = oldcall.level and
(mode_param(oldcall.invoke.param) = write or
mode_param(oldcall.invoke.param) = ReadWrite) and

```

```

    not write_locked_by_another(call, OPENAT)
    then
        update_OPENFOR(oldcall, OPENFOR);
        update_OPENAT(oldcall, OPENAT);
    reply := fill_reply(oldcall, thisproc, thishost,
DD, false, NoMesg);
        send reply to kport;

    update_OPENFOR(oldcall_sh, OPENFOR_sh);
    update_OPENAT(oldcall_sh, OPENAT_sh);
    reply_sh := fill_reply(oldcall_sh, thisproc, thishost,
        DD, false, NoMesg);
        send reply_sh to kport_sh;

    elif (dominates(oldcall.level,
unset(call.reply.objlbl)) and
    mode_param(oldcall.invoke.param) = read)
    then
        host_call := fill_call(oldcall, CreateODBEntry,
thisproc, thishost);
        pending_ops(oldcall.level) :=
pending_ops(oldcall.level) <: oldcall
        with (.invoke.operation := marker2);
        send host_call to kport;

        host_call_sh :=
            fill_call(oldcall_sh, CreateODBEntry, thisproc,
thishost);
        pending_ops_sh(oldcall_sh.level) :=
pending_ops_sh(oldcall_sh.level) <:
            oldcall_sh with
            (.invoke.operation := marker2);
        send host_call_sh to kport_sh;

    else
        reply := fill_reply(oldcall, thisproc, thishost, DD,
            true, NoPermission);
        send reply to kport;
        reply_sh := fill_reply(oldcall_sh, thisproc, thishost,
DD, true, NoPermission);
        send reply_sh to kport_sh;

    end;

is marker2:

    host_call := fill_call(oldcall, CopyODBEntry,
thisproc, thishost);
        pending_ops(oldcall.level) :=
pending_ops(oldcall.level) <: oldcall with
            (.invoke.operation := marker1);
        send host_call to kport;

```

```

host_call_sh := fill_call(oldcall_sh, CopyODBEntry,
    thisproc, thishost);
    pending_ops_sh(oldcall_sh.level) :=
pending_ops_sh(oldcall_sh.level) <:
oldcall_sh with
(.invoke.operation := marker1);
    send host_call_sh to kport_sh;

is marker1:

    update_OPENFOR(oldcall, OPENFOR);
    update_OPENAT(oldcall, OPENAT);
update_GT(oldcall, GhostTable);
    reply := fill_reply(oldcall, thisproc, thishost, DD,
        false, NoMesg);
    send reply to kport;

update_OPENFOR(oldcall_sh, OPENFOR_sh);
    update_OPENAT(oldcall_sh, OPENAT_sh);
update_GT(oldcall_sh, GhostTable_sh);
reply_sh := fill_reply(oldcall_sh, thisproc, thishost,
    DD, false, NoMesg);
    send reply_sh to kport_sh;

is DeleteFile:

    purge_OPENFOR(call, OPENFOR);
    purge_OPENAT(call, OPENAT);
    purge_GT(call, GhostTable);
    reply := fill_reply(oldcall, thisproc, thishost, DD,
        false, NoMesg);
    send reply to kport;

    purge_OPENFOR(call_sh, OPENFOR_sh);
    purge_OPENAT(call_sh, OPENAT_sh);
    purge_GT(call_sh, GhostTable_sh);
reply_sh := fill_reply(oldcall_sh, thisproc, thishost,
    DD, false, NoMesg);
    send reply_sh to kport_sh;

else: {case}

reply := fill_reply(oldcall, thisproc, thishost,
    call.reply.param, call.reply.error, call.reply.errcode);
    send reply to kport;
reply_sh := fill_reply(oldcall_sh, thisproc, thishost,
    call_sh.reply.param, call_sh.reply.error,

```



```

    call_sh.reply.errcode);
        send reply_sh to kport_sh;
    end; {case}

    leave;

    else {if not call.reply.error}

        reply := fill_reply(oldcall, thisproc, thishost,
        DD, call.reply.error, call.reply.errcode);
        send reply to kport; {relay kernel's reply back to client}
        reply_sh :=
        fill_reply(oldcall_sh, thisproc, thishost, DD,
        call_sh.reply.error, call_sh.reply.errcode);

        send reply_sh to kport_sh;

        leave;

        end; {if not reply.error}

        else {if i = ...}
            i := i + 1;
        end; {if i > size(pending_ops)}

    end; {loop}

end; {if InvokeOp}

else {of of dominates.....}

if call.control = InvokeOp
then
    tmpcall := call;
    tmpcall.invoke.object := map_onto_ghost(call, GhostTable);

    case call.invoke.operation

    is Openfile:

        if already_open(tmpcall.openat) or already_open(call.openat)
        then
            reply := fill_reply(call, thisproc, thishost, DD, true, AlreadyOpen);

```

```

        send reply to kport;

        elif write_locked_by_another(call,openat) and
            (mode_param(call.invoke.param) = write or
             mode_param(call.invoke.param) = ReadWrite)
            then
                reply :=
                fill_reply(call,thisproc,thishost,DD,true,InUse);
                send reply to kport;

        else
            host_call := fill_call(call,ReadSDBEntry,thisproc,thishost);
            pending_ops(call.level) := pending_ops(call.level) <: call;
            send host_call to kport;

        end; {if already_open}

is ReadFile:

        if not has_access(tmpcall, openfor) or
            not already_open(tmpcall,openat)
            then
                reply := fill_reply(call,thisproc,thishost,DD,true,NotOpen);
                send reply to kport;
            else
                host_call := fill_call_temp(call,map_onto_ghost(call,
GHOSTTABLE),ReadODBEntry,thisproc,thishost);
                pending_ops(call.level) :=
                    pending_ops(call.level) <: call;
                send host_call to kport;
            end; {if not has_access}

is WriteFile:

        if not has_access(call, openfor) or
            not already_open(call,openat)
            then
                reply := fill_reply(call,thisproc,thishost,DD,true,NotOpen);
                send reply to kport;
            else
                host_call := fill_call(call,WriteODBEntry,thisproc,thishost);

```

```

        pending_ops(call.level) :=
            pending_ops(call.level) <: call;
        send host_call to kport;

    end; {if not has_access}

is CloseFile:

    if not has_access(tmpcall, openfor) or
        not already_open(tmpcall, openat)
    then
        reply := fill_reply(call, thisproc, thishost, DD, true, NotOpen);
        send reply to kport;
    else
        purge_openfor(tmpcall, openfor);
        purge_openat(tmpcall, openat);
        purge_GT(tmpcall, GhostTable);
    end; {if not has_access}

if tmpcall.invoke.object ne call.invoke.object
then
    host_call := fill_call_temp(call, map_onto_ghost(call,
GHOSTTABLE), RemoveODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;
end;

is CreateFile:

    host_call := fill_call(call, CreateODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;

is DeleteFile:

    host_call := fill_call(call, RemoveODBEntry, thisproc, thishost);
    pending_ops(call.level) := pending_ops(call.level) <: call;
    send host_call to kport;

else: {of case}
reply := fill_reply(call, thisproc, thishost, DD, true, UndefOp);
    send reply to kport;

end; {case}

```

```

else {if call.control = ReplyOp}

{REPLIES FROM THE KERNEL -
  in cases where an entry exists in pending_ops for the reply, additional
  action is taken,
  otherwise, the response is relayed to the client}

  i := 0;

  loop
  assert
    not dominates(l,call.level) and
    (all call:sendmessage, dominates(l,call.level) ->
    [as_ghost(call,GhostTable) iff has_ghost(call,GhostTable_sh)] and
    [map_onto_ghost(call,GhostTable) =
    map_onto_ghost(call,GhostTable_sh)] and
    [pending_ops(call.level) = pending_ops_sh(call.level)] and
    [has_access(call,OPENFOR) iff has_access(call,OPENFOR_sh) ] and
    [write_locked_by_another(call,openat) iff
    write_locked_by_another(call,openat_sh)] and
    [already_open(call,OPENAT) iff already_open(call,OPENAT_sh)]) and
    [purge(outto(kport,myid)) = purge(outto(kport_sh,myid))]);

    if i>size(pending_ops(call.level)) then leave;
    elif call.level = pending_ops(call.level)(i).level and
      call.transaclbl = pending_ops(call.level)(i).transaclbl
    then

      oldcall := pending_ops(call.level)(i);

      pending_ops(call.level) :=
      pending_ops(call.level) with (seqomit(i));

      if not call.reply.error
      then

        case oldcall.invoke.operation

        is OpenFile:

```

372 APPENDIX B. TRANSFORMED SPECIFICATIONS FOR THE FILE MANAGER

```

        if unset(call.reply.objlbl) = oldcall.level and
        (mode_param(oldcall.invoke.param) = write or
         mode_param(oldcall.invoke.param) = ReadWrite) and
         not write_locked_by_another(call, OPENAT)
        then
            update_openfor(oldcall, openfor);
            update_openat(oldcall, openat);
            reply := fill_reply(oldcall, thisproc, thishost,
                                DD, false, NoMesg);
            send reply to kport;

        elif (dominates(oldcall.level,
                        unset(call.reply.objlbl)) and
              mode_param(oldcall.invoke.param) = read)
        then
            host_call :=
                fill_call(oldcall, CreateODBEntry, thisproc,
                          thishost);
            pending_ops(oldcall.level) :=
                pending_ops(oldcall.level) <: oldcall
            with (.invoke.operation := marker2);
            send host_call to kport;

        else
            reply := fill_reply(oldcall, thisproc, thishost,
                                DD, true, NoPermission);
            send reply to kport;

        end;

    is marker2:

        host_call :=
            fill_call(oldcall, CopyODBEntry, thisproc, thishost);
            pending_ops(oldcall.level) :=
                pending_ops(oldcall.level) <: oldcall with
            (.invoke.operation := marker1);
            send host_call to kport;

    is marker1:

        update_openfor(oldcall, openfor);
        update_openat(oldcall, openat);
        update_GT(oldcall, GhostTable);
        reply := fill_reply(oldcall, thisproc, thishost,
                            DD, false, NoMesg);
        send reply to kport;

```

```

is DeleteFile:

    purge_openfor(call,openfor);
    purge_openat(call,openat);
    purge_GT(call,GhostTable);

    reply := fill_reply(oldcall,thisproc,thishost,
DD,false,NoMesg);
    send reply to kport;

    else: {case}

reply := fill_reply(oldcall,thisproc,thishost,
call.reply.param,call.reply.error,call.reply.errcode);
    send reply to kport;

    end; {case}

    leave;

    else {if not call.reply.error}

reply := fill_reply(oldcall,thisproc,thishost,DD,
call.reply.error,call.reply.errcode);
    send reply to kport;

    leave;

    end; {if not call.reply.error}

    else {if i = ...}
        i := i + 1;
    end;{if i > size(pending_ops)}

    end; {loop}

    end; {if InvokeOp}

    end; {if dominates}

    end; {outermost loop}
end; {file_manager}

end; {sdos scope}

```

374 APPENDIX B. TRANSFORMED SPECIFICATIONS FOR THE FILE MANAGER

The function and type declarations are same as in the File Manager specifications given in Appendix A.

Appendix C

Glossary

- **ACI: Actual client identity.**
The principal bound to a client process that is used for discretionary access control. See section 3.6.4.
- **ACL: Access control list.**
Associated with each object, it is a list associated with an object that is used for discretionary access controls to determine if a client process may access the object.
- **C2: Command and control.**
An application area defined by the Department of Defense (DoD).
- **CCI: Contextual Client Identity.**
An identity of a client from a particular context that includes a principal, project, and role. Used for discretionary access control. See section 3.6.4.
- **CI: Client identity.**
Name for all the identities used by discretionary access control mechanisms to identify a client process. See section 3.6.4.
- **CSP: Concurrent Sequential Processes.**
A language create by C.A.R. Hoare to describe the concurrent execution of processes in a computer system.
- **DBMS: Database management system.**
Software that provides associated access to data.
- **DAC: Discretionary access controls.**
A security policy and mechanism used to control the access of clients to objects. See section 3.6.4.
- **DOS: Distributed operating system.**
A set of software that allows the integration and interaction of applications executing on more than one host.

- **DTCB: Distributed TCB.**
The trusted computing based that executes across many hosts that enforces the mandatory security policy in a secure distributed operating system. See section 5.1.3.
- **DTLS: Descriptive top-level specification.**
A specification of the interface of a system as seen from outside the system. See section 3.5.
- **FTLS: Formal top level specification.**
A document that presents a formal description of interface of a software system. Verification of the software requires that a formal correspondence be established between the FTLS and the formal model.
- **IACL: Initial access control list.**
The default access control list that an object has before it is created.
- **IP: Internet Protocol**
A network layer protocol used throughout the Internet. See Section 3.5.
- **MLS: Multilevel secure**
A software or hardware component that is trusted to maintain and keep separate information from more than one access class.
- **NCSC: National Computer Security Center**
The agency of the U.S. Government that establishes the evaluation criteria for secure systems (see [DoD Criteria 85] and [NCSC TNI 87]) and for evaluating systems' security for the government.
- **NTCB: Network trusted computing base**
Defined by the Trusted Network Interpretation [NCSC TNI 87] as the trusted computing base in network systems.
- **ODB: Object database**
The component of the kernel that is responsible for maintaining the representation of abstract objects on secondary storage. Object managers use it to store persistent objects. It is a multi-level secure component of the trusted computing base in SDOS.
- **OSI: Open Systems Interconnection**
A design methodology and model for building distributed system that is based on layered communication between heterogeneous hosts. See Section 3.5.
- **PDU: Protocol data units**
Generic unit of information being transferred between peers in a communication hierarchy. See section 3.5.
- **PM: Process Manager**
The SDOS manager that is responsible for the management of processes. The Process Manager executes on every host and is a component of the kernel. See section 3.6.2.

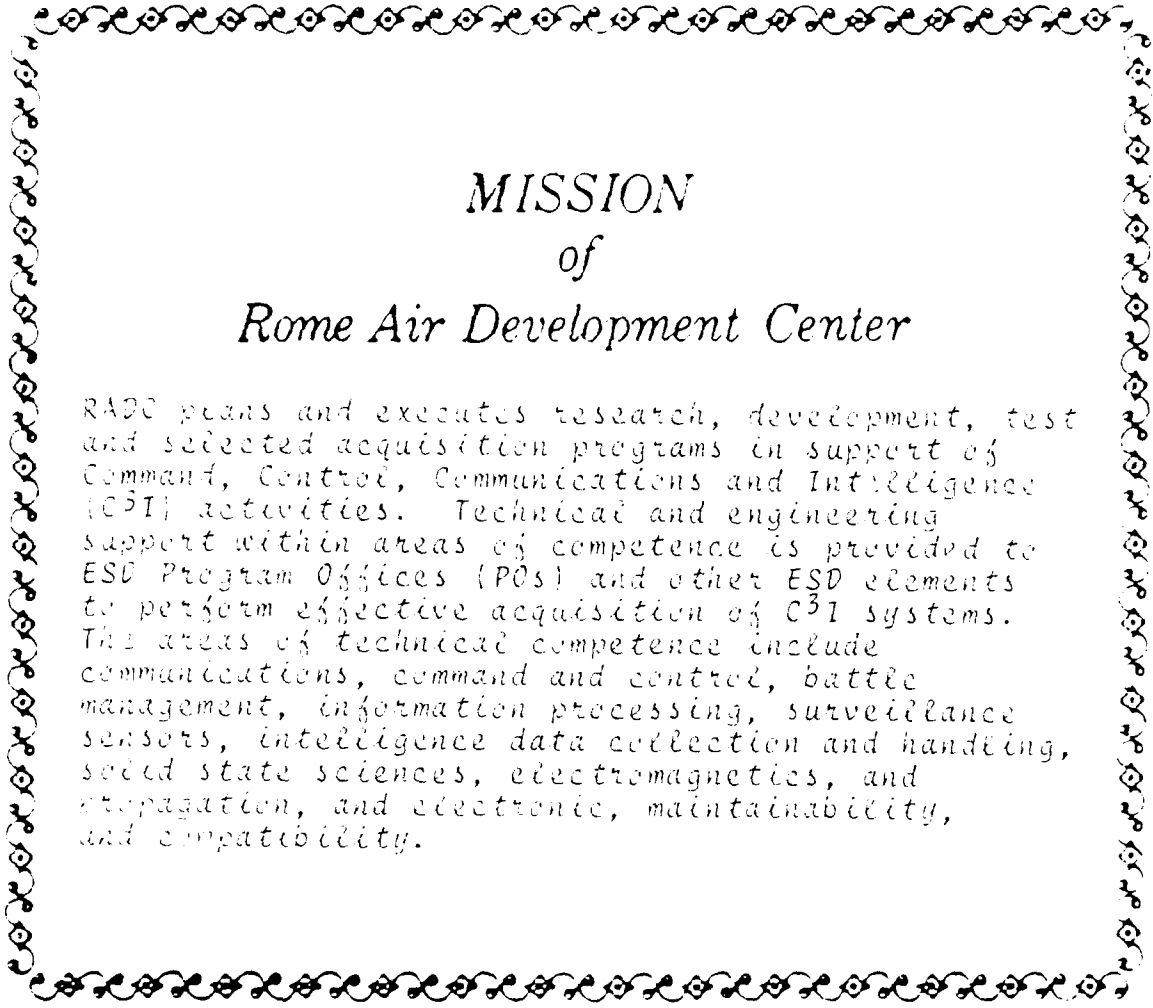
- **SDB: Security Database**
The Security Database is a component of the kernel that is responsible for maintaining the security labels for all objects on the host. See Section 3.5.2.1.
- **SDOS: Secure Distributed Operating System**
Name of the system described in this report.
- **SDU: Service data units**
Generic unit of information being transferred between adjacent layers in a communication hierarchy. See section 3.5.
- **SNI: Strong noninterference**
A security property on a procedure that says for any level l and any history of message-passing events, a new history could be constructed by purging all events not visible at level l , including both parameter passing events and control events. See Section 4.2
- **TCP: Transport communication protocol**
A DoD standard connection-based transport layer protocol that is implemented on top of IP.
- **TCSEC: Trusted Computer System Evaluation Criteria**
The *Orange Book* produced by the National Computer Security Center that is used to evaluate the security of centralized computer systems (see [DoD Criteria 85]).
- **TIP: Terminal Interface Process**
A trusted, multi-level secure process in SDOS that provides a trusted path for users to other MLS components. It is a user interface process. See [DoD Criteria 85] for a complete description.
- **TNI: Trusted Network Interpretation**
The equivalent of the Trusted Computer System Evaluation Criteria (Orange Book) for network systems, it is used to evaluate the security of network systems (see [NCSC TNI 87]).
- **UID: Unique identifier**
A unique identifier that is used by the SDOS kernel to locate and identify an object.
- **UNO: Unique number**
A component of a UID, a unique number is generated by concatenating the host identifier with a sequence number.
- **WNI: Weak noninterference**
WNI states that for every trace α there is another trace α' which bears a certain relation to α . See section 4.2.2.3.

Bibliography

- [Bell and LaPadula 76] Bell, D.E., LaPadula, L.J., "Secure Computer Systems: Unified Exposition and Multics Interpretation," *Mitre Corp. Technical Report MTR-2997*, Revision 2, March 1976.
- [Berets et al. 85] Berets, J.C., Mucci, R.A., Schantz, R.E., "Cronus: A Testbed for Developing Distributed Systems," *Proceedings of the IEEE MILCOM Communications Conference*, October 1985.
- [Biba 77] Biba, K.J., "Integrity Considerations For Secure Computer Systems," *Mitre Corp. Technical Report MTR-3159*, April 1977.
- [Brookes et al. 84] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W., "A Theory of Communicating Sequential Processes", *Journal of the ACM*, vol. 31, no. 3, 1984.
- [Casey et al. 88] Casey, T.A., Vinter, S.T., Weber, D.G., Varadarajan, R., Rosenthal, D., "A Secure Distributed Operating System", *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.
- [Cronus 88] BBN Laboratories, "Cronus User's Reference Manual," January 1988.
- [DoD Criteria 85] "Department of Defense Trusted Computer System Evaluation Criteria", National Computer Security Center, Standard DOD 5200.28-STD, December 1985.
- [DoD Guidance 85] "Guidance for Applying the DoD Trusted Computer System Evaluation Criteria in Specific Environments", DoD Computer Security Center, CSC-STD-004-85, June 1985.
- [DoD Password 85] "Department of Defense Password Management Guideline", DoD Computer Security Center, CSC-STD-002-85, June 1985.

- [Goguen and Meseguer 82] Goguen, J.A., Meseguer, J., "Security Policy and Security Models," *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.
- [Good et al. 78] Good, G.I., et al., "Report on the Gypsy Language, Version 2.0," *Technical Report TR ICSCA-CMP-10*, Institute of Computer Science, University of Texas, Austin, September 1978.
- [Jones 78] Jones, A.K., "The Object Model: A Conceptual Tool for Structuring Software," *Operating Systems, An Advanced Course; Lecture Notes in Computer Science*, Springer-Verlag, Editors Bayer, Graham, and Seegmuller, 1978.
- [Lamport 78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, 1978.
- [Landwehr 83] Landwehr, C.E., "The Best Available Technologies for Computer Security," *IEEE Computer*, vol. 16, no. 7, July 1983.
- [Landwehr et al. 84] Landwehr, C.E., Heitmeyer, C.L., McLean, J., "A Security Model for Military Message Systems," *Naval Research Laboratory Report 8806*, May 1984.
- [McCauley and Drongowski 79] McCauley, E.J., Drongowski, P.J., "KSOS - The Design of a Secure Operating System," *Proceedings of the AFIPS National Computer Conference*, June 1979.
- [McCullough 87] McCullough, D. "Specifications for Multi-Level Security and a Hook-Up Property", *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, May 1987.
- [NCSC TNI 87] "Trusted Network Interpretation of the TCSEC", National Computer Security Center, NCSC-TG-005, Version-1, July 1987.
- [Reed and Kanodia 79] Reed, D.P., and Kanodia, R.K., "Synchronization with Eventcounts and Sequencers", *Communications of the ACM*, Vol. 22, No. 2, February 1979, pp. 115-124.
- [Schantz et al. 86] Schantz, R., Thomas, R., Bono, G., "The Architecture of the Cronus Distributed Operating System," *Proceedings of the IEEE 6th International Conference on Distributed Computing Systems*, Boston, MA, May 1986, pp 250-259.

- [Schell and Tao 84] Schell, R. R., and Tao, T. F., "Microcomputer-Based Trusted Systems for Communication and Workstation Applications," *Proceedings of the 7th Annual DoD/NBS Computer Security Conference*, NBS, Gaithersburg, MD, Sept. 1984.
- [Sullivan 86] Sullivan, E.C., Lunt, T. F., Proctor, N., "A Multilevel Object Security Model," *RADC-TR-86-10*, March 1986.
- [Ulysses 87] "Foundations of Ulysses: The Theory of Security", ORA Tech. Report for RADC contract F30602-85-C-0098, April 1987.
- [Vinter 88] "Extended Discretionary Access Controls," *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.
- [Weber87] Weber, D.G., and Lubarsky, R., "The SDOS Project—Verifying Hook-up Security," *Proceedings of the 3rd Aerospace Computer Security Conference*, December 1987.
- [Workshop 85] *Proceedings of the Department of Defense Computer Security Center Invitational Workshop on Network Security*, March 1985.
- [Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic maintainability, and compatibility.