1987

Highly-structured software for network systems and its protection

Masaaki Mizuno
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17”x 23” black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6”x 9” black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
Highly-structured software for network systems and its protection

Mizuno, Masaaki, Ph.D.
Iowa State University, 1987

Copyright ©1987 by Mizuno, Masaaki. All rights reserved.
Highly-structured software for network systems
and its protection

by

Masaaki Mizuno

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major: Computer Science

Approved:

[Signature was redacted for privacy.]

In Charge of Major Work
[Signature was redacted for privacy.]

For the Major Department
[Signature was redacted for privacy.]

For the Graduate College

Iowa State University
Ames, Iowa
1987

Copyright © Masaaki Mizuno, 1987. All rights reserved.
# TABLE OF CONTENTS

## 1 INTRODUCTION
1.1 Overview ............................................................................. 1
1.2 Statement of Problems .................................................. 3
   1.2.1 Issues in access control ........................................... 3
   1.2.2 Issues in information flow control ......................... 4
1.3 Organization of Thesis .................................................. 6

## 2 THE RESOURCE MODULE SYSTEM ............................................ 9
2.1 An Overview of the Resource Module System ...................... 9
   2.1.1 Access component ................................................ 9
   2.1.2 Synchronization component .................................. 11
   2.1.3 Protection component ......................................... 11
   2.1.4 Flow of messages ............................................... 12
2.2 Classes, Instances, Names and Types ............................... 14
   2.2.1 Class concept and class hierarchy .......................... 14
   2.2.2 Name management ............................................. 15
   2.2.3 Classes and types ............................................. 15
2.3 Program Examples ..................................................... 16
2.4 Class Operations and Instance Operations ......................... 25
3 THE ACCESS CONTROL MECHANISM

3.1 Access Matrix Model ........................................ 41
3.2 Access Control Mechanisms .................................. 43
3.3 Comparison of Capabilities and Access Control Lists .......... 44
  3.3.1 Efficiency ................................................. 44
  3.3.2 Ownership ................................................ 45
  3.3.3 Revocation of access ..................................... 46
  3.3.4 Propagation of access .................................... 47
  3.3.5 Review of access ......................................... 48
  3.3.6 Least privilege ........................................... 48
3.4 Proposed Access Control Mechanism ................................ 54
  3.4.1 Four-tuple ACL entry ..................................... 54
  3.4.2 Access-rights expression .................................. 60

4 THE INFORMATION FLOW MECHANISM ............................. 66

4.1 Definition of Flow Control and a Review of Previous Work ........ 66
  4.1.1 Access matrix model ...................................... 66
  4.1.2 Information flow model .................................... 68
  4.1.3 Restrictions of the access matrix model .................... 73
  4.1.4 Problems of applying existing information flow models to the
        RM system ................................................ 74
4.2 Overview of the Information Flow Control Mechanism .............. 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>Introduction</td>
<td>83</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Handling implicit flows</td>
<td>94</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Some remarks on probes and dynamically bound state variables</td>
<td>105</td>
</tr>
<tr>
<td>4.3</td>
<td>The Compile-Time Algorithm</td>
<td>107</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Reduction rules</td>
<td>107</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Generation of information flow templates</td>
<td>109</td>
</tr>
<tr>
<td>4.4</td>
<td>The Run-Time Algorithm</td>
<td>124</td>
</tr>
<tr>
<td>4.5</td>
<td>Program Examples</td>
<td>130</td>
</tr>
<tr>
<td>4.5.1</td>
<td>A program with no dynamically bound state variables</td>
<td>130</td>
</tr>
<tr>
<td>4.5.2</td>
<td>A program with dynamically bound state variables</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>LINK-TIME INFORMATION FLOW CERTIFICATION</td>
<td>155</td>
</tr>
<tr>
<td>5.1</td>
<td>The Link-Time Mechanism</td>
<td>157</td>
</tr>
<tr>
<td>5.2</td>
<td>Termination and Correctness of the Link-Time Algorithm</td>
<td>173</td>
</tr>
<tr>
<td>5.3</td>
<td>The Modified Link-Time Mechanism</td>
<td>176</td>
</tr>
<tr>
<td>5.4</td>
<td>The Link-time/Run-time Algorithm Allowing Dynamically Bound State Variables</td>
<td>194</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparisons of the Information Flow Mechanisms</td>
<td>205</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSION</td>
<td>212</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>212</td>
</tr>
<tr>
<td>6.2</td>
<td>Areas of Future Research</td>
<td>215</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Access component</td>
<td>215</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Protection component</td>
<td>216</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1. Internal Structure of a Resource Module .................. 10
Figure 2.2. A Program with No Resource Sharing ....................... 18
Figure 2.3. A Program for a Telephone Directory ....................... 20
Figure 2.4. A Program for a Private Mail Service ...................... 22
Figure 2.5. An Object as Encapsulated Data .......................... 27
Figure 2.6. An Object as a Computational Entity ....................... 28
Figure 2.7. A Program for MATRIX without Class Operations ........ 30
Figure 2.8. A Program for MATRIX with Class Operations .............. 31
Figure 2.9. Structuring Software Using Inheritance ................... 34
Figure 2.10. A Program for Traffic Simulation Using a Trait .......... 36
Figure 2.11. Prototype Support Software for Implementing Resource Modules 38

Figure 4.1. Lattice Structure Defining an Information Flow Policy ..... 69
Figure 4.2. A Simple Program Example to Illustrate the Information Flow
Control Mechanism .................................................. 90
Figure 4.3. Implicit Flows Across Module Boundaries ................ 96
Figure 4.4. An Example Program with No Dynamically Bound State Variables and Its Information Flow Templates .......... 131
Figure 4.5. An Example Program with Dynamically Bound State Variables and Its Information Flow Templates 145

Figure 5.1. Information Flow Templates for Program with No Recursion 161

Figure 5.2. An Example Program with Recursion and Its Information Flow Templates 168

Figure 5.3. Over-Classification Caused by the Link-Time Algorithm 177

Figure 5.4. Information Flow Paths Due to Procedure Invocations (Case 1) 183

Figure 5.5. Information Flow Paths Due to Procedure Invocations (Case 2) 184

Figure 5.6. Another Example for the Modified Link-Time Algorithm 193

Figure 5.7. Information Flow Templates for Program with Dynamically Bound State Variables 197
1 INTRODUCTION

1.1 Overview

In recent years, technological advances have made it cost effective to construct large computer systems from collections of computers connected via networks. In order to support such systems, there is a growing need for effective ways to organize and maintain distributed programs in which modules reside and execute at geographically distinct locations.

The modularity afforded by an object-based approach is believed to be especially valuable in a distributed network environment. Object-oriented programming systems have been widely investigated in the field of operating systems and programming languages [2, 26, 32, 38, 42, 51]. An object is an instance of an abstract data type which encapsulates data along with operations on the data. These operations are called exported procedures of an object. The only way to access encapsulated data from the outside of an object is to invoke the exported procedures. An object is assumed to have an extensively long life, usually longer than the lifetime of most programs. This characteristic differentiates an object-oriented approach from the original abstract data type approach such as that defined by SIMULA-67 [13]. Theoretically, these objects are addressed by system wide one-time unique identifiers.

Even though other fundamental characteristics to construct effective network
software, such as synchronization of access [4,10], protection [11,16,17,41,45] and reliability [38,47], have been major issues in the field of operating systems, they have been studied, in most cases, separately, or, in some cases, combined with non-object-oriented systems.

The research presented in this thesis forms part of a larger project involving a highly-structured, distributed object-oriented system. The purposes of this project are to develop protection, synchronization and recovery mechanisms and to integrate these ideas into a single programming system for network applications.

The perceived usefulness of this system is in the design of operating systems and user application programs. Specific goals of our system are to realize

1. total encapsulation of data (along with operations on the data)
2. resource sharing with powerful mechanisms for
   - synchronization of access; and
   - three aspects of protection (type checking, access control and information flow control).
3. separate specification of protection, synchronization and access in order to facilitate programming and proofs of correctness, and
4. high software reusability — capability for building software based on the previous work of others.

Our basic unit for constructing a network object is a "resource module (RM)" [40] which is in complete charge of its own protection, synchronization, access of encapsulated resource (device or data), and error recovery. RMs can communicate by
only one method: invoking exported procedures in other RMs along with subsequent replies. By integrating numerous principles of software design into a single programming system, the system can provide an effective easy-to-use facility for building network software systems.

An RM is internally constructed from three components: protection, synchronization and access. Each of the three components is separately specified and encapsulated within the parent module and the three may be implemented as concurrent communicating processes. Previous research on the specification of a synchronization language has been done by Headington, Oldehoeft, Kumar and Jennings (Headington [30], Oldehoeft and Headington [31], Kumar [35], Oldehoeft and Jennings [39]). Issues involving recovery and an implementation of the prototype system are left to future research. The contributions of the author to this project are to

1. specify an overall implementation model of the entire system, and

2. develop new protection mechanisms suitable for a distributed object-oriented environment.

1.2 Statement of Problems

1.2.1 Issues in access control

In order to effectively share resources in the system, two types of security policies are significant: an access control policy and an information flow control policy [16,21,45]. An access control policy specifies authorization for access to objects based on the identities of subjects. Two types of mechanisms are widely used to implement an access control policy: capabilities and access control lists [16,21,45].
Some researchers feel that access control lists provide more security than capabilities [21,32,45]. However, capabilities can implement the principle of "least privilege" more effectively than access control lists [11,21]. We need to develop an access control mechanism which has the advantages of access control lists and yet effectively enforces the principle of least privilege.

Controlling the order of procedure invocations to an object is sometimes important and should be specified by the designer of the object. Consider a file object which defines OPEN, READ, WRITE and CLOSE operations as exported procedures. The OPEN procedure must be invoked before any READ or WRITE operations. Also, the CLOSE operation should follow all invocations of READ or WRITE. An access-rights expression was developed by Kieburtz and Silberschatz to specify the order of invocation of procedures in Concurrent Pascal [34]. However, in their original work, the use of access-rights expressions was controlled by the cooperation of a programmer of the procedures and a user. Hence, there is a way for a user to avoid the restriction on the order of invocation of procedures enforced by the access-rights expressions [34]. We find access-rights expressions very useful to control the behavior of individual users of an object and would like to incorporate them with an access control mechanism to effectively enforce the use of an RM in a way defined only by the designer and owner of the RM.

1.2.2 Issues in information flow control

An information flow control policy regulates the flow of information between classified objects. Given a set of "security classes" corresponding to the sensitivities of information and a specification of all the paths among objects by which information
can flow (an information flow policy), an information flow mechanism must guarantee that the flows caused by program executions do not violate the specification. Denning introduced the use of a lattice structure to define an information flow policy [15].

Based on a policy defined by a lattice, Denning developed an algorithm for certifying the secure execution of a program in an environment in which the security class of each object (program variable or file) remains constant throughout the lifetime of the program. In this environment, a programmer needs to specify the security class of each program variable. The algorithm certifies the security of a given program at compile time. Since constant security classes of parameters must also be specified, separate versions of functionally equivalent procedures are required to handle different security classes of parameters. This restriction severely reduces the possibility of resource sharing.

Denning also developed a run-time certification algorithm in an environment in which the security class of each object changes, during execution of the program, according to the information contained in the object. This approach relies on a special hardware support. Without such hardware, the mechanism is very inefficient. Since security classes of output devices or sensitive files will change during execution, a secure system cannot be constructed based on this approach.

Andrews and Reitman developed a compile-time certification algorithm based on Hoare's technique of program verification [6]. This mechanism allows the security class of each variable either to remain constant or to change during execution of the program. In this mechanism, the verification of a procedure invocation requires previous verification of the body of the called procedure and an access to its pre/post conditions. In a general distributed object-oriented system, modules of a program may
be geographically distributed or constructed at different points in time. Therefore, it may not be feasible to access protection information of other modules at compile time.

We need to develop an information flow control mechanism which

1. allows the security class of an object either to remain constant or to change during execution of a program,

2. can establish the “internal” information flow security of each exported procedure in a module independently of other modules (this indicates that the certification of the entire program is deferred until link time or run time), and

3. is efficient.

1.3 Organization of Thesis

This dissertation consists of six chapters. Chapter 2 describes the overall implementation model of the RM system. We first define the role of each component, the naming convention, and the inter-module and intra-module message flows of the RM system. Simple program examples are shown for easy understanding of the system. We then introduce a class concept, class and instance operations, and inheritance. Finally, we show a prototype implementation model.

The following three chapters are devoted to two common protection issues: access control and information flow control. Chapter 3 describes the access control mechanism of the system. We first illustrate two common implementations of an access control policy: capabilities and access control lists. We then compare capabilities and access control lists and propose an access control mechanism for the RM system.
which has all the advantages of access control lists and yet realizes the principle of least privilege as provided by capability-based systems. Finally, we combine access-rights expressions with the proposed access control mechanism and demonstrate the resulting power.

In Chapters 4 and 5, the proposed information flow control mechanisms are presented. Chapter 4 reviews previous work and introduces a proposed information flow mechanism which is a combination of compile-time and run-time algorithms. The compile-time algorithm establishes the internal information flow security of individual procedures independently of other modules and creates a special data structure for efficient run-time certification of inter-module communication. The run-time mechanism completes the certification of the entire program at message passing time by verifying every information flow caused by exported procedure invocations. Two program examples are shown.

The information flow mechanism proposed in Chapter 4 may have significant run-time overhead. In Chapter 5, we propose a link-time algorithm that uses the data structure created by the compile-time algorithm described in Chapter 4 and certifies the security of the program at link time by verifying all possible information flows which may occur during execution of the program. This mechanism has no run-time overhead, but it does not allow the classes of encapsulated data to change at run time.

Since actual execution paths of a program may not be known at link time, the certification algorithm must account for all possible information flows. Consequently, the algorithm sometimes regards a secure program to be insecure. We therefore propose an improved link-time mechanism which, to some extent, considers the individual execution paths of a program. We also propose a mechanism which is a combination
of compile-time, link-time and run-time approaches and allows a module to contain encapsulated data whose classes change during execution of the program. Several program examples are shown to facilitate understanding the mechanisms.

Chapter 6 summarizes the results of this research and indicates areas of future research.
2 THE RESOURCE MODULE SYSTEM

2.1 An Overview of the Resource Module System

Figure 2.1 illustrates the internal construction of an RM.

2.1.1 Access component

This component is the reason for the existence of the module. It may encapsulate the state values for data or devices along with operations on the value, or it may provide the facility for defining an executable program. More precisely, the access component contains

1. state variables local to the RM,

2. program bodies of exported procedures and local procedures within the RM,

and

3. an initialization procedure which is executed when the RM is first instantiated.

The invocation of an exported procedure or the initialization procedure results in the creation of an independent active entity (in the access component) called a process. Synchronous communication occurs when a process in the access component in one RM invokes an exported procedure in another RM. The calling process is suspended
arriving messages

departing messages

PROTECTION

protection state

SYNCHRONIZATION

synchronization state

ACCESS

resource value

Figure 2.1. Internal Structure of a Resource Module
awaiting a reply. This is similar to a remote procedure call and is implemented in numerous other systems [8,23,28,29,49].

The final version of the specification language for the access component has not been defined. In this research, we assume a PASCAL-like structured language to specify the component.

2.1.2 Synchronization component

As the result of procedure calls made to an RM, numerous processes may be executing concurrently in the access component. The synchronization component controls the initiation of new processes in the access component and manages any necessary blocking and resumption of those processes which invoke exported procedures in other RMs. "Extended open predicate path expressions (EOPPEs)" have been proposed to specify synchronization constraints [30,39]. They are an extension of the very high-level notation of both open path expressions and predicate path expressions [4,9,10]. Semantically, EOPPEs are compatible with highly-parallel applicative languages and architectures. Syntactically, an EOPPE consists of a very-high level path expression which extends to a high-level language in those cases where the very high-level notation lacks the required expressive power.

A discussion of the details of the synchronization specifications is beyond the scope of this research.

2.1.3 Protection component

In order to construct a system for secure resource sharing, powerful protection mechanisms should be provided at a base language level, for (1) strong typing, (2)
information flow control, and (3) access control. In our model, the protection component deals with the above three types of protection on incoming messages. Type checking insures compatibility between the types of the actual parameters and the type specifications of the formal parameters. When a call is made from one RM to another, the types of the actual parameters are packed into the message along with the actual parameter values. The type checking is based on structured equivalence. If a complex type (e.g., structure) is involved, it is transformed into a list of its component types, which are system pre-defined types (integer, real, boolean, character, string etc.) or the identifiers of RM classes. Access control and information flow control mechanisms are described in Chapters 3, 4 and 5.

The protection component also manages the authorization of returns from procedures, called in external RMs, by establishing an expendable "return capability". A return capability avoids an erroneous return message from being received and processed by the module.

2.1.4 Flow of messages

The inter-module and intra-module flow can be described in the following manner. A message, arriving from some external module, is first processed by the protection component of the receiving module. If the message is to invoke a procedure exported by this module, it is checked for the correct designation of the class of the receiver, a valid procedure name and acceptable parameter types. Then, information flow and access control are checked. If the specified protection is violated, the message is

---

1An identifier of an RM is described in Section 2.2.2. A class is described in Section 2.2.1.
rejected; otherwise, the protection component records the reply capability (which is established by the calling module when the invocation was initiated), and forwards the message to the synchronization component. The synchronization component, based on the integrity specification in the EOPPE, may temporarily delay the message. When the integrity specifications are satisfied, the message is forwarded to the access component and the access component initiates a new process to execute the specified exported procedure.

If a message to the module is a return from an exported procedure in another module previously invoked, the expendable reply capability is validated, and type checking and information flow certification of the return values are performed in the protection component. If no error is detected, the message is forwarded to the synchronization component. The synchronization component updates the synchronization state and forwards the message to the access component and the access component resumes the invoking process which has been blocked.

A procedure in the access component may call an exported procedure in another module and reply to a calling module. The resulting message is forwarded to the synchronization component which records the corresponding suspension or possible termination of the calling procedure. The message is then forwarded to the protection component which

1. establishes an expendable reply capability for the called module if it is a request for an external procedure invocation in another module, or

2. finds the corresponding reply capability if it is a reply message.

The protection component also determines the security classes of actual parameters.
Finally, the message is sent to the underlining message passing system for delivery to the destination module.

2.2 Classes, Instances, Names and Types

2.2.1 Class concept and class hierarchy

The concept of a class, along with related ideas, has been used by other researchers [13,26], and we are adopting a similar approach. A "class" describes the implementation of a set of RM s, all of which have the same description of protection (list of exported procedures, although the specifications of access control and information flow control may be different), synchronization and access components. The individual RM s described by a class are called its instances. A class describes the form of its instances' private memory, and how they carry out their operations.

A class itself is an RM and has its own protection component, synchronization component, and access component. The protection component contains protection specifications for some system defined exported procedures. These procedures include such operations as "INSTANTIATE" (to instantiate its instance RM s) and "DESTROY.CLASS" (to destroy itself). The synchronization specifications for these operations are typically null, and the access component has, in addition to a description of the exported procedures, a definition of state variables which serves as template information necessary to create its instances.

A class is an instance of the distinguished class called METACLASS. METACLASS is unique in the system and provides only for the creation of classes. METACLASS, classes, and instances define a three-level tree structure.
2.2.2 Name management

An RM is identified by a site generated unique identifier (ID). The "INSTANTIATE" operation binds an ID to a user defined symbolic name of an instance. Instances of the system defined class DIRECTORY, which we will call directories, are used to maintain the correspondences between symbolic names and their local IDs. This is done by having a directory encapsulate an array of pairs as a state variable. Each pair consists of a symbolic name and the unique ID of the corresponding module. These directories form a UNIX-like tree structured directory system, with each site of the network having a unique root directory RM.

Referring to an RM by name at a remote site requires prefixing the site name to the RM's path name. The site name is resolved locally and the RM name is resolved by the remote directory to form a unique system identification.

Note that within a program, a called module can be represented either by a symbolic name or by a variable whose type is the class of the called module. If a symbolic name is used, the compiler determines the ID of the called module. If a variable is used, the compiler finds the representation of the type of the variable (i.e., the ID of the class of the called module) and the ID of the called module is resolved at run time. In both cases, this is done by consulting directories.

2.2.3 Classes and types

The type of an RM instance is designated by the ID of its class RM, and copies of a class may exist at more than one site with the same ID or different IDs. To provide flexibility, it is desirable that these copies designate the same type. Consider the following example. Assume that a modules M1 is an instances of a class C1, and
both reside at a site $S_1$. Also, assume that a class $C_2$ is a copy of $C_1$ and resides at a site $S_2$. A procedure $P_2$ in a module $M_2$ at $S_2$ specifies a formal parameter of type $C_2$. If a procedure at $S_1$ calls $P_2$ with actual parameter $M_1$, the request message carries $C_1$ as well as $M_1$. Since $C_2$ is a copy of $C_1$, the actual parameter type matches the formal parameter type. However, it is not feasible for the protection component of $M_2$ to keep track of the equivalence relation for classes. In order to maintain this equivalence, a system-wide database is maintained. Each entry in the database is a list of (site identifier, class identifier) pairs of equivalent types. In the above example, the database maintains a list

$$\ldots, (S_1, C_1), \ldots, (S_2, C_2), \ldots$$

When the ID of an RM is carried by a message across site boundaries, the run-time type checking facility will consult this database to determine the equivalence of the actual and formal parameter types.

The database is explicitly managed by the programmer, using METACLASS operations CREATE.CLASS_PUBLIC, COPY_CLASS_PUBLIC, and UNPUBLIC to add/delete class IDs to/from the database.

### 2.3 Program Examples

In this section, several program examples are given to illustrate the utility of RMs. The examples range from very simple programs, which require no explicit resource sharing, to more complex examples where the user explicitly defines shared resources.

Even though the precise syntax for specifying a class has not been defined, we
assume the following:

- A class definition starts with "class name = class".

- The declaration following "specification =" specifies the external interface to this module. Parameters following IN are "call-by-value" and those following OUT are "call-by-result".

- The declaration following "protection =" specifies the default access control for each exported procedure of its instances. This default may be overwritten by different specifications in the instantiate operation or may be modified by the owner of the module.

- The declaration following "synchronization =" defines a synchronization specification.

- The declaration following "access =" specifies the access component.

- The declaration of a variable whose security class is constant, say SECRET, is followed by "of security class SECRET". This is further described in Chapters 4 and 5.

- Procedure declarations following class operations define class operations and those following instance operations define instance operations. Class operations and instance operations are explained in Section 2.4.

- The declaration for a state variable follows "state".

The first example, shown in Figure 2.2, illustrates a program which has no explicit resource sharing. Since instances of the class do not export any procedures, the
P1 = class
    specification = null
    protection = null
    synchronization = null
    access =
        state x : integer;
    class operations
    instance operations
        initialize
        begin
            while not IO.eof do
                begin
                    IO.read(x);
                    IO.write(x+1);
                end
        end
end P1

Figure 2.2.  A Program with No Resource Sharing
parameter type, access control, and synchronization specifications are all null. In this example, integers are read from the default input device and their incremented values are written to the default output device.

The second example shown in Figure 2.3 illustrates the construction of a telephone directory. State variables are an array of directory entries with components for names and telephone numbers (directory), and a pointer which keeps track of the last entry of the directory (num_names). Note that directory and num_names are bound to the constant security class CONFIDENTIAL. The two instance operations on this resource are specified as procedures LOOKUP and ENTER (code left unspecified). Since LOOKUP is a read-type operation and ENTER is a write-type operation, the synchronization of concurrent operations is specified as a readers-writers open path expression. The path expression specifies that, at any point in time, either a single ENTER procedure or a single burst of overlapping LOOKUP procedures may be executing.

The final example, shown in Figure 2.4, illustrates a private mail service, the use of which is restricted to a specified set of users. For simplicity, a fixed number of users, identified by unique (but publicly known) identities 1 ... m, is assumed. An instance MAIL of the class "post.office" is used to encapsulate the identity of queues of modules, one for each user of the mail service. Each member of a queue encapsulates an individual mail message. The latter are instantiations of class "message.bin" and are dynamically created by MAIL and assigned system generated unique names (which we will denote as $Mi). In order to trace the flow of messages, suppose some entity sends mail to user i by invoking the SND procedure in MAIL. MAIL calls on a system defined "INSTANTIATE" procedure to create an instance, say $M3, of class
tel.dir = class
  specification =
    class operations
    instance operations
      LOOKUP(IN name : string, OUT number : integer);
      ENTER(IN name : string, number : integer);
  protection =
    (access control specification) : LOOKUP
    (access control specification) : ENTER
  synchronization =
    path 1 : {{LOOKUP}, ENTER} end
  access =
    type
      entry = record
        u.name : string;
        tel.no : integer;
      end
    state
      directory : array [1..100] of entry
        of security class CONFIDENTIAL;
      num.names : 0..100 of security class CONFIDENTIAL;

Figure 2.3. A Program for a Telephone Directory
class operations
instance operations

procedure LOOKUP(IN name : string; OUT number : integer);
    var ptr : integer; found : boolean;
    begin
        ptr := 1; found := false;
        while (ptr <= num_names) and not found do
            if directory[ptr].uname = name
                then found := true
                else ptr := ptr + 1;
        if found
            then number := directory[ptr].tel_no
            else number := 0;
    end

procedure ENTER( ... ) ...

initialize
    begin
        num_names := 0;
        for i := 1 to 100 do
            end

end tel_dir

Figure 2.3. (continued)
post_office = class
  specification =
    class operations
    instance operations
      SND(IN receiver1 : userid, message : string)
      RCV(IN receiver2 : userid; OUT string)
  protection =
    (access control specification for SND) : SND
    (access control specification for RCV) : RCV
  synchronization =
    state active : array [1..m] of boolean = false;
    start_func
      RCV : active[receiver2] := true;
      SND : active[receiver1] := true;
    end_func
      RCV : active[receiver2] := false;
      SND : active[receiver1] := false;
    path RCV[active[receiver2] = false], SND[active[receiver1] = false]
  access =
    type msg.buf.type =
      record
        first : message.bin;
        last : message.bin;
      end
    state mess.buff : array[userid] of msg.buf.type;
  class operations
  instance operations
    procedure SND (IN receiver1 : userid, message : string);
      var tmp : message.bin.type;
      begin
        message.class.INSTANTIATE(OUT tmp);
        tmp.WRITE(IN message);

Figure 2.4. A Program for a Private Mail Service
if msg_buf[receiver1].first = null
  then
    begin
      msg_buf[receiver1].first := tmp;
      msg_buf[receiver1].last := tmp;
    end
  else
    begin
      msg_buf[receiver1].last.PUTNEXT(IN tmp);
      msg_buf[receiver1].last := tmp;
    end
end

procedure RCV(IN receiver2 : userid; OUT message : string);
var tmp : message.class;
begin
  if msg_buf[receiver2].first = null
    then message := 'NO MAIL'
  else
    begin
      tmp := msg_buf[receiver2].first;
      tmp.GETNEXT(OUT msg_buf[receiver].first);
      tmp.READ(OUT message);
      tmp.DESTROY_INSTANCE();
    end
end

initialize
var i : userid;
begin
  for i := 1 to m do
    begin
      msg_buf[i].first := null;
      msg_buf[i].last := null;
    end
end

end post-office

Figure 2.4. (continued)
message.bin = class
specification =
class operations
instance operations
  WRITE (IN message : string)
  READ (OUT message : string)
  PUTNEXT (IN next : message.bin)
  GETNEXT (OUT next : message.bin)
protection =
  (access control specification for WRITE) WRITE;
  (access control specification for READ) READ;
  (access control specification for PUTNEXT) PUTNEXT;
  (access control specification for GETNEXT) GETNEXT;
synchronization =
  path 1 : (WRITE, READ, PUTNEXT, GETNEXT)
access =
type message.type
  record
    message : string;
    next : message.type;
  end;
state
  mail_message : message.type;
class operations
instance operations
  procedure WRITE(IN message : string);
    begin mail_message.message := message; end;
  procedure READ(OUT message : string);
    begin message := mail_message.message; end;
  procedure GETNEXT(IN next : message.bin);
    begin mail_message.next_nest := next; end;
  procedure PUTNEXT(OUT next : message.bin);
    begin next := mail_message.next_nest; end;
end message.bin

Figure 2.4. (continued)
message.bin. MAIL then invokes $M3.WRITE, requesting $M3 to store the message, and calls $M2.PUTNEXT to accomplish the linking of $M2 to $M3. Finally it changes its internal table to reflect the fact that $M3 is now at the tail of the queue. Suppose user i asks for mail by invoking MAIL.RCV. MAIL retrieves the first item of mail for user i by invoking $M1.READ. It then determines the next member of the queue by invoking $M1.GETNEXT, changes its internal table to reflect that $M2 is now the head of the queue, and calls on a system defined “DESTROY.INSTANCE” procedure to eliminate $M1. Finally, it returns the mail as a message to user i. The path expression guarantees an exclusive access to each entry of “msg.buf” but allows concurrent access mail activity for different “msg.buf” entries [40].

2.4 Class Operations and Instance Operations

Ishikawa and Tokoro classify object-oriented systems into two models: the model of an object as encapsulated data and the model of an object as a computational entity [32]. In the first model, an object is protected data. Even if a subject owns the identifier of an object, it cannot access the internal structure of the object directly (the object is “sealed”). The internal structure of an object can be manipulated only through invocation of a procedure which is declared as an exported procedure in the class of the object (the object is “unsealed”). Conceptually, exported procedures reside in the class in which these procedures are defined. Thus, in order for a subject to access its object, the subject sends a message with an invocation request to the class of the object along with the identifier of the object and a requested procedure name as parameters. Since a message to invoke an exported procedure is sent to the class, this type of exported procedure is called a “class operation”. To create a new
object, a subject sends a message with a creation request to the class. This object model is used by CLU, ALPHARD and HYDRA [32].

For example, suppose class MATRIX defines the structure of a two-dimensional array and the exported procedure "+'", which adds each element of two arrays. Assume variables "a" and "b" hold identifiers of instances of MATRIX. If statement "c := a + b" is executed, a message is sent to class MATRIX with "+'", "a" and "b" as parameters as shown in Figure 2.5. Exported procedure "+'" can unseal "a" and "b" and manipulate each element of "a" and "b" because "a" and "b" are instances of MATRIX. A new object is created as the result of adding the matrices. Finally, the newly created object is sealed, and its identifier is sent back to the subject and stored in variable "c".

In the second model, an object encapsulates data along with exported procedures. A class has a template necessary to instantiate an object and the operation to create instances. Conceptually, exported procedures reside in each instance object whose class defines these procedures. In order for a subject to access its object, it sends a message with an invocation request to the object along with the requested procedure name. Since a message to invoke an exported procedure is sent to an instance of the class which defines the procedure, this type of procedure is called an "instance operation". To create a new object, a subject sends a message to the class to invoke the creation procedure.

Now consider the same example as described above. Since the "+'" operation resides in each object, the execution of "c := a + b" causes a message to be sent to object "a" with "+'" and the identifier "b" as parameters as shown in Figure 2.6. When object "a" receives the message, it creates a new object which is the result of
Figure 2.5. An Object as Encapsulated Data
Figure 2.6. An Object as a Computational Entity
adding each element of its internal data to the corresponding element of "b" and the identifier of the newly created object is sent back to the subject. This object model is used by SMALLTALK, LOOPS and ACTOR [32].

An RM is classified as a computational entity (i.e., the second model). A module totally encapsulates state variables along with exported procedures manipulating these variables, and a message to invoke an exported procedure is sent to an instance module (not to a class). However (as in SMALLTALK), the system also provides essence of the first object model by allowing users to define class operations (in addition to system defined default operations such as "INSTANTIATE" and "DESTROY_CLASS"). However, since the system does not provide the seal and unseal operations, exported class operations must invoke exported instance operations of these modules that are passed as parameters in order to access their internal data.

In order to implement the above example, there are two possibilities. Figure 2.7 shows the case in which exported procedure "plus" is implemented as an instance operation. Suppose "a" and "b" are instances of class MATRIX. The statement "a.plus(IN b, OUT c)" adds each element of matrices "a" and "b" and stores the identifier of the resulting module in "c". Figure 2.8 shows the second case (exported procedure "plus" is implemented as a class operation). In order to access each element of an instance of the MATRIX class, instance operations "get" and "put" are provided. The statement "MATRIX.plus(IN a, b, OUT c)" adds "a" and "b", and stores each result of addition into "c". "MATRIX.plus" invokes the "get" operation on formal IN parameters "x" and "y" to access each element of the modules passed as parameters. It also invokes the put operation on formal OUT parameter "z" to store a resulting value in each element of "c". The choice of whether an instance operation or a class
MATRIX = class
    specification =
      class operations
      instance operations
        procedure get(IN i, j : integer; OUT x : real);
        procedure put(IN i, j : integer; x : real);
        procedure plus(IN b : MATRIX; OUT c : MATRIX);
    protection = ...;
    synchronization = ...;
    access =
      state A : array [1..5, 1..5] of real;
    class operations
    instance operations
      procedure get(IN i, j : integer; OUT x : real);
        begin x := A[i, j]; end;
      procedure put(IN i, j : integer; x : real);
        begin A[i, j] := x; end
      procedure plus(IN b : MATRIX; OUT c : MATRIX);
        var i, j : integer; x : real;
        begin
          MATRIX.INSTANTIATE(OUT c);
          for i := 1 to 5 do
            for j := 1 to 5 do
              begin
                b.get(IN i, j, OUT x);
                c.put(IN i, j, x + A[i, j]);
              end;
            end;
        end MATRX

Figure 2.7. A Program for MATRIX without Class Operations
MATRIX = class
  specification =
    class operations
      procedure plus(IN a, b: MATRIX; OUT c : MATRIX);
    instance operations
      procedure get(IN i, j : integer; OUT x : real);
      procedure put(IN i, j : integer; x : real);
  protection = ...;
  synchronization = ...;
  access =
    state A : array [1..5, 1..5] of real;
  class operations
    procedure plus(IN a, b : MATRIX; OUT c : MATRIX);
      var i, j : integer; x, y : real;
      begin
        MATRIX.INSTANTIATE(OUT c);
        for i := 1 to 5 do
          for j := 1 to 5 do
            begin
              a.get(IN i, j, OUT x);
              b.get(IN i, j, OUT y);
              c.put(IN i, j, x+y);
            end;
      end;
    instance operations
      procedure get(IN i, j : integer; OUT x : real);
        begin x := A[i, j]; end;
      procedure put(IN i, j : integer; x : real);
        begin A[i, j] := x; end
    initialize
      begin
      end
  end MATRIX

Figure 2.8. A Program for MATRIX with Class Operations
operation is used depends on the situation and the implementer's preference.

2.5 Inheritance

We are adopting the concept of inheritance in the RM system [13,26]. A class can be defined to be a modification of a previously defined class called its superclass. The newly defined class is called a subclass of the superclass and inherits attributes from the superclass. The subclass can inherit attributes from more than one independent superclass and be a superclass of other classes thereby allowing an arbitrary directed acyclic graph structure of superclass-subclass relations.

In some object-oriented languages such as SMALLTALK, a subclass inherits all state variables and exported procedures of the superclass. In general, the subclass can also override a definition of an exported procedure in the superclass by defining a new exported procedure with the same name. This approach raises a problem in the RM system since concurrent access of state variables, via exported procedures, might be allowed. Concurrent access of the state variables of any instance may be restrained via an EOPPE in a synchronization component. If new exported procedures are defined which access state variables declared in the superclass, concurrent access problems may arise unless the EOPPE is redefined for the subclass. This new EOPPE would have to include not only the newly declared exported procedures but also all exported procedures declared in all of the ancestor superclasses. If there are many ancestor superclasses or there are many procedures declared in ancestor superclasses, it might be difficult to redefine the EOPPE. Thus, a different approach is required in our system.

In an EOPPE, all of the procedure names to be exported must appear. There are,
however, often cases in which some exported procedures in the ancestor superclasses may not need to be inherited by the subclass. Thus, our system assumes the default of not inheriting exported procedures from its superclasses. Procedures to be inherited from the superclasses must be explicitly specified and a new EOPPE in the subclass must be defined only upon those procedures inherited from the superclasses and those procedures newly declared in the subclass. We call this approach "multiple partial inheritance".

An example of inheritance is shown in Figure 2.9. M is specified to be a superclass by M'. O1 is inherited from M, procedure O2' replaces O2 (simply by not inheriting O2 and defining a new O2), and O3 and R2 are newly defined in M'. O2' and O3 may access R1 ∪ R2. In this way, the class M' builds on selected parts of the class M. The synchronization component must be redefined in M' for O1, O2' and O3.

The system defines the distinguished class RESOURCE. Every class but RESOURCE and METACLASS directly inherits the attributes of RESOURCE by default. The class operations defined in RESOURCE includes "INSTANTIATE" and "DESTROY-CLASS". The instance operations defined in RESOURCE includes "DESTROY-INSTANCE".

The RM system is strongly typed. This requires some special facilities since strong typing can severely degrade the flexibility of inheritance. For example, suppose we want to simulate a traffic system by realizing cars, people and a signal in object forms. People and cars waiting at the signal are represented by instances of "PERSON" class and instances of "CAR" class and are linked together to form "people queue" and "car queue", respectively. We may want to have two levels of an inheritance hierarchy. The first level is a LINK class to allow construction of gen-
\[ M' = \text{class specification} = \]
\[ \text{superclass} = M \]
\[ \text{protection} = \]
\[ \text{(access control specification for } O1) \ O1 \]
\[ \text{(access control specification for } O2') \ O2' \]
\[ \text{(access control specification for } O3) \ O3 \]
\[ \text{synchronization} = \text{"new specs for } O1, O2', O3; \]
\[ \text{access} = \]
\[ \text{state } R2 \]
\[ \text{class operations} \]
\[ \text{instance operations} \]
\[ \text{procedure } O1(\ldots) : \text{inherit from } M \]
\[ \ldots \]
\[ \text{procedure } O2'(\ldots) \]
\[ \ldots \]
\[ \text{procedure } O3(\ldots) \]
\[ \ldots \]
\[ \text{initialize} \]
\[ \ldots \]
\[ \text{end } M' \]

Figure 2.9. Structuring Software Using Inheritance
veral linked list structures. The LINK class has a state variable "NEXT" to store a pointer to the next element (RM) in a linked list. Classes in the second level of the inheritance hierarchy are "CAR" and "PERSON" which inherit the LINK class. In this structure, however, a type of NEXT is not known at the LINK level. It must be of type "CAR" to be inherited by "CAR", or of type "PERSON" to be inherited by "PERSON".

Many object-oriented systems are dynamically typed, thereby avoiding this problem. In order to maintain the flexibility of inheritance in a strongly typed circumstance, we use the "traits" concept introduced in the XEROX STAR workstation [12] or the "abstract superclass" concept in SMALLTALK [26]. Traits are introduced to optimize compiled code, and abstract superclasses are introduced to construct mutual superclasses for several classes which share a part of their descriptions and yet none is properly a subclass of another. These special classes can be inherited by other classes but cannot be instantiated. In the RM system, a special class called a "trait.class", which does not have instantiation power, is introduced for those cases where strong typing restricts flexibility. A trait.class is allowed to use a DUMMY.RM in its variable declaration to indicate that the actual class will not be known until inheritance takes place, at which time a DUMMY.RM will be replaced by an actual class name.

The solution to the above example is shown in Figure 2.10.

2.6 Implementation of the RM System

In this section, we suggest a possible implementation of a prototype system.

An outline of the support system is shown in Figure 2.11. The architecture layer provides the following execution environment facilities:
LINK = trait.class
  specification =
    class operations
    instance operations
      procedure PUTNEXT(...)
      procedure GETNEXT(...)

  access =
    state
      NEXT : DummyRM;
  class operations
    instance operations
      procedure PUTNEXT(...);
      ...
      procedure GETNEXT(...);
      ...
    ...
end LINK

PERSON = class
  specification =
    ...
  superclass = LINK
    ...
  access =
    state
      NEXT : DummyRM = PERSON;
  class operations
    ...

Figure 2.10. A Program for Traffic Simulation Using a Trait
instance operations

    procedure PUTNEXT(...);
    inherit from LINK;
    procedure GETNEXT(...);
    inherit from LINK;

end PERSON

CAR = class
    specification =

    superclass = LINK

    access =

    state
        NEXT : DummyRM = CAR;

class operations

    instance operations

    procedure PUTNEXT(...):
    inherit from LINK;
    procedure GETNEXT(...):
    inherit from LINK;

end CAR

Figure 2.10. (continued)
Figure 2.11. Prototype Support Software for Implementing Resource Modules
1. the management of the run-time stacks for the processes in the RMs,

2. RM memory management (relocation, garbage collection, main/secondary memory management),

3. the management of the execution of each component in an RM and the inter/intra-module communication, and

4. basic type operations (e.g., integer, boolean, real and character operations, etc.) and basic I/O operations.

Porting an implemented system from one machine to another would require rewriting the architecture layer.

The kernel implements the distinguished classes METACLASS and RESOURCE. The system layer is where system defined classes and some system instances exist. This layer contains classes and instances of high-level I/O devices and the RM system compiler, the directory class, and the file class.

The highest layer is the application layer, where all user defined RMs exist. In addition to the standard directory and file objects, provided by the system layer, the user has, at his disposal, a virtual address space of objects of his own choosing.

Each component of a module (protection, synchronization and access) may be implemented by a separate process, called a manager process in order to differentiate them from processes created as a result of procedure invocations. Since a manager process in each component can access only its own state, each may be executed on a separate processor. Furthermore, since the synchronization component regulates concurrent accesses of user processes to state variables encapsulated in the access
component, user processes may run on separate processors which access a shared memory storing state variables.
3 THE ACCESS CONTROL MECHANISM

3.1 Access Matrix Model

Access control regulates the authorization of access to objects based on the identity of subjects. The access matrix model was developed as a framework for describing protection systems [27,36]. The model is defined in terms of a triple \((S, O, A)\) where

1. \(S\) is a set of subjects which are active entities that access objects;
2. \(O\) is a set of objects to which access must be controlled; and
3. \(A\) is an access matrix in which rows correspond to subjects and columns correspond to objects, where an entry \(A[s,o]\) stores the access rights of subject \(s\) to object \(o\).

In operating systems, the objects may be files, directories, memory segments or processes, and the subjects may include users or processes. Typical access rights are read, write, search (in the case of directory) and execute rights. At any point of execution, a subject is associated with a protection environment called a domain. A domain is a set of objects which the subject is authorized to access; thus, a domain may be represented by a list of objects and the corresponding access rights that the subject currently has to these objects.

In object-oriented systems, two levels of access rights must be considered.
1. encapsulation of objects: read/write access authorizations of exported procedures to object variables, local variables and parameters (if basic types are passed as parameters); and

2. procedure invocation rights: rights of subjects to invoke exported procedures of an object.

For example, the access rights for an object of type integer_stack may be invocation rights to push and pop procedures from the user's point of view. However, the pop and push procedures have access authorization only to the internal representation of the stack (which may be an array of integer and a pointer to the array), and an integer parameter to be pushed on to the stack (in the case of the push procedure).

This research focuses attention on the representation of invocation rights to exported procedures. We will not consider mechanisms for encapsulation of objects. The RM system assumes an underlying layer to implement this latter feature.

In this view, the protection state of the system (S, O, A) may be restricted in the following way.

1. S is a set of subjects, which may be users.

2. O is a set of modules in the system.

3. A is an access matrix of which entry A[s,o] lists the exported procedures of module o which subject s is authorized to invoke.
3.2 Access Control Mechanisms

Implementation of a protection mechanism requires complete mediation of every access to an object by a reference monitor [3,16]. In RM systems, the reference monitor is the code of the protection component and implemented in the kernel. The protection component intercepts every message coming to a module. The reference monitor checks not only access authorization but also information flow and parameter type compatibility. The details of the mechanism for information flow control will be discussed in Chapters 4 and 5.

Conceptually, in order to implement access control mechanisms, the reference monitor needs to refer to the access matrix. Direct realization of the access matrix as a two-dimensional structure is impractical because of the sparseness of the matrix. Instead, the access matrix is retained in either a row-based or column-based representation in real implementations. Two types of implementations are widely used: capabilities and access control lists (ACLs).

Conceptually, a capability-based system is an implementation of the row-based representation. Each subject “s” holds a list of pairs (o, A[s,o]) called a capability, where A[s,o] is a nonempty entry of the access matrix. Thus the list, called a capability list or C-list, corresponds to a single row of the access matrix and defines the domain of the subject. An important assumption is that each object in the system has an unforgeable unique identifier, and the capability (o, A[s,o]) itself is unforgeable. The capability is like a ticket in that possession authorizes its holder “s” to access object “o” with rights specified in A[s,o]. Under our assumption, the A[s,o] entry of capability (o, A[s,o]) lists the exported procedures of module “o” which “s” is authorized to invoke.
An access control list is an implementation of the column-based representation. Each object stores a list of pairs \((s, A[s,o])\), where \(A[s,o]\) is a nonempty entry of the access matrix. Thus a list, called an access control list or ACL, corresponds to a single column of the access matrix. Implementation requires that each subject in the system be associated with an unforgeable unique identifier \(s\), called the subject ID, and each entry in an access control list \((s, A[s,o])\) itself be unforgeable. For each object \(o\), only subject \(s\) whose id is on the access control list is allowed access to \(o\) with rights given in \(A[s,o]\).

### 3.3 Comparison of Capabilities and Access Control Lists

There are several well-known protection problems to be considered when comparing access control mechanisms [11,21,45]. In the following subsections, we will introduce these problems and compare capabilities with access control lists in order to make clear the advantages and disadvantages of each implementation.

#### 3.3.1 Efficiency

In a capability-based system, possession of the capability \((o, A[s,o])\) authorizes subject \(s\) to access object \(o\) with rights given in \(A[s,o]\). Thus, the reference monitor only compares \(A[s,o]\) with the access request. The \(A[s,o]\) part of the capability \((o, A[s,o])\) is usually represented by a bit vector. Therefore, testing whether a capability contains a certain right requires a simple bitwise logical comparison.

On the other hand, an ACL implementation requires the reference monitor to search the list for every access request to verify the access authorization. Since a list search tends to be much slower than a simple bitwise logical comparison, the ACL
approach is much more inefficient than the capability-based approach. In particular, if the method is applied to such objects as memory segments, the ACL approach may be unrealistic. In RM systems, however, the list search is necessary only when a message arrives at a module and therefore differences in the costs of these two implementations may not affect system performance significantly as compared to applications with frequently accessed objects.

3.3.2 Ownership

The owner of an object has the exclusive authority to grant/revoke access rights to the object to/from other users, and to delete the object. Usually, the creator of an object becomes the owner of the object unless he transfers the ownership of the object to another user.

The strict ownership policy has advantages and disadvantages. The creator of a object may want total control of his object. However, users of the object may want a guarantee that their rights to access the object will not be revoked by the owner after they have written some valuable information into the object. The existence of owners is a philosophical problem. Some implementations are based on an ownership policy [44]. Others deny the existence of an exclusive owner [11].

ACL systems are suitable for implementing an ownership policy. Typically, the owner of an object is the only one who can modify the access control list of the object. Capability-based systems, on the other hand, have a diffuse concept of ownership. Any holder of a capability for an object shares control of the object according to the access rights to the object. This diffuseness of ownership is one of the reasons for the difficulty of solving, in capability-based systems, the security problems discussed
in the following two subsections (revocation of access and propagation of access). Because of this, it appears necessary to us to introduce a clear ownership policy. Note that even with a strict ownership policy, the existence of a special user with no access restrictions (even the right to modify access control states) similar to the UNIX super user is useful. For example, the owner of an object may leave the organization in which the system exists, while other users still want to utilize his objects.

3.3.3 Revocation of access

In a capability-based system, if subject S1 wants to share object O1 with subject S2, S1 can give the capability for O1 to S2. In an ACL system, sharing can be done by modifying the ACL associated with O1. If S1, who is the owner of O1, adds user identifier of S2 to the ACL of O1, S2 can access O1. In an ACL system, in order for S1 to revoke S2's access to O1 later, S1 can simply delete the entry for S2 from the ACL associated with O1.

However, a capability-based system does not provide an easy way to revoke access once it has been given to other subjects. Redell proposed a solution to this problem by using indirect addressing [43]. Instead of giving the capability for O1 to S2, S1 creates an new object O2 that stores a capability for O1 and gives a capability for O2 to S2. S2 can access O1 indirectly though O2. Revocation is accomplished by deactivating or removing O2. If S1 gives access for O1 to several subjects, S1 has to construct a separate indirect object for each different subject in order for S1 to have the ability to selectively revoke access for O1. In spite of this solution, a capability-based approach has less control over revocation of access and is more dangerous. If S1 did not make an indirect object and passed the original capability for O1 to S2 by mistake, there is
no way for S1 to revoke S2's access for O1 unless S1 destroys O1.

3.3.4 Propagation of access

In the above discussion, after S2 acquires access for O1, S2 may pass the access rights to a third subject S3 without asking S1 for permission. This cannot happen in an ACL system. In order for S2 to pass the access rights for O1 to S3, S2 has to modify the ACL of O1 to include S3's entry. However, S2 is not the owner of O1 and cannot modify the ACL of O1 even though S2's entry is in the ACL.

In capability-based systems, controlling propagation of access is more difficult. S2 can give a copy of the direct or indirect capability for O1 to S2 without S1's permission. Some capability-based systems provide for limited mechanism to control propagation. Hydra's solution to this problem is to introduce the ENVRTS right in a capability [11]. A capability may be stored in an object only if the capability contains ENVRTS right. If S1 stores a capability for O1 in O2 without ENVRTS right and grants S2 indirect capability to O2, S2 cannot store the capability for O1 in another object, say O3, in order to give S3 an indirect access to O1. Note that S1 must be sure that the capability for O2 in the domain of S2 also does not contain ENVRTS. Otherwise, S2 can pass its capability for O2 to S3 and S3, in turn, acquires an indirect access to O1. Even with the ENVRTS feature, the capability-based system provides for less control of the propagation of access than that provided by ACL systems. S1 must consider S2's capability-list before giving access to S2.
3.3.5 Review of access

A subject may sometimes want to know the list of all subjects who have access to its object O1. This can be done very easily in an ACL system by searching the list associated with O1. On the other hand, a capability-based system requires the searching of every capability-list in the system in order to determine who can access O1. The search must account not only for direct capabilities but also for indirect capabilities.

3.3.6 Least privilege

A Trojan Horse is a program which performs functions other than that specified by the program. For example, a Trojan Horse compiler may copy or destroy files in the user's domain which are not relevant to the given source file. In order to protect objects or minimize damage from Trojan Horses or undebugged programs, each program should be executed in the smallest domain necessary for the task. This principle is called "least privilege" or "need-to-know".

Since capabilities naturally provide mechanisms to change a domain dynamically for each procedure invocation, the capability-based systems nicely implement the least privilege principle at the procedure invocation level. A principal feature to realize this mechanism is based on an "enter" capability [19] or, similarly a "call" operation and CALLRTS access rights [11]. The following paragraphs describe how Hydra's call operation realizes the least privilege principle at the procedure invocation level.

3.3.6.1 A procedure invocation mechanism in Hydra Each procedure in Hydra has its own C-list, which includes capabilities for the procedure's code (with
read and execute rights set), capabilities for local objects (with probably read and write rights set) and templates for the parameters. There are two types of templates; parameter templates and amplification templates.¹ A template does not reference a specific object but is considered as a prototype capability for all objects of a given type. The purpose of a parameter template is for type checking of actual parameters; thus it contains a type field and a required-rights field. The type and rights field of a capability passed as an actual parameter are compared with those of the corresponding template. If the type fields match and the rights field of the capability is a subset of that of the template, the actual parameter is valid. An amplification template contains a type field, a required-rights field and a new-rights field. The type field and the rights field of the capability passed as an actual parameter are validated in the same way as a parameter template. If valid, a new capability is produced with the rights specified in the new-rights field of the template.

In order to invoke a procedure, the currently executing domain (C-list) must contain the capability for the procedure with its CALLRTS bit set. A call operation requires capabilities for a procedure and capabilities and rights masks for objects passed as actual parameters. For each invocation, a new C-list is created based on the C-list of the invoked procedure. Each C-list slot, that corresponds to a parameter template or an amplification template, is filled with the matching capability passed as a parameter. In the case of a parameter template, the resulting capability of the newly created C-list holds the rights field restricted by the associated mask. Therefore, to some extent, the caller can restrict the usage of objects passed as a parameters from the

¹Actually there are three types. However, a creation template is excluded from our discussion, since it has nothing to do with implementing the least privilege principle.
invoked procedure. For an amplification template, the resulting capability holds the rights field specified by the new-rights field. An amplification template is a powerful tool for data encapsulation of an abstract data type.

If a subject S instantiates object X of class C, the system gives S a capability for X with read/write bits off as well as capabilities for a subset of exported procedures of C with CALLRTS bits on. S then may call these exported procedures with X as a parameter. The C-list corresponding to these exported procedures has amplification templates for their instances to be passed as parameters (in this case, X) with read/write bits in their new-rights field. Therefore, S cannot read or write from/to X directly, but the exported procedures can read or write from/to X only when these procedures are invoked and X is passed as a parameter (encapsulation of objects).

Hydra does not distinguish between the two levels of protection: encapsulation of objects and procedure invocation rights.

3.3.6.2 Implementation of least privilege in capability-based systems
From the above discussion, it is now clear how a capability-based system can nicely realize the least privilege principle. For each invocation, a totally new C-list (an execution environment) is created, and upon termination of the procedure, the corresponding C-list is erased. The C-list associated with a procedure invocation contains only capabilities necessary to do the task and a caller can mask out those rights bits of capabilities that are irrelevant. More detail is shown by the following example.

Assume that object HEALTH stores medical information and object SALARY stores income information about a community of users. Exported procedures are GET and PUT for retrieving or inserting records to/from these objects. INSURANCE
and TAX are other objects. One of the exported procedures HEALTH.CHECK in INSURANCE takes object HEALTH as a parameter to determine the current health status of a user. Exported procedure FEDERAL.TAX.CALC in TAX takes object SALARY as a parameter to calculate the federal tax of a user.

There are two protection issues related to the least privilege principle.

1. Users should not be able to access data stored in HEALTH and SALARY objects even though users have capabilities for (or can locate) these objects. GET and PUT procedures of these objects, thus, should not be invoked directly by users. This may sound like a protection issue of encapsulating objects as discussed earlier, since this issue may be avoided if a programmer combines object HEALTH and SALARY into INSURANCE and TAX objects, respectively. However, there may be other totally independent objects which require access to HEALTH and SALARY objects and a decision of dividing the problem into objects is totally up to the programmer. Thus, this kind of situation must also be considered as an issue of procedure invocation rights.

2. Procedure HEALTH.CHECK should have call access to only the GET procedure of HEALTH when it is invoked with HEALTH as a parameter. It should not have access authorization to PUT of HEALTH and GET/PUT of SALARY. Similarly, FEDERAL.TAX.CALC should have call access to only the GET procedure of its parameter SALARY object and should not have access to PUT of SALARY nor any access to HEALTH.

The above scenario can easily be implemented with capability-based systems as follows:
• HEALTH.CHECK procedure in object INSURANCE holds an amplification template whose type field is the class of HEALTH and the new-rights field contains a call right for GET;

• The FEDERAL.TAX.CALC procedure in object TAX holds an amplification template whose type field is the class of SALARY and the new-rights field contains a call right for GET; and

• The C-list of a subject contains capabilities for the HEALTH and SALARY objects with no GET/PUT access rights, a capability for the INSURANCE object with CALLRTS access right for HEALTH.CHECK and a capability for TAX with CALLRTS access right for FEDERAL.TAX.CALC.

Even though the subject holds capabilities for the HEALTH and SALARY objects, it cannot invoke their exported procedures PUT and GET directly because it does not have call rights for these procedures within the capabilities. If the HEALTH.CHECK procedure is invoked by the user with a capability for the HEALTH object passed as a parameter, a new C-list is created which contains a capability for only the HEALTH object with an amplified call right for the GET procedure. Thus, the invoked procedure has access authorization only to the state variables of the INSURANCE object and procedure GET in the HEALTH object. Similarly, if FEDERAL.TAX.CALC is invoked by the user with a capability for SALARY as a parameter, a newly created C-list contains a capability for only the object SALARY with an amplified call right for the GET procedure.

3.3.6.3 Implementation of least privilege in ACL systems

Now we will consider an implementation of the least privilege principle with an ACL system. In
an ACL system, a subject usually corresponds to a user. Under this assumption, the same level of the least privilege principle as provided by capability-based systems cannot be achieved. In the previous example, in order for the HEALTH.CHECK procedure executing on behalf of user S to acquire a call access authorization for GET in the HEALTH object, the ACL for HEALTH must contain the (S, GET) entry. Similarly, SALARY must contain the (S, GET) entry in its ACL in order for FEDERAL.TAX.CALC executing on behalf of S to access GET in SALARY. This allows S to invoke GET procedures in HEALTH and SALARY directly. Moreover, HEALTH.CHECK can access not only GET in HEALTH but also GET in SALARY. In the same manner, FEDERAL.TAX.CALC has access authorizations for GET in both SALARY and HEALTH.

One way of realizing the least privilege principle in ACL systems is the concept of a group [41]. A group is a mechanism for representing a set of subjects; thus, multiple subjects are identified by a group name. The group mechanism in Multics (called projects) allows a subject to be active in only one group at a time, even though it can be a member of more than one group. The possible improvement in the above example may be as follows:

- User S is a member of two different groups, say G1 and G2;
- The ACLs for HEALTH and INSURANCE contain entries (G1, GET) and (G1, HEALTH.CHECK), respectively; and
- The ACLs for SALARY and TAX contain (G2, GET) and (G2, FEDERAL_.TAX.CALC), respectively.

Then, HEALTH.CHECK, when executing on behalf of S, can only access GET in
HEALTH, and FEDERAL.TAX.CALC, executing on behalf of S, has access only to GET in SALARY. However, even with this mechanism, S can still access GET in HEALTH or SALARY directly. Moreover, changing the active group in Multics requires users to log off the system and log in again under a new group identifier. ACL systems need a mechanism to change domains dynamically for each procedure invocation in order for the systems to realize the same flexibility of implementation of the least privilege principle provided by capability-based systems.

3.4 Proposed Access Control Mechanism

3.4.1 Four-tuple ACL entry

As discussed in the previous section, capability-based systems have difficulties in the following areas: revocation of access, control of propagation of access, and review of access. ACL-based systems solve all these problems nicely, but are inefficient and cannot implement the least privilege principle as effectively as capability-based systems.

As mentioned by other researchers [21,32,45], the problems of capability-based systems are fundamental and it is difficult to construct secure systems based on them. These problems come from the fundamental characteristic of capability-based systems — diffuseness of clear ownership. On the other hand, the inefficiency of ACL-based systems, as mentioned earlier, may not be a big obstacle if the mechanisms are applied at places where less frequent traffic is expected. The use in RM systems is considered to be in this category, because access control checks are done only when messages arrive at a module. Thus, we apply an ACL approach to RM systems to implement
discretionary protection for individual users to invoke exported procedures of each
module. However, as discussed in the previous section, the choice of mechanisms to
encapsulate states variables within a module is left to the implementor. Thus, this
discussion does not preclude capabilities to implement protection at lower levels.

In order to enforce the principle of least privilege as provided by capability-based
systems, we use a four-tuple entity in the access control list. Each subject “s” in an
ACL entry (s, A[s,o]) is represented by a four tuple, (user ID, class ID, module ID,
exported procedure name), not just by a simple user ID. In ACL systems, changing
a subject ID is the essence of changing domains, because subject IDs are in one-to-
one correspondence with their domains [45]. The four-tuple uniquely determines the
exported procedure in the system which is currently under execution on behalf of the
user. A subject specified by a four tuple dynamically changes its ID for each procedure
invocation, thereby realizing the least privilege principle at the procedure invocation
level. This is the same level of flexibility offered by capability-based systems.

From another point of view, the four-tuple gives users a fine grain of access
control. The first component provides access control at the user level. The second
component may be used to enforce access through a front end module of a specified
class. This is useful in presenting a “view” for the user, as in database systems, and
preventing direct access which would circumvent the view. Since numerous instances
of the same class may exist in the system, the third component of an entity may
be used to further qualify the second component and force access through a specific
instantiation of a designated class. The fourth component allows fine grain control by
forcing the calling module through one of its specific procedure. These four levels are
a natural feature of the system. Since owners of classes, owners of instantiations, and
users of instantiations may all be different, the four levels of control provide significant flexibility in structuring and specifying protection for a system of modules. An "*" may be used in any component of a four tuple to denote a wild card (don't care) specification in the case the owner of a module does not need this level of fine grain of control.

The example discussed in the previous section can be modified as follows by using the four-tuple ACL:

- The ACL of object HEALTH contains the entry ((S, INSURANCE.CLASS, INSURANCE, HEALTH.CHECK), GET);
- The ACL of SALARY contains the entry ((S, TAX.CLASS, TAX, FEDERAL.TAX.CALC), GET);
- Object INSURANCE has the ((S, *, *, *), HEALTH.CHECK) in its ACL; and
- The TAX object contains the ((S, *, *, *), FEDERAL.TAX.CALC) in its ACL.

Then, S cannot access the GET procedure in the HEALTH or SALARY object directly. Also, on behalf of S, procedure HEALTH.CHECK has call access to only the GET procedure of object HEALTH, and procedure FEDERAL.TAX.CALC has call access to only procedure GET of the SALARY object. In this solution, S can invoke HEALTH.CHECK in INSURANCE and FEDERAL.TAX.CALC in TAX from any procedure of any instance of any class.

Note that the rights amplification mechanism which capability-based systems provide is not necessary in the four-tuple ACL, since for each procedure invocation,
a user may obtain a totally different execution environment from the previous environment by changing its subject ID. In the above example, the purpose of the rights amplification of capability-based mechanisms is to prevent a user from direct access to the HEALTH and SALARY objects, and to provide access to these objects only through invocations of the FEDERAL.TAX.CAL and HEALTH.CHECK procedures. This can be easily done in the four-tuple ACL by specifying procedure names FEDERAL.TAX.CALC and HEALTH.CHECK in the fourth elements of the ACL entities in SALARY and HEALTH. In this example, the names of the modules encapsulating these procedures and their classes are also specified in the second and third elements of the ACLs to avoid name conflicts.

In order to illustrate that the four-tuple ACL has the power to realize the least privilege principle, we will show a possible four-tuple ACL implementation of the bibliography example, which was introduced in [50]. The bibliography system provides for the following four procedures:

1. \( U(p_1, \ldots, p_n) \) — update
2. \( P(p_1, \ldots, p_m) \) — print
3. \( PWOA(p_1, \ldots, p_0) \) — print without annotation
4. \( E() \) — erase

where \( p_i \)'s are parameters.

There are five instances of bibliography, B1, B2, B3, B4 and B5, and three users, U1, U2 and U3. Each user has the following access rights:

1. U1 has capabilities with CALLRTS for all of the procedures U, P, PWOA and E. U1 also has a capability for B1 with U, P, PWOA and E rights and a capability
for B2 with U and PWOA rights.\textsuperscript{3} Thus, U1 may invoke all those operations on B1, but can only perform U and PWOA on B2.

2. U2 has capabilities for all of the procedures and three bibliographies B2, B3 and B4. The capabilities for the procedures have CALLRTS. The capability for B2 has only a PWOA right, but the capabilities for B3 and B4 have U, P and E rights. Thus, U2 can invoke only PWOA on B2, but may perform U, P or E on B3 and B4.

3. U3 has capabilities with CALLRTS for U, P and PWOA. It has also capabilities for B1 with P right, B4 with U and P rights, and B5 with U, P and E rights. Therefore, U3 may invoke U or P on B4 and B5, and P on B1. Note that even though it has a capability for B5 with right to invoke E, U3 cannot actually perform the operation since it does not have a capability for E.

With the four-tuple ACL in the RM system, the solution may be as follows. Since a module totally encapsulates data along with operations on the data, users need to know only the references (unique identifier) to the bibliography objects, which are the instances of class BIB\_CLASS. The five instances of BIB\_CLASS are B1, B2, B3, B4, and B5. Rights to invoke exported procedures U, P, PWOA and E in those instances are specified by the ACLs. The ACL of each module contains the following entries:

1. \(B1\) — \(((U1,*,*,*) \{U, P, PWOA, E\})\) and \(((U3,*,*,*) \{P\})\).

\textsuperscript{3}In Hydra, procedures are also objects. Thus, in order for a user to invoke procedure P on data object O, the user must hold a capability for P with access rights CALLRTS, as well as a capability for O with access rights P.
2. \( B_2 \) — \(((U_1,*,*,*) \{ U, PWOA \}) \text{ and } ((U_2,*,*,*) \{ PWOA \})\).

3. \( B_3 \) — \(((U_2,*,*,*) \{ U, P, E \})\).

4. \( B_4 \) — \(((U_2,*,*,*) \{ U, P, E \}) \text{ and } ((U_3,*,*,*) \{ U, P \})\).

5. \( B_5 \) — \(((U_3,*,*,*) \{ U, P \})\).

This realizes precisely the same access environment for \( U_1, U_2 \) and \( U_3 \) as provided by Hydra.

In order to modify an entry of an ACL, a subject must hold the "ACL modification right" to the module. An instantiator of a module automatically becomes the owner of the module and by default is granted the ACL modification right to the module. Each module has only one owner. An owner can transfer the ownership of a module to another subject. However, once transferred, the ownership can not be regained unless a new owner transfers the ownership to the old owner. In addition to the owner of a module, two sets of subjects have rights to modify an ACL of a module; the system administrator and managers. The system administrator is a special user with no access restrictions, including the ACL modification rights, to any modules. The system administrator is similar to the UNIX super user. A manager must be granted the right to modify the ACL of the module by the owner and the owner can revoke the right from the manager any time. In order to implement a concept of managers, we introduce a special class operation called "MODIFY_ACL". The subjects whose IDs are listed in the ACL for MODIFY_ACL operation are managers of the module. Managers can modify ACLs for all operations but MODIFY_ACL. Only the owner of the module and the system administrator can modify the entries of the ACL for MODIFY_ACL operation.
In addition to user-defined class and instance operations, all the class operations inherited from RESOURCE, such as DESTROY.CLASS and INSTANTIATE, have associated ACLs. Note that inheritance is also a right which must be protected and therefore it has an associated ACL.

3.4.2 Access-rights expression

Kieburtz and Silberschatz define two types of consistency for typed objects [34]: internal consistency and external consistency. Internal consistency states that each operator (exported procedure) transfers the resource state into an acceptable state. Internal consistency can be determined from the class definition alone. External consistency specifies a sequential constraint upon the order of operators invoked by any individual user. For example, in order to access a file, a user must open the file, perform any number of read or write operations, then finally close the file. Determining external consistency requires the observation of each program that uses objects.

Kieburtz and Silberschatz developed, in Concurrent Pascal, the concept of access-rights expressions in order to enforce external consistency. An access-rights expression defines the order of invocation of procedures that each user must follow. This order is specified by a use of procedure names and four separators: the semicolon to denote sequencing, the comma to denote alternation, the bracket to denote repetition and parentheses to designate association. For example, the access-rights expression “P; Q; R” specifies that any user is allowed to invoke procedures P, Q and R only in the specified order. Expression “P, Q, R” allows a user to invoke only one of P, Q or R. Expression “[P; R]” specifies that a user is allowed to invoke the sequence P followed by R, zero or more times. Parentheses are used for grouping. Thus, “P; (Q, R); S”
specifies two possible sequence of invocations: “P; Q; S” or “P; R; S”.

Kieburtz and Silberschatz incorporated access-rights expressions in a manager, which is an extension of a monitor (Kieburtz and Silberschatz [33], Silberschatz et al. [46]), and in a class of Concurrent Pascal. For example, suppose manager Reader-Writer defines procedures OpenRead, CloseRead, OpenWrite and CloseWrite, and the Filetype class defines procedures Read and Write. Note that in Concurrent Pascal, an instance of a class allows concurrent invocations of its exported procedures but an instance of a monitor (including manager) allows only a sequential invocation. In this case, the manager Reader-Writer implements the synchronization constraint under which only one user can access Write exclusively or an arbitrary number of users can access Read concurrently on an instance of the Filetype class.

Class Filetype specifies the following access-rights expressions:

\[
\text{access-rights Input} = [\text{Read}] \ || \ \text{Output} = [\text{Write}]
\]

Manager Reader-Writer contains the following declarations:

\[
\text{type Reader-Writer = manager of File_ref: Filetype;}
\]

\[
\text{access-rights Reader} = [\text{OpenRead}; \text{Input}; \text{CloseRead}] \\
\quad \quad || \quad \text{Writer} = [\text{OpenWrite}; \text{Output}; \text{CloseWrite}]
\]

\[
\text{begin}
\quad \text{var Sharedfile: Filetype;}
\quad \quad \\
\text{end.}
\]

Here, an instance RW of the Reader-Writer manager is called a resource manager and an instance of class Filetype is called a managed resource. The access-rights expres-
sions defined by manager RW have embedded subexpressions “Input” and “Output” which are defined by the corresponding managed resource.

In order to access instances of these types, a user must instantiate the Reader-Writer manager and the Filetype class in a program as follows:

```pascal
var RW : Reader-Writer;
Infile : Filetype (Reader) from RW;
```

The `from` clause declares that Infile is a type managed by RW. “(Reader)” is an instantiation parameter to select the access-rights expression “Reader = [OpenRead; Input; CloseRead]” defined in the Reader-Writer manager. It controls access to Infile by substituting “[Read]” for Input to form the access-rights expression “[OpenRead; [Read]; CloseRead]” The user now has static access binding to RW but yet no access binding to Infile. Thus, at this point, the user cannot invoke Read in Infile. In order to acquire a binding to Infile, the user must execute “Infile.OpenRead”. Even though procedure OpenRead is defined in Reader-Writer whose instance is RW, this invocation is valid. A resource manager enhances its managed resource by adding the manager-defined procedures. This is called an enhanced type [33]. Use of a manager allows a programmer to dynamically allocate access binding to a managed resource. Thus, the execution of this statement dynamically binds Infile to the user; thereafter, the user can issue an arbitrary number of “Infile.Read” operation.

The control of external consistency requires the cooperation of a programmer of classes and a user. If a user forgoes the enhanced operations provided by a manager, access may be acquired to a shared instance of a resource (in the above example, access to Infile) by statically allocating Infile [34]. This can happen because monitor-based systems do not totally encapsulate resources. In this case, the user can read
and write from/to Infile without any constraint.

We found access-rights expressions to be very useful in imposing external consistency and have decided to incorporate this concept into our system. However, we believe that the control of external consistency should be in hands of a programmer of a resource module, not a user. Our proposed mechanism is to combine access-rights expressions with the four-tuple ACLs. The \( A[s,o] \) part of an access control list entry \((s, A[s,o])\) lists access-rights expressions that specify not only procedures in module “o” that “s” can invoke but also specifies the orders of invocations to these procedures which “s” must follow. There is no way “s” can circumvent the sequences of invocations specified by a module. Syntactically, instead of a module having a list of \((s, A[s,o])\) entries, we associate each access-rights expression with an access control list. Thus, each ACL is a list of four-tuple entities, followed by an access-rights expression which governs the behavior of each entity in the list.

As an example, suppose that an entity is required to open a file for reading (writing) prior to attempting to read (write) or close the file. The ACLs, in an instance of a file module, may appear as

\[
\{(uid1,*,*,*),(uid2,*,*,*)\}:\text{OpenR};\text{Read};\text{CloseR}
\]

\[
\{(uid1,\text{RM1},\text{inst1},\text{proc1})\}:\text{OpenW};\text{Write};\text{CloseW}.
\]

The first ACL specifies that any procedure of any instance of any class, executing on behalf of users “uid1” or “uid2”, is constrained to the sequence “OpenR” followed by any number of “Reads” followed by “CloseR”. The second ACL specifies that procedure “proc1” of the “inst1” instance of the module class “RM1” executing on behalf of user “uid1” is constrained to the sequence “OpenW”, followed by any number of “Writes” followed by “CloseW”. Note that an associated EOPPE defined in the
synchronization component might have the form

\[
\text{path} = 1 : (\{\text{OpenR; Read}; \text{CloseR}\}, (\text{OpenW; Write}; \text{CloseW})).
\]

The reader can observe that this is very similar to access-rights expressions. However, the open path expression imposes synchronization constraints on concurrent accessing entities (processes) and has nothing to do with the behavior of each individual entity, and if the expression is not satisfied, the corresponding request is temporarily blocked. On the other hand, the access-rights expression specifies the behavior of each accessing entity and if the expression is not satisfied, the corresponding request is rejected. In the degenerate case, an access-rights expression is simply a set of procedure identifiers. For example, the ACL for a procedure X might be

\[
\{(\text{A,stack,stack1,push}),(*,\text{queue,*}),*)\}:X.
\]

Unlike those of Kieburtz and Silberschatz, our access-rights expressions specify the invocation sequence of procedures within one module. If external consistency imposes a particular sequence on invocations of exported procedures defined in more than one module, a programmer can simply create a front-end module which accepts all the requests and invokes the corresponding requests in other modules. In the above file example, assume that procedures OpenR, OpenW, Read, Write, CloseR and CloseW are defined in totally different modules, say, in modules \text{M.OpenR}, \text{M.OpenW}, \text{M.Read}, \text{M.Write}, \text{M.CloseR} and \text{M.CloseW}, respectively (even though this is probably an unrealistic assumption). Then, the programmer may create a front-end module, called File of class File\_CLASS, which declares exported procedures \text{P.OpenR}, \text{P.OpenW}, \text{P.Read}, \text{P.Write}, \text{P.CloseR} and \text{P.CloseW}, respectively and specifies access-rights expressions as follows:
Each exported procedure is defined to call the corresponding procedure in another module. For example, the body of P.OpenR may be declared as follows:

```pascal
procedure P.OpenR();
begin
    M.OpenR.P.OpenR();
end.
```

The access control list of module M.OpenR is defined as follows:

```plaintext
{(*, File.Class, File, P_OpenR)}:OpenR.
```

The access control lists of other modules can be similarly defined. In order to grant or revoke access rights, only the ACLs in module File need to be modified. This is another example of the usefulness of the *-specification in four-tuple entities. Uidl and uid2 can not directly invoke OpenR, OpenW, Read, Write, CloseR or CloseW without going through front-end module File. If uid1 and uid2 invoke the operations in File, they have to follow the sequences specified by access-rights expressions

```plaintext
P.OpenR;|P_Read;P_CloseR or P.OpenW;|P_Write;P.CloseW.
```
4 THE INFORMATION FLOW MECHANISM

4.1 Definition of Flow Control and a Review of Previous Work

An information flow control policy specifies the manner in which classified information may flow from one object to another. Let \( P \) be the set of all flows in an information flow system authorized by a given flow policy, and let \( E \) be the set of all the flows certified by a flow control mechanism. A flow control mechanism is

1. **secure** if \( E \subseteq P \),
2. **insecure** if \( E - P \neq \emptyset \),
3. **precise** if \( E = P \), and
4. **imprecise** if \( E \subset P \).

Two types of flow control models are well known: the access matrix model and the information flow model.

4.1.1 Access matrix model

In the access matrix model, the state of the system is defined by a triple \((S,O,A)\), where \( S \) is a set of subjects, \( O \) is a set of objects, and \( A \) is an access matrix. Bell and LaPadula show how the rules of an access matrix can incorporate a flow control policy.
[7]. Every subject is given a "clearance", and each object is assigned a "classification" that is based on the sensitivity of the information held in the object. Each subject also has a "current security level" which must be equal to or less than the clearance of the subject. In order to maintain the secure system state, the following two properties must hold:

1. simple security property: A subject is not allowed to hold read access to any object unless the classification of the object is less than or equal to the clearance of the subject.

2. *-property:
   - A subject is not allowed to have write access to an object unless the classification of the object is greater than or equal to the current security level of the subject.
   - A subject is not allowed to hold read/write access to an object unless the classification of the object is equal to the current security level of the subject.
   - A subject is not allowed to hold read access to an object unless the classification of the object is less than or equal to the current security level of the subject.

Bell and LaPadula also define a set of operations which effect a transition from one state to another (e.g., get-read-access, give-write-access etc.), and they show that each operation preserves these properties. In general, access control models enforce the rules of an information flow control when subjects acquire access rights to objects.
4.1.2 Information flow model

Information flow models regulate the flow of information between objects independently of the classification of subjects. To formally describe an information flow policy, Denning introduced a lattice model \((SC, \leq, \oplus, \ominus)\) \([15]\), where

1. \(SC\) is a finite set of security classes;

2. \(\leq\) is a binary relation which induces a partially ordering on the security classes in \(SC\);

3. \(\oplus\) is an associative and commutative binary operator on \(SC\), denoting the least upper bound, e.g., \(A \oplus B\) is the least upper bound of classes \(A\) and \(B\);

4. \(\ominus\) is an associative and commutative binary operator on \(SC\), denoting the greatest lower bound, e.g., \(A \ominus B\) is the greatest lower bound of classes \(A\) and \(B\); and

5. \(SC\) has a unique greatest lower bound \(LOW\) and a unique least upper bound \(HIGH\) such that \(LOW \leq A\) and \(A \leq HIGH\) for any \(A\) in \(SC\).

For notational convenience, if \(x\) is a storage object, then the security class of \(x\) will be denoted by \(x\).

Military organizations commonly designate a security class by an ordered pair (classification, compartment), where compartment is a subset of a set of departments. If “\(a\)” and “\(b\)” are classifications and “\(x\)” and “\(y\)” are compartments, then a partial order on the security classes is defined by

\[(a,x) \leq (b,y)\] iff \(a \leq b\) and \(x \subseteq y\).
The lattice defined by this partial order defines the information flow policy. For example, let the classifications consist of TOPSECRET, SECRET, CONFIDENTIAL and UNCLASSIFIED, and the departments consist of PACIFIC and ATLANTIC. Then there are four compartments: (PACIFIC, ATLANTIC), (PACIFIC), (ATLANTIC) and (). The lattice defining the resulting information flow policy is shown in Figure 4.1. For simplicity, the examples in this research assume a linear lattice of security classes consisting of TOPSECRET (= HIGH), SECRET, CONFIDENTIAL and

\[
\{C, (P, A)\} = \text{HIGH} \\
\{C, (A)\} \\
\{U, (P, A)\} \\
\{C, (P)\} \\
\{U, (A)\} \\
\{C, ()\} \\
\{U, (P)\} \\
\{U, ()\} = \text{LOW}
\]

\(U: \text{UNCLASSIFIED}\)  
\(C: \text{CONFIDENTIAL}\)  
\(P: \text{PACIFIC}\)  
\(A: \text{ATLANTIC}\)

Figure 4.1. Lattice Structure Defining an Information Flow Policy
A variable may be either statically or dynamically bound to a security class. A "statically bound variable" is assigned a fixed security class at compile time. A security class of a "dynamically bound variable" changes with the class of its associated information.

An information flow from variable A to variable B is denoted by $A \Rightarrow B$. If B is statically bound, the flow $A \Rightarrow B$ is secure if and only if the relation $A \leq B$ is implied by the lattice. Otherwise, a security violation occurs. If B is dynamically bound, B becomes A and no security violation occurs.

Flows can be classified as explicit or implicit. An explicit flow from variables $a_1, \ldots, a_n$ to variable x occurs when an execution directly assigns information derived from $a_1, \ldots, a_n$ to x. An implicit flow from variables $a_1, \ldots, a_n$ to variable x occurs when the execution of an assignment to x is conditioned upon values derived from $a_1, \ldots, a_n$. Note that implicit flows occur even without the execution of an assignment statement. For example, the statement

$$\text{if } a > 0 \text{ then } x := y \text{ else } y := z$$

causes an explicit flow from "y" to "x" only when $a > 0$, and from "z" to "y" only when $a \leq 0$. The statement also causes implicit flows from "a" to both "x" and "y" regardless of the value of "a". Implicit flows must be considered since the values of "x", "y" and "z" before and after execution of the statement may be used to deduce whether or not "a" is greater than zero.

The execution of a program causes both "direct" and "indirect" flows between variables. An information flow from A to B is called direct if it is the result of a
single assignment or conditional statement. It is called indirect if it is the result of a combination of statements. Consider the following program:

```plaintext
if X = 0 then
    while Y > 0 do
        begin
            C := A + B;
            Y := Y - 1;
        end;
D := C.
```

There are direct flows from A, B, X and Y to C and from X and Y to Y. There is also a direct flow from C to D. Information also flows to D from A, B, X and Y, and these flows are indirect. Formally, we can say that an execution of a program defines a direct flow relation on the variables appearing in the program. The transitive closure of this relation gives all the flows that occur in the execution.

Information flow models can be characterized by

1. their ability to handle statically bound or dynamically bound variables, and
2. whether or not security is verified at compile time or run time.

Fenton developed a run-time certification mechanism for programs with statically bound variables [24,25]. The mechanism is an extension of the Data Mark Machine in which each storage object has a tag field containing the security class of the object. The machine also has a hardware stack containing the class of a program counter in order to account for implicit flows. The top of the stack holds the
least upper bound of the classes of variables on which the statement being executed is conditioned. If the instruction is an assignment to \( x \), the hardware checks that "the class on the top of the stack" \( \leq x \) as well as checking the explicit flow. Fenton proved that for a program with only statically bound variables, implicit flows that occur in absence of execution (e.g., when assignment statements are skipped) can be ignored. However, if a flow violation is detected, then

1. execution of the statement causing the flow violation must be skipped,

2. an error must not be reported to the subject,

3. execution must continue as if no flow violation had occurred, and

4. the program must terminate with a low security state in the program counter stack.

Otherwise, an insecure program could leak one bit of information. For a program containing dynamically bound variables, the handling of implicit flows is even more complicated [24].

Denning developed a compile-time certification procedure for each statement type (e.g., assignment, if statement, while statement etc.) [14,17], assuming that all variables in a program are statically bound to security classes. Denning has also expanded Fenton’s run-time approach to account for implicit flows in programs with only dynamically bound variables [14]. This approach has the additional expense of run-time overhead.

Andrews and Reitman’s approach is based on Hoare’s program verification method [6]. Axioms, based on the lattice model of information flow policy, are developed for
the various types of statements in a programming language, including the synchronization statements — wait and signal.

4.1.3 Restrictions of the access matrix model

We consider access matrix models as described in the previous subsection to be too restrictive compared to information flow models for the following reasons:

1. Access matrix models are applied to large storage objects such as files, while information flow models can be applied to variables in a program [37].

2. Access matrix models involve the clearance of a subject. Information flow models certify flows between objects. For example, suppose subject $S$ reads data from object $O_1$ and writes it to object $O_2$, and the relationship of the classifications is $S < O_1 < O_2$. If a program running on behalf of subject $S$ is memoryless, information flows only from object $O_1$ to object $O_2$ and this is a legal flow according to the lattice model of a flow policy. Access matrix models, however, regard the program to be insecure.

3. Access matrix models do not consider flows of information which are internal to the program. This can lead to over-classification of variables. Hence, some secure programs may be regarded to be insecure (i.e., the mechanism is imprecise). For example, because of the $\ast$-property, a subject $S$ cannot hold read access to an object $O_1$ and write access to an object $O_2$ unless the security level of $O_1$ is less than or equal to the current security level of $S$, and the current security level of $S$ is less than or equal to the security level of $O_2$ ($O_1 \leq S \leq O_2$). However, it may be that information written into $O_2$ is not functionally
dependent on information in O1 and no illegal flow occurs. In order to reduce the possibility of over-classification, the \( * \)-property is redefined by Feiertag et al. as follows \cite{22}:

A subject can modify object O1 which is functionally dependent on object O2 only if \( O1 \leq O2 \).

Information flow models, on the other hand, analyze the internal structure of a program and certify actual flows caused by the execution of program statements. Thus, information flow models may provide more precise certification than access matrix models.

4. Access matrix models must maintain a changing set of conditions and properties; therefore, they are more complicated than information flow models, which simply require that all information flows obey the flow relation specified by a lattice \cite{37}.

For these reasons, we have decided to adopt information flow models for our research.

4.1.4 Problems of applying existing information flow models to the RM system

4.1.4.1 Denning's compile-time mechanism Denning's compile-time mechanism allows only for statically bound variables. In \cite{17}, Denning presented certification semantics for various types of programming statements. For example, the certification semantics of the assignment statement \(<\text{variable}> := <\text{expression}>\), where \(<\text{expression}>\) contains operands \(x_1, \ldots, x_m\), are given by
\[ <\text{expression}> = x_1 \oplus \cdots \oplus x_m \]
if not \((<\text{expression}> \leq <\text{variable}>)\)
then certify the assignment statement
else report security violation.

Thus, if the assignment statement to be certified is "\(X := A + B \times C;\)" , the certification mechanism requires "\(A \oplus B \oplus C \leq X\).

Because of the transitive property of the "\(\leq\)" relation, sequences of secure states are secure. Therefore, certification of a program requires only certification of direct flows caused by individual statements within the program. This approach is attractive from an efficiency point of view since no run-time certification is necessary. One major difficulty of this approach, however, lies in procedure handling. Let \(Q\) be a procedure with formal IN parameters \(x_1, \ldots, x_m\), and formal OUT parameters \(y_1, \ldots, y_n\). Suppose the certification of \(P\) encounters the statement

\[ \text{call } Q(\text{IN } a_1, \ldots, a_m, \text{OUT } b_1, \ldots, b_n). \]

The certification semantics of a procedure call requires verification that

1. \(Q\) is secure,
2. \(a_k \leq x_k\) for \(k=1, \ldots, m\), and
3. \(y_j \leq b_j\) for \(j=1, \ldots, n\).

If \(P\) and \(Q\) are compiled separately, the certification mechanism must defer the verification of the call. The compiler will output all the information necessary to verify the call, and subsequently a linker will use the information to certify the linkage.
flows. In this model, since all parameters must be statically bound, separate versions of functionally equivalent procedures are required for parameters of different security classes. This is not only inconvenient for programmers, but it also severely reduces the possibilities for resource sharing. One possible solution for this problem is to impose the restriction that a procedure cannot access a global variable and cannot contain a variable with a lifetime longer than that of an execution of the procedure itself. Under this restriction, since the output parameters are functions of only the input parameters, the security of a procedure call may be established by verifying that

\[ a_1 \otimes \ldots \otimes a_m \leq b_1 \otimes \ldots \otimes b_n. \]

where \( a_1 \ldots a_m \) are the actual IN parameters and \( b_1 \ldots b_n \) are the actual OUT parameters of the call. The inability to effectively handle state variables is considered to be a major restriction. Also, every actual OUT parameter is not always necessarily a function of all the actual IN parameters (possibly some subset of actual IN parameters). Thus, this verification mechanism potentially over-classifies security classes of actual OUT parameters of a procedure. Over-classification of security classes of variables in a program leads to potential imprecision in a certification mechanism.

### 4.1.4.2 Denning's run-time mechanism

This approach assumes that all variables are dynamically bound and program certification is performed at run time. The certification procedure relies on a hardware support mechanism which includes a tag field in each memory cell and processor register for storing the security class, and a stack HS which contains the security class on which the currently executing statement is conditioned. Stack HS is used to account for implicit flows. The security
class on the top of HS is denoted by \( HS \). The "push(e)" operation places \( HS \oplus e \) at the top of HS and the "pop" operation removes the class on top of HS. The compile-time mechanism transforms conditional statements by inserting push/pop operations into the program as follows:

\[
\text{push}(e); \\
\text{if } e \text{ then } S_1 \text{ [else } S_2]; \Rightarrow \text{if } e \text{ then } S_1 \text{ [else } S_2]; \\
\text{pop};
\]

\[
\text{push}(e); \\
\text{while } e \text{ do } S; \Rightarrow \text{while } e \text{ do begin } S; \text{ pop; push}(e); \text{ end}; \\
\text{pop}.
\]

When an assignment statement

\[
x := F(a_1, \ldots, a_n)
\]

is executed, the hardware automatically updates \( x \) to

\[
a_1 \oplus \ldots \oplus a_n \oplus HS.
\]

This mechanism is not powerful enough to detect illegal implicit flows that occur in the absence of the execution of statements. This is illustrated by the following example [16]:

\[
y := 0; z := 0; \\
\text{if } x = 0 \text{ then } z := 1; \\
\text{if } z = 0 \text{ then } y := 1,
\]
where x is initially bound to SECRET and takes a value of either one or zero, y and z are initially bound to UNCLASSIFIED. After execution of the program, the value of y is equal to the value of x. However, y erroneously remains UNCLASSIFIED since the mechanism described above is unable to detect implicit flows which occur when an execution of an explicit assignment is skipped. This type of information leakage occurs when a dynamically bound variable serves as an intermediary for implicit flow from one conditional statement to another.

In order to account for implicit flows which occur in the absence of the execution of assignment statements, the compile-time mechanism must also insert into the source program "update b" operations, which update b to "b \oplus HS". The statement

\[
\text{if } e \text{ then } S_1 \text{ } | \text{else } S_2
\]

is transformed to

\[
\text{push}(e); \\
\text{if } e \text{ then } S_1 \text{ } | \text{else } S_2; \\
\text{for } x \text{ in } ((V_1 \cup V_2) - (V_1 \cap V_2)), \text{update } x; \\
\text{pop},
\]

where V1 and V2 are sets of variables to which values are assigned in S1 and S2, respectively.

Similarly, the statement

\[
\text{while } e \text{ do } S
\]

is transformed into
push(e);
while e do begin S; pop; push(e) end;
for x in V update x;
pop,

where V is a set of variables to which values are assigned in S.

With this addition to the mechanism, in the above example, z will be increased to SECRET even when the "then" branch is skipped. In turn, y will be increased to SECRET (= z) regardless of the value of z.

Denning also presents a compile-time algorithm which avoids the insertion of unnecessary update operations [14]. This algorithm simulates the execution of a given program in order to calculate the lowest and highest possible security classes that may be assumed at execution time by each variable and HS. If, in the above transformed code for the if statement, a variable "x" in \(( (V_1 \cup V_2) - (V_1 \cap V_2) )\) or V has the lowest possible class which is greater than or equal to the highest possible security class of HS, then \(x \oplus HS\) must equal x and the statement "update x" does not have to be inserted.

The drawback to both of Denning's run-time approaches is that it requires special architecture and incurs significant run-time overhead. In addition, using this approach in our RM system would also require a mechanism to detect flow violations in the access component. This would violate our major design goal of completely separating the functions of protection from the access code.

4.1.4.3 Andrews and Reitman's certification mechanism This method uses a compile-time verification technique. Since the classes of long term variables
(such as state variables) must be determined at compile time, these variables must be statically bound. Other variables may be either statically or dynamically bound. Implicit flows are classified into two types: local flows and global flows. A local flow is an implicit flow within a statement. A global flow is either (a) an implicit flow from the conditional variables of an iteration statement to all subsequent statements, or (b) a flow caused by process synchronization. For example, the sequence of statements

```
x := 0;
while y > 0 do;
x := 1;
```

causes a global flow from y to x since the last statement is conditionally executed, depending on the value of y. The following is an example of a global flow caused by signal/wait operations:

```
process 1:
   if y > 0 then signal(sem);

process 2:
   x := 0;
   wait(sem);
   x := 1.
```

Here, a flow takes place from y to x via the semaphore variable "sem".

In order to handle these two types of flows, Andrews and Reitman introduced special certification variables, local and global. A value in local becomes

\[ \text{local} \oplus \text{exp} \]
within a conditional statement, where \( \text{exp} \) denotes the class of the conditional expression. Upon completion of the conditional statement, the value in \( \text{local} \) reverts to its previous value. \( \text{Global} \), on the other hand, represents an accumulation of the classes of conditions which would be in effect upon completion of the execution of the body of a \text{while} statement or a \text{wait} statement. For example, \( \text{global} \) becomes

\[
\text{exp} \oplus \text{local} \oplus \text{global}
\]

immediately after a \text{while} statement. Note that \( \text{global} \) accumulates not only \( \text{exp} \) but also \( \text{local} \) in order to handle cases in which the \text{while} statement itself is nested within other conditional statements.

Using these certification variables, Andrews and Reitman present proof rules for various types of statements. For example, the rule for \( \text{"x := E;"} \) is

\[
\{P|\ x \leftarrow \ E \oplus \text{local} \oplus \text{global}\} \ x \ := \ E \ \{P\}
\]

where \( P \) is an assertion that is true after executing \( x:=E \), and \( P[x \leftarrow y] \) means \( P \) with every free occurrence of \( x \) replaced by \( y \).

Suppose the assignment statement \( \text{"y := x + 1;"} \) has the precondition

\[
[\text{local} = z, \text{global} = \text{LOW}].
\]

Then, the post condition is

\[
[y = x \oplus z, \text{local} = z, \text{global} = \text{LOW}],
\]

which states that after execution, the class of \( y \) becomes at most as high as \( x \oplus z \) and the classes in \( \text{local} \) and \( \text{global} \) remain unchanged.

To illustrate the proof rule for procedure invocation, assume the procedure head-
procedure PROC(x; var y);
    S /* procedure body */

with the following pre/postconditions for body S,

\{P, y_i \leq \hat{c}_i, \text{local} \leq \hat{l}, \text{global} \leq \hat{g}\}  
S  
\{Q\}.

Here,

1. "x" is a sequence of "call by value" parameters, and "y" is a sequence of "call by value-result" parameters;
2. \(\hat{c}_i\) denotes an initial bound on the value-result parameter \(y_i\); and
3. \(\hat{l}\) and \(\hat{g}\) denote the value of local and global at the point of call.

The invocation "call PROC(IN a, OUT b)" is verified as

\{P[x \leftarrow a, y \leftarrow b], \ b_i \leq \hat{c}_i, \text{local} \leq \hat{l}, \text{global} \leq \hat{g}, \ R\} 
call PROC(a, var b);  
\{Q[x \leftarrow a, y \leftarrow b, \hat{c}_i \leftarrow \hat{c}_i, \hat{l} \leftarrow \hat{l}, \hat{g} \leftarrow \hat{g}, \ R\}

where,

1. \(\hat{c}_i\), \(\hat{l}\) and \(\hat{g}\) are actual bounds on the classes of the parameters and indirect flows at the point of the call, and
2. \(R\) is an assertion for program variables that cannot be changed by \(S\).
The verification of a procedure invocation requires previous verification of the body of the called procedure and previous establishment of the pre/postconditions. Also, the manner in which recursive calls are verified is not clear.

For the RM system, we need a certification mechanism which can verify the "internal" security of a module independently of other called modules, some of which may not yet be certified or for which security information is not yet available.

4.2 Overview of the Information Flow Control Mechanism

4.2.1 Introduction

In the RM model, we extend Denning's and Andrews and Reitman's information flow mechanisms to incorporate the following features:

1. The security classes of program variables can be either dynamically or statically bound. This eliminates the need for more than one version of an exported procedure.

2. Each procedure exported by an RM can be compiled and its "internal" security established independently of other procedures or modules.

3. RMs invoked within a computation may be instantiated or determined dynamically at run time.

4. For efficiency and to satisfy our design goal, run-time information flow security checks will be performed by the protection component only at message passing time.
5. Since a shared RM and its internal state variables have a lifetime which may exceed that of individual programs calling on the procedures exported by the RM, the information flow control mechanism takes into account the security classes of state variables.

6. **OUT** parameters of a procedure in an RM are not restricted to be functions of only **IN** parameters. Each **OUT** parameter is a function of some subset of the **IN** parameters and possibly the state variables of this and other RMs which are subsequently called.

In order to implement these features, we employ both compile-time and run-time mechanisms. The compile-time mechanism establishes the internal security of individual procedures and creates the necessary information structures to allow for efficient run-time certification of inter-module communications. The run-time mechanism completes the certification of the entire program at message passing time by verifying the information flow caused by the procedure invocations.

Before explaining the compile-time and run-time mechanisms in detail, we first identify all possible input and output values to or from a procedure in an RM. We use the terms "input variable" and "output variable" to stand for variables which carry input values to the procedure and output values from the procedure, respectively.

The syntax of the interface specification for an exported procedure of an RM is

\[
\text{procedure PROC (IN } x_1, \ldots, x_i; \text{ OUT } y_1, \ldots, y_m).\]

Then, the possible input variables of a called procedure **PROC** are:

1. **formal IN** parameters of **PROC**;
(2) state variables read by PROC (the values of the state variables of the module which contains PROC when the call is instantiated are considered input values to PROC);

(3) actual OUT parameters from exported procedures which are called by PROC; and

(4) constants and literals local to PROC (assigned the lowest security class of the system (=LOW)).

The possible output variables of PROC are:

(5) formal OUT parameters of PROC;

(6) state variables written by PROC (the values of the state variables when the call terminates are considered output values from PROC); and

(7) actual IN parameters for exported procedures which are called by PROC.

The purpose of the compile-time mechanism is to generate equations which express the run-time information flow symbolically in terms of classes of input variables. The classes of dynamically bound input variables cannot be determined until run time. During compilation, the classes of these input variables are represented by security variables and are denoted by

- procedure-name.parameter-name (for formal IN parameters),

- module-name.procedure-name.parameter-name (for actual OUT parameters returned from external procedures), or
- **state-variable-name** (for state variables).

For example, if the procedure being compiled is specified by \( F(\text{IN} \ a, \ b, \ \text{OUT} \ c) \), then the classes of "a" and "b" are denoted by \( F.a \) and \( F.b \), respectively. If this procedure invokes a procedure \( G \) of a module \( RM1 \) as \( RM1.G(\text{IN} \ x, y, \ \text{OUT} \ z) \), the class of the return value is denoted by \( RM1.G.z \). These security variables are replaced by corresponding actual security classes carried by messages at run time to complete verification of the entire program. **Symbolic class expressions** are generated for output variables (5)-(7) in terms of the classes of the input variables (1)-(4). A symbolic class expression represents the class of a piece of information and consists of a security class and security variables connected by \( \oplus \) operators. For example, the class of information in the expression

\[
A + B ^* C - D / E,
\]

where \( A, B, C, D \) and \( E \) are input variables, is symbolically denoted by

\[
A \oplus B \oplus C \oplus D \oplus E.
\]

Our experience using simple reduction rules indicates that flow expressions tend to be short even for large programs.

Based on these symbolic class expressions, the compile-time algorithm generates two types of symbolic equations: **symbolic class equations** and **symbolic flow equations**. Symbolic class equations are used at run time to calculate the outgoing security classes of output variables. One such equation is created for each parameter in (5) and (7), regardless of whether it is dynamically or statically bound, and for each dynamically bound state variable in (6). A symbolic class equation has the form
variable ≡ "symbolic class expression"

and means that the information in "variable" has a security class given by the "symbolic class expression".

Symbolic flow equations are used to detect flow violations. An equation is created for each statically bound variable. A symbolic flow equation has the form

\[ \text{variable} \equiv \text{security class} \leftarrow "\text{symbolic class expression}" \]

and means that the class of "variable" is statically bound to "security class" and the information whose class is given by "symbolic class expression" flows to "variable" during the execution. Both types of symbolic equations are stored in an information flow template in the object.

The components of an information flow template are:

**EXPORT category**: This stores symbolic class equations for the formal OUT parameters of the procedure.

**IMPORT category**: This consists of symbolic class equations for actual IN parameters of externally invoked procedures. Since there may be more than one externally invoked procedure, this part of the template consists of a list of all such procedure names, each of which is followed by equations for associated actual IN parameters. If the same external procedure is invoked from several different places in the procedure being certified, each invocation is treated as a distinct procedure since each invocation could have a different set of values (and, consequently, different security classes) for the actual IN parameters.\(^1\)

\(^1\)The formation of distinct names could be carried out by a preprocessor prior to compilation.
**STATE category:** This consists of symbolic class equations for dynamically bound state variables. Every dynamically bound state variable is assumed to have an associated tag value that represents its current security class. Class expressions in this category are used to update these tag values upon termination of execution of the procedure.

**STATIC category:** This consists of symbolic flow equations for statically bound variables.

A security class is not associated with a module since information flow is checked at the level of procedure invocation within a module. Therefore, if an identifier of a resource module is passed as a parameter or is stored in a state variable, the compile-time mechanism does not need to generate a symbolic class equation or a symbolic flow equation associated with the module identifier. Flow control is invoked when exported procedures of the module are called or information is returned to a caller.

The information flow templates generated by the compile-time algorithm for the initialization procedure and for all exported procedures are stored in the class module.

At run time, for each invocation of a procedure, an **information flow instance** is created in the protection component of the instance module. The information flow instance is a temporary copy of the information flow template stored in its class module. When message passing takes place, the run-time mechanism uses the information flow instance corresponding to the procedure invocation and completes certification. It does this by replacing the security variables in symbolic class expressions with the actual security classes of the corresponding parameters carried by the message. Upon termination of the procedure invocation, the corresponding information flow instance is destroyed. If a procedure F calls a procedure G in another RM, part of the veri-
fication of F may have to be deferred until G completes. Nevertheless, all run-time verification is confined to the protection components of the involved RMs.

The program shown in Figure 4.2 will be used as the basis for a detailed discussion of these ideas. For simplicity, we will not consider implicit flows between modules in this example. These will be discussed later in this section. More complete examples which includes such implicit flows will be shown in Section 4.5.

In this example, procedure “max” has two IN parameters “a” and “b” and passes values “a” and “d” to procedure “mult” in an external module “R1”. The OUT parameter value returned from “mult” is assigned to “c”. Because of the if statement, the formal OUT parameter “d” receives implicit flows from the state variable “RS” and from “c” as well as an explicit flow from either “RS” or “c”. “a”, “c” and “d” are dynamically bound variables. “b” and “RS” are statically bound to CONFIDENTIAL and SECRET, respectively.

Based on this information, the compiler constructs the following information flow template which consists of equations for each of the output and statically bound variables. The template is stored in class module “R.class”. The following information flow instance of this template becomes part of the protection state (within the protection component) in “R”:

**EXPORT**

\[
\text{max(IN a, b; OUT c)}
\]

\[
c \equiv \text{SECRET } \oplus \text{R1.mult.e}
\]

**IMPORT**

\[
\text{R1.mult(IN a, d, OUT e)}
\]

\[
a \equiv \text{max.a}
\]
Figure 4.2. A Simple Program Example to Illustrate the Information Flow Control Mechanism
\[ d \equiv \text{CONFIDENTIAL} \]

\textbf{STATE}

\textbf{STATIC}

\[ b \equiv \text{CONFIDENTIAL} \leftarrow \text{max}.b \]

\[ \text{RS} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{R1}.mult.e. \]

The symbolic class equation in the \textbf{EXPORT} category reflects the information flow to the caller. The symbolic class equations for \( a \) and \( d \) under procedure name \( \text{R1}.mult \) in the \textbf{IMPORT} category reflect the information flows to “\text{R1}.mult”. Since there is no dynamically bound state variable in this example, the \textbf{STATE} category is empty. The symbolic flow equations in the \textbf{STATIC} category reflect the flows to statically bound variables in an instance of \( \text{R}.\text{class} \). “\text{Max}.a” and “\text{max}.b” denote the security variables corresponding to the formal IN parameters of “\text{max}”. “\text{R1}.mult.e” is the security variable for the actual OUT parameter returned by procedure “\text{R1}.mult”.

Suppose “\text{max}” is called from another module by the statement

\[ \text{R.max(IN x, y, OUT z)}, \]

where \( x = \text{SECRET} \) and \( y = \text{CONFIDENTIAL} \). The run-time information flow mechanism in the protection component creates a new information flow instance of the template, replaces all the occurrences of “\text{max}.a” and “\text{max}.b” by \text{SECRET} and \text{CONFIDENTIAL}, respectively, and verifies whether flow violations occur for statically bound variables. The information flow instance at this point is:

\textbf{EXPORT}

\[ \text{max(IN a, b; OUT c)} \]
\[ \mathbf{c} \equiv \text{SECRET} \oplus \text{R1.mt} \text{.e} \]

**IMPORT**

\[
\text{R1.mt}(\mathbf{N} \ a, \ d, \ \text{OUT} \ e)
\]

\[
\begin{align*}
\mathbf{a} & \equiv \text{max.} \ a = \text{SECRET} \\
\mathbf{d} & \equiv \text{CONFIDENTIAL}
\end{align*}
\]

**STATE**

**STATIC**

\[
\begin{align*}
\mathbf{b} & \equiv \text{CONFIDENTIAL} \leftarrow \text{max.} \ b = \text{CONFIDENTIAL} \\
\text{RS} & \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{R1.mt} \text{.e}.
\end{align*}
\]

The run-time algorithm validates the flows

\[
\begin{align*}
\mathbf{b} & \equiv \text{CONFIDENTIAL} \leftarrow \text{CONFIDENTIAL} \\
\text{RS} & \equiv \text{SECRET} \leftarrow \text{SECRET}.
\end{align*}
\]

Since no flow violation is detected, the message is forwarded to the access component via the synchronization component. A new process corresponding to this message is created in the access component, and the execution is carried out. When "R1.mt" is invoked, a message is constructed and forwarded to the protection component (via the synchronization component). In the protection component, the actual security classes SECRET and CONFIDENTIAL are attached to the parameter values "a" and "d". Suppose "R1.mt" terminates normally and returns an OUT parameter value with security class SECRET. The run-time mechanism replaces R1.mt.e with SECRET, and the information flow instance becomes:

**EXPORT**
\begin{align*}
\text{max(IN } a, b; \text{ OUT } c) \\
&= \text{SECRET } \oplus R1.\text{mult}.e \\
&= \text{SECRET } \oplus \text{SECRET } = \text{SECRET}
\end{align*}

\textbf{IMPORT}

\begin{align*}
R1.\text{mult}(\text{N } a, d; \text{ OUT } e) \\
a &\equiv \text{SECRET} \\
d &\equiv \text{CONFIDENTIAL}
\end{align*}

\textbf{STATE}

\textbf{STATIC}

\begin{align*}
b &\equiv \text{CONFIDENTIAL } \leftarrow \text{CONFIDENTIAL} \\
RS &\equiv \text{SECRET } \leftarrow \text{SECRET } \oplus R1.\text{mult}.e = \text{SECRET } \oplus \text{SECRET} \\
&= \text{SECRET}.
\end{align*}

Since "RS" is statically bound, the run-time mechanism validates the flow

\begin{align*}
RS &\equiv \text{SECRET } \leftarrow \text{SECRET}.
\end{align*}

Since this flow is secure, the message is forwarded to the access component and the execution of "max" resumes. (Note that a return value from "R1.mut" with security class TOPSECRET would cause a flow violation, resulting in abortion of the process and the initiation of some recovery mechanism (e.g., roll back)). Upon completion of the execution of "max", the access component constructs a message containing the return value. When the message reaches the protection component, the run-time mechanism attaches security class SECRET to the message and sends it to the calling module via the underlying inter-module message passing facility. Finally, the corresponding information flow instance is destroyed. As illustrated in the above
example, the flow equations are generated at compile time and actual flow is certified at run time.

4.2.2 Handling implicit flows

Selection and iteration statements warrant special consideration.

4.2.2.1 Information flow semantics of selection For selection statements, the compile-time algorithm accounts for the possibility of executing either alternative. For example, in the statement

if a > 0 then x := b else x := c,

the algorithm constructs the symbolic class expression “x \equiv b \oplus c \oplus a”, accounting for the implicit flow from “a” and the explicit flows from both “b” and “c”.

Since information flows across module boundaries are certified at run time, special treatment of implicit inter-module flows is required. For example, suppose the following statement appears in procedure PROC:

if a > 0 then b := R1.f(x).

Then, there is an implicit inter-module flow from “a” to the local variables of procedure R1.f() and to state variables encapsulated by R1 (and modules called by procedures in R1, etc.).

To handle this flow, the compile-time algorithm constructs a special symbolic class equation in the entry for R1 under the **IMPORT** category of the information flow template for PROC. The class of the outgoing implicit inter-module flow is denoted by *implicit* and represents the least upper bound of the classes of the variables conditioning the procedure invocation. The equation has the form
\[ \text{implicit} \equiv SV_1 \oplus \ldots \oplus SV_n \oplus \text{PROC.implicit}, \]

where \(SV_i\) denotes the \(i^{th}\) variable on which the invocation is locally conditioned and \text{PROC.implicit} denotes the incoming implicit inter-module flow to PROC from the module which called it. Note that an entry in the template for each external procedure has a symbolic class equation for \text{implicit} as well as a symbolic class equation for each of its \text{IN} parameters.

At run time, a message from PROC for a procedure invocation of \(R1.f()\) causes the run-time algorithm to evaluate the symbolic class expression for \text{implicit} of \(R1.f()\). The resulting value (which denotes the class of the outgoing implicit inter-module flow to \(R1\)), as well as the security class of each parameter, is attached to the message. When object \(R1\) receives this message, an information flow instance for \(R1.f()\) is created. The implicit flow class attached to the message replaces the security variable \(f.\text{implicit}\) which appears in class expressions in the information flow instance. Since \(f.\text{implicit}\) appears in each one of the symbolic class equations for \text{implicit} in the \text{IMPORT} category, subsequent procedure invocation requests from this object carry an accumulated class of all the previous implicit inter-module flows to this module.

Implicit flows across module boundaries occur even when procedure invocations are skipped. This is clearly illustrated in the example program (adapted from [16]) shown in Figure 4.3.\(^2\) \(P1, R1\) and \(Q1\) are instances of classes \(P, R\) and \(Q\), respectively. Assume that actual \text{IN} parameter \("x"\) to procedure \(h(\text{IN} x, \text{OUT} y)\) is bound to \text{SECRET} and takes value either one or zero, dynamically bound formal \text{OUT} parameter \("y"\) of \(h(\text{IN} x, \text{OUT} y)\) is initially bound to \text{UNCLASSIFIED}, and dynamically

\(^2\)Only access specifications are described.
Figure 4.3. Implicit Flows Across Module Boundaries
bound state variable Z in R1 and W in Q1 are initially bound to UNCLASSIFIED. Suppose procedure h(IN x, OUT y) in module P1 is executed. First assume "x" has value one. Since the invocation R1.f() is skipped, the value of Z in R1 remains zero and its security class is still UNCLASSIFIED. The invocation R1.g(OUT a) returns the value true with class UNCLASSIFIED. Then, Q1.k() is invoked, and W in Q1 becomes one and remains UNCLASSIFIED. Finally, the invocation Q1.m(OUT y) causes "y" to become one with class UNCLASSIFIED. Now assume the value of "x" is zero. Since R1.f() is invoked, the value of Z in R1 becomes one and has class SECRET. Thus, the invocation R1.g(OUT a) returns the value false with class SECRET in "a". Since Q1.k() is skipped, W in Q1 remains zero and UNCLASSIFIED. Therefore, Q1.m(OUT y) returns zero and UNCLASSIFIED in "y". Note that after execution of h(IN x, OUT y), "y" is equal to "x". However, y erroneously remains UNCLASSIFIED.

In order to take care of implicit flows which occur when invocations are skipped within conditional statements, the compiler must insert code to send run-time "probe" messages to cover branches not containing the calls. If necessary, the compiler creates such branches. A probe carries the class of the expression conditioning the procedure invocation. This class is the same as that of implicit for the procedure in which the invocation occurs. When a module M receives a probe for a skipped invocation M.PROC(), the run-time algorithm must verify the implicit flow to each one of the statically bound variables listed in the STATIC category of the information flow template for M.PROC() and send copies of the probe to potentially subsequently invoked modules listed in the IMPORT category. If all the subsequent modules certify the flow, the run-time algorithm updates the class of each dynamically bound
state variable \( X \) listed in the \texttt{STATE} category of the template to

\[
X \oplus \text{"the class carried by the probe".}
\]

This mechanism corresponds to the "update" instruction introduced by Denning (see Section 4.1.4.2). Procedure \( h(\texttt{IN} \; x, \; \texttt{OUT} \; y) \) in the above example would be translated into the following:

\[
\texttt{procedure} \; h(\texttt{IN} \; x, \; \texttt{OUT} \; y);
\]

\[
\begin{align*}
\texttt{var} \; a : \text{boolean}; \\
\texttt{begin} \\
\quad y := 0; \\
\quad \text{if } x = 0 \text{ then } R1.f() \text{ else } \text{send.probe}(R1.h); \\
\quad R1.g(\texttt{OUT} \; a); \\
\quad \text{if } a \text{ then } Q1.k() \text{ else } \text{send.probe}(Q1.k); \\
\quad Q1.m(\texttt{OUT} \; y);
\end{align*}
\]

In the above example, the symbolic class expression associated with the probe initiated by "send.probe(R1.f)" is "\( \texttt{LOW} \oplus x \oplus h.\text{implicit} \)", and the expression for "send.probe(Q1.k)" is "\( R1.g.a \oplus h.\text{implicit} \)".

It is still possible for an insecure program to leak one bit of information to the user by reporting a security violation. Here, we assume that the clearance of the subject, denoted by \texttt{SUBJECT}, is equal to the security class of the user's output device or the information (file) that will be displayed on the device. The following example shows how implicit flows can be conveyed to a subject by means of a system generated message. Suppose that a program contains the statement
\[ \text{if } a = 0 \text{ then } M.f(\text{OUT } x) \text{ else } \text{send.probe}(M.f). \]

Furthermore, assume that

- "x" is statically bound to SECRET,
- the class of "a" is SECRET,
- the user knows that \( M.f(\text{OUT } x) \) returns TOPSECRET information in "x" and that the inter-module implicit flow to \( M.f(\text{OUT } x) \) does not cause a flow violation, and
- the clearance of the subject executing the program is CONFIDENTIAL (denoted by \( \text{SUBJECT} = \text{CONFIDENTIAL} \)).

If "a" is nonzero, the "then" branch is skipped and no flow violation is detected. If "a" is zero, however, the flow violation is detected because \( M.f(\text{OUT } x) \) returns TOPSECRET information in variable x which is statically bound to SECRET. Thus, the subject can deduce whether "a" is zero or nonzero by observing the appearance or nonappearance of a flow violation error message on the console. Since this is not an information leakage from "a" to "x", the problem is not solved by introducing an else branch to certify the flow from "a" to "x" at run time. This problem occurs because the run-time certification could not check the flow from the return value of "m.f" to "x" until \( M.f(\text{OUT } x) \) is actually invoked.

If the subject does not know the internal details of the program code, the error message may not play an important role. Otherwise, the following two solutions eliminate the problem.
The first solution is proposed by Fenton [24,25]. If a flow violation is detected at run time,

1. the statement which causes the violation is skipped,

2. the flow violation is not reported to the subject, and

3. execution continues as if no flow violation were detected.

Thus, the subject cannot tell whether or not a flow violation has occurred. However, this solution may be impractical. If a program skips some statements and continues execution, it may store incorrect data in state variables. Hence, integrity of information may be lost when a flow violation is detected. Recovery could restore information integrity, but recovery may also cause a leak of information to the subject.

The second solution is to disallow the invocation of an external procedure if the class of conditional expressions on which the invocation is conditioned is not less than or equal to the clearance of the subject. If such an invocation is attempted, the program is aborted. This approach involves not only the classes of objects but also the clearance of the subject. Thus, it does not strictly follow the definition of a pure information flow policy defined by a lattice model. However, since the clearance of the subject is involved only when external procedures are invoked, this approach is less restrictive than the access matrix model.

The second approach can be readily implemented by the run-time algorithm. Upon receipt of a message at run time, after security variables are replaced by the corresponding security classes of parameters carried by a message, the algorithm compares each implicit in the IMPORT category of an information flow template (which denotes the class of conditional expressions on which invocation of the procedure is
conditioned) with the clearance of the subject. If the relation "implicit \leq \text{SUBJECT}" does not hold for every implicit, the algorithm aborts the execution and the error may be reported to the subject. The subject cannot deduce values of variables since execution was not allowed to begin in the called module. Our run-time algorithm, presented in the following sections, assumes this approach.

4.2.2.2 Information flow semantics of iteration Iterative constructs also require special consideration. For example, consider

\begin{verbatim}
a := x;
while a > 0 do
begin
  R1.f1(IN a, OUT b);
  R2.f2(IN b, OUT a);
end.
\end{verbatim}

The first time the body of the loop is executed, the security class of actual IN parameter "a" for R1.f1 is x. However, in subsequent iterations, the class of "a" is the security class of the OUT parameter value from R2.f2(IN b, OUT a) determined in the previous iteration. Since the number of times the loop body will be executed is unknown at compile time, the compile-time mechanism must provide for verification of all possible information flows. This requires simulating iterations of the body of the while statement until the symbolic class expressions stabilize.

In addition, the compiler must insert code immediately following the body of the loop to send probe messages to the external procedures contained in the loop body. This is necessary to account for implicit flows from the conditional expression when
the body of the loop is skipped. The transformed code for the above example is:

```plaintext
a := x;
while a > 0 do
  begin
    R1.f1(IN a, OUT b);
    R2.f2(IN b, OUT a);
  end;
send.probe(R1.f1);
send.probe(R2.f2).
```

Since there are possible flows from "x" and the return value of "R2.f2" to "a", it may appear that the symbolic class equation for "a" should be

```plaintext
a ≡ x ⊕ R2.f2.a.
```

Then, the run-time mechanism would simply replace $R2.f2.a$ with the class of the OUT parameter value when the object receives a return message from the first invocation of $R2.f2(IN b, OUT a)$. But this is incorrect since the security variable $R2.f2.a$ would then disappear from the symbolic class equation for "a" and the equation would not reflect the actual information flows from subsequent invocations of $R2.f2(IN b, OUT a)$. In order to correctly account for flows due to procedure calls in all iterations, the run-time mechanism must give special treatment to security variables corresponding to actual OUT parameters of procedures appearing in loops. In order to identify these security variables, the compile-time mechanism attaches an
accumulation flag (denoted by \((*)\)) to them. Thus, the symbolic class equation for “a” in the above example is

\[ a \equiv x \oplus R2.f2.a(*). \]

When a module receives a return message carrying a security class for a flagged security variable, the run-time algorithm must do the following:

1. If the security variable appears in the symbolic class expression part of a symbolic flow equation, the mechanism adds the class of the return value to the symbolic class expression instead of replacing the security variable.

For example, suppose that an invocation \(R.g(OUT \ x)\) is in a loop and \(R.g.x\) appears in a symbolic flow equation

\[ S \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus R.g.(*) \]

If \(R.g(OUT \ x)\) returns CONFIDENTIAL information in “x” in the first iteration of the loop, the resulting symbolic flow equation is

\[ S \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus R.g.(*) \oplus \text{CONFIDENTIAL} \]

\[ \text{SECRET} \oplus R.g.(*) \]

Suppose that \(R.g(OUT \ x)\) returns TOPSECRET information in “x” in the second iteration. The resulting symbolic flow equation is

\[ S \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus R.g.(*) \oplus \text{TOPSECRET} \]

\[ \text{TOPSECRET} \oplus R.g.(*) \]

and the run-time mechanism detects the flow violation.
2. If the security variable $SV(*)$ appears in the symbolic class expression part of a symbolic class equation, the run-time mechanism replaces it with $SC\ (SV(*))$, where $SC$ is the class of the corresponding return value and $(SV(*))$ is called a "security variable tag". The security variable tag indicates that security class $SC$ is the result of replacing the security variable $SV(*)$.

For example, if invocation R.g(OUT x) in a loop returns TOPSECRET data for x, $R.g.x(*)$ is replaced by "TOPSECRET $(R.g.x(*))$". The class of a return value from R.g(OUT x) in a subsequent iteration replaces "TOPSECRET $(R.g.x(*))$". The value of a symbolic class equation containing security classes with security variable tags is the least upper bound of all the security classes with or without security variable tags.

Suppose that $R.g.x(*)$ appears in the symbolic class equation:

$$y \equiv CONFIDENTIAL \oplus R.g.x(*)$$

Assume $R.g(OUT x)$ returns TOPSECRET in "x" in the first iteration. The run-time mechanism replaces $R.g.x(*)$ with "TOPSECRET $(R.g.x(*))$" and the symbolic class equation becomes

$$y \equiv CONFIDENTIAL \oplus TOPSECRET\ (R.g.x(*))$$

The class of "y" at this point is TOPSECRET. Now assume that $R.g(OUT x)$ returns CONFIDENTIAL in "x" in the second iteration. The run-time algorithm replaces "TOPSECRET $(R.g.x(*))$" in the class equation with "CONFIDENTIAL $(R.g.x(*))$". The resulting symbolic class expression is
The class of "y" at this point is CONFIDENTIAL.

4.2.3 Some remarks on probes and dynamically bound state variables

Consider the following program segment:

if a > 0
  then
    begin
      (instantiate a new module and stores its ID in Q1)
      Q1.p();
    end
  else send.probe(Q1,p),

where Q1 is a variable whose value is determined within the then branch. If a ≠ 0 and the else branch is executed, Q1 is undefined and the run-time mechanism cannot determine to which instance module a probe should be sent.

There are two approaches to this problem:

1. Requiring all state variables to be statically bound.

2. Requiring called modules to be specified by literal names.

In the first approach, variables can be used to specify called modules. If procedure p and (potentially) subsequently called procedures do not access dynamically bound state variables, the run-time algorithm does not have to determine which specific
module is called. It can simply check for a flow violation in the STATIC category of the
information flow template for procedure p (which exists in the class of Q1), since the
same template applies to all instances of the class of Q1. In other words, the run-time
algorithm can send probes to class modules instead of to specific instance modules.
Since state variables are statically bound in the case of most actual implementations,
we do not consider this solution to be too restrictive. Note that with this approach
information flow templates will not have STATE categories.

The second approach, which prohibits the use of variables to specify invoked
modules, allows state variables to be dynamically bound. Since the identity of a called
module is determined at run time (before the corresponding send.probe statement is
executed), the run-time mechanism can update the classes of dynamically bound state
variables by sending a probe to a specific instance module.

If the second approach is used, the following problem must be considered. In an
RM, information flow certification is a function of a protection component. Run-time
flow certification is performed on incoming request messages before the requests reach
the synchronization component. The order in which the protection component receives
request messages may be different from the order of the execution of these requests in
the access component. Therefore, the security verification process in the protection
component might not reflect actual flows of information in the access component.

For example, suppose a request A assigns CONFIDENTIAL data to a dynam-
ically bound state variable ST; a request B assigns the value of ST to a variable X
that is statically bound to CONFIDENTIAL; and the class of ST is currently SE-
CRET and message A arrives before B. Then, the security of each message will be
certified. However, if request A should happen to be blocked in the synchronization
component until B terminates, or if B assigns the value of ST to X before A assigns CONFIDENTIAL data to ST, then a flow violation occurs which was not detected in the protection component.

A possible solution to this problem is to perform information flow certification after the synchronization specification of the module is enforced. If this is done, the order of executing requests will be the same as the order in which the information flow certification on these requests is performed. This indicates that a part of the function of a protection component should be placed after the synchronization component. In this case, the synchronization specification needs to enforce exclusive write access to dynamically bound state variables. We assume this solution throughout this chapter.

4.3 The Compile-Time Algorithm

The purpose of the compile-time algorithm is to generate information flow templates for exported procedures and the initialization procedures of modules.

4.3.1 Reduction rules

The symbolic class expressions for program variables are given initial values as follows:

1. The class expression for a statically bound variable is its fixed security class.
2. The class expression for a dynamically bound local variable or formal OUT parameter is NULL.
3. The class expression for a dynamically bound formal IN parameter is its corresponding security variable.
Each time the algorithm changes the symbolic class expression for a variable, the expression is reduced to a minimal form. A minimal form is either NULL or consists only of a fixed security class and zero or more security variables connected by "⊕" operators. A symbolic class expression is reduced as follows:

Expansion Step:
Each local or output variable is replaced by the symbolic class expression representing the class of the variable.

Minimization Step:

1. All duplicate security variables are deleted. For example,

   "a ⊕ b ⊕ a" becomes "a ⊕ b".

2. If the expression contains NULL along with security classes or security variables, NULL is removed. For example,

   "NULL ⊕ a" becomes "a".

3. If the expression contains more than one security class, all the security classes are replaced by their least upper bound. For example,

   "LOW ⊕ HIGH ⊕ LOW" becomes HIGH, and

   "CONFIDENTIAL ⊕ TOPSECRET" becomes TOPSECRET.

The following illustrate how reductions are performed after changes are made for two successive assignment statements:
4.3.2 Generation of information flow templates

The compile-time certification algorithm will be specified for programs using the following constructs:

1. declaration statement (the declaration of an exported procedure or local variable),
2. assignment statement,
3. compound statement,
4. if statement,
5. while statement, and
6. procedure invocation statement.

The compile-time algorithm requires two special compile-time variables: a stack type variable STACK and a simple variable GLOBAL. STACK contains the security classes of the expressions on which the statement currently being analyzed is
conditioned. Thus, STACK accounts for implicit flows (local flows, in Andrew and Reitman's model). Following Denning's notation, we use

1. **STACK** to denote the class on the top of STACK,

2. "STACK.push(e)" to denote the operation that adds "STACK ⊕ e" to the top of STACK, and

3. "STACK.pop" to denote the operation that removes the class on the top of STACK.

GLOBAL contains a class which reflects

1. implicit flows which will occur after completion of the executions of "while" statements, and

2. the incoming implicit inter-module flow from the caller of a procedure being certified.

If PROC is the procedure being certified, the implicit inter-module flow to PROC is denoted by security variable **PROC.implicit**. We use **GLOBAL** to denote the class contained in **GLOBAL**. **GLOBAL** is initialized to **PROC.implicit**.

The algorithm also uses compile-time array variables SC and EXP. The domain of SC is the set of all the variables which are used in a procedure being certified. The domain of EXP consists of all the statically bound variables used in the procedure. For a dynamically bound variable \(x\), \(SC[x]\) is the symbolic class expression for \(x\). If \(x\) is an output variable, the algorithm uses \(SC[x]\) to construct the symbolic class equation

\(^3\)Andrews and Reitman's global includes these flows.
If \( X \) is a formal \textbf{OUT} parameter of the procedure being compiled, this equation is placed in the \textbf{EXPORT} category of the information flow template. If \( x \) is an actual \textbf{IN} parameter of a procedure to be invoked in another module, the equation is placed in the \textbf{IMPORT} category. If \( x \) is a dynamically bound state variable (assuming dynamically bound state variables are allowed), the equation is placed in the \textbf{STATE} category.

If \( x \) is statically bound, \( SC[x] \) contains the fixed security class to which \( x \) is statically bound and \( EXP[x] \) contains the corresponding symbolic class expression. The algorithm combines these two to construct the symbolic flow equation

\[
X \equiv SC[x] \leftarrow EXP[x],
\]

and places it in the \textbf{STATIC} category of the information flow template.

\textbf{Declaration statements} For a procedure declaration of the form

\begin{verbatim}
procedure PROC(IN x_1 : var.type [of security class C_{x_1}];
   
   \vdots

   x_k : var.type [of security class C_{x_k}];

   OUT y_1 : var.type [of security class C_{y_1}];

   \vdots

   y_m : var.type [of security class C_{y_m}]);
\end{verbatim}

and local variable declarations
var \( a_1 : \text{var}\_\text{type}[\text{of security class } C_{a1}] \);

\[ \vdots \]

\( a_r : \text{var}\_\text{type}[\text{of security class } C_{ar}] \);

the algorithm initializes \( SC \) and \( EXP \) as follows:

1. For a formal IN parameter "\( x_i \)" of \( \text{PROC} \),
   - if \( x_i \) is statically bound to \( C_{xi} \),
     \[ SC[x_i] := C_{xi} \]
     \[ EXP[x_i] := \text{PROC}.x_i \]
   - if \( x_i \) is dynamically bound,
     \[ SC[x_i] := \text{PROC}.x_i \]

   where \( \text{PROC}.x_i \) denotes the security variable for formal IN parameter "\( x_i \)" of "\( \text{PROC} \)."

2. For a local variable or formal OUT parameter \( x_i \) of \( \text{PROC} \),
   - if \( x_i \) is statically bound to \( C_{xi} \),
     \[ SC[x_i] := C_{xi} \]
     \[ EXP[x_i] := \text{NULL} \]
   - if \( x_i \) is dynamically bound,
     \[ SC[x_i] := \text{NULL} \]

The algorithm also creates entries in \( SC \) and \( EXP \) for each state variables as follows:
• for a statically bound state variable \( SV \) bound to \( C_{SV} \),

\[
\begin{align*}
SC[SV] & := C_{SV} \\
\text{EXP}[SV] & := SV, \text{ and}
\end{align*}
\]

• for a dynamically bound state variable \( SV \) (assuming they are allowed),

\[
SC[SV] := SV.
\]

Finally, the algorithm initializes \( \text{STACK} \) to \( \text{LOW} \) and \( \text{GLOBAL} \) to \( \text{PROC.implicit} \).

**Assignment statements**  For an assignment statement of the form

\[
x := f(a_1, \ldots, a_m),
\]

1. if "\( x \)" is a dynamically bound variable, the algorithm makes the assignment

\[
C[x] := SC[a_1] \oplus \ldots \oplus SC[a_m] \oplus \text{STACK} \oplus \text{GLOBAL}.
\]

2. if "\( x \)" is a statically bound variable, the algorithm assigns

\[
\text{EXP}[x] \oplus := SC[a_1] \oplus \ldots \oplus SC[a_m] \oplus \text{STACK} \oplus \text{GLOBAL},
\]

where "\( A \oplus := B \)" stands for "\( A := A \oplus B \)".

The following example shows why updating (rather than replacing) class expressions is necessary for statically bound variables. Suppose \( A \) is a statically bound variable and initially \( SC[A] = \text{CONFIDENTIAL} \) and \( \text{EXP}[A] = \text{NULL} \). Assume statement \( S_i \) assigns the value of variable \( X \) to \( A \) and, later in the text, statement \( S_j \) assigns the value of \( Y \) to \( A \). Using simple replacement, the symbolic class expressions generated for \( S_i \) and \( S_j \) would be
EXP[A] = X ⊕ STACK ⊕ GLOBAL, and
EXP[A] = Y ⊕ STACK ⊕ GLOBAL.

If "STACK ⊕ GLOBAL" is PROC.implicit for both $S_i$ and $S_j$, and there are no other statements that assign values to A after $S_j$, the flow equation

$$A \equiv \text{CONFIDENTIAL} \leftarrow Y \oplus \text{PROC.implicit}$$

would be constructed and placed in the STATIC category of the template. Assume that at run time the classes of X, Y and the incoming implicit inter-module flow are SECRET, CONFIDENTIAL and LOW, respectively. The run-time certification algorithm would replace Y and PROC.implicit in the symbolic flow equation for A with CONFIDENTIAL and LOW and would certify the flow. Even though A holds CONFIDENTIAL information at the end of the execution, the program violates the flow policy by storing SECRET information (class of X) in variable A during the time period between the executions of $S_i$ and $S_j$. Therefore, instead of replacement, the class expressions for statically bound variables must be accumulated using the ⊕ operator in order to account for all possible information flows. The correct symbolic flow equation for A in the above example is

$$A \equiv \text{CONFIDENTIAL} \leftarrow X \oplus Y \oplus \text{LOW} \oplus \text{PROC.implicit}.$$ 

Suppose a record is involved in an assignment statement. Each field of the record may have its own security class and should be treated as a distinct variable. This is also true for arrays, but for simplicity, we assume that every element of an array has the same security class. As pointed out by Denning [17], when an array receives a value, information can flow from the subscripts of the array to the array itself. For
example, suppose array X which is initialized to all zeros receives a non-zero value to its $n^{th}$ element (e.g., $X[n] := 1$). Then, by locating the position of the non-zero element of the array after the assignment, the value of "n" can be determined. For the same reason, if an array reference is in an expression on the right hand side of an assignment, there are flows from both the array and the subscripts to the receiving variable.

**Compound statements** For the statement "begin $S_1; \cdots ; S_m$ end", the compile-time certification algorithm is applied successively to $S_1, \cdots , S_m$.

**If statements** For the statement "if $E$ then $S_1$ [else $S_2$]", the algorithm must first perform STACK.push($E$) in order to account for implicit flows caused by the conditional expression $E$. The variables receiving flows in $S_1$ and/or $S_2$ can be classified into the following three categories:

(i) the variables which are assigned values in both $S_1$ and $S_2$,

(ii) the variables which are assigned values only in $S_1$, and

(iii) the variables which are assigned valued only in $S_2$.

Since the actual execution path is not determined until run time, the algorithm must account for all three possibilities. Assume the temporary arrays of symbolic class expressions $SC_1$ and $SC_2$ are initialized to $SC$ before the if statement is processed. The algorithm is recursively applied to $S_1$ using $SC_1$ and to $S_2$ using $SC_2$. Then, the symbolic class expression for the dynamically bound variables for the if statement is derived as follows:
(1) For a variable "x" in category (i), assign $SC[x] := SC1[x] \oplus SC2[x]$.

(2) For a variable "x" in category (ii), assign $SC[x] \ominus := SC1[x]$.

(3) For a variable "x" in category (iii), assign $SC[x] \ominus := SC2[x]$.

For the statically bound variables, the algorithm accumulates symbolic class expressions in the global array $EXP$ during the processing of $S1$ and $S2$. Upon termination, the algorithm restores $STACK$ to its previous value (by performing "STACK.pop"). Finally, it inserts, in the appropriate branches of the if statement, "send_probe" statements for procedure calls invoked within either $S1$ or $S2$. For example, if procedure $RM.PROC()$ is invoked only within $S1$, the "send_probe(RM.PROC)" statement is inserted in $S2$.

The algorithm for the if statement is

1. Perform $STACK.push(E)$.

2. Create new sets of class expressions $SCI$ and $SC2$, and initialize $SC1[x]$ and $SC2[x]$ to $SC[x]$ for all $x$.

3. Apply the compile-time certification algorithm (recursively) to $S1$ using $SCI$ and $EXP$.

4. Apply the compile-time certification algorithm (recursively) to $S2$ using $SC2$ and $EXP$.

5. Let $V1 = \{x \mid x$ is a dynamically bound variable that receives a flow in $S1\}$, and $V2 = \{x \mid x$ is a dynamically bound variable that $x$ receives a flow in $S2\}$.

Update $SC[x]$ by:
6. Perform STACK.pop.

7. Let \( FN_1 = \{ x \mid x \text{ is a procedure call invoked in } S_1 \} \), and \( FN_2 = \{ x \mid x \text{ is a procedure call invoked in } S_2 \} \).
   
   For \( x \) in \( FN_2 - (FN_1 \cap FN_2) \),
   
   insert "send_probe(x)" at the end of \( S_1 \).
   
   For \( x \) in \( FN_1 - (FN_1 \cap FN_2) \),
   
   insert "send_probe(x)" at the end of \( S_2 \).

**While statements** The information flows that occur due to the statement

\[
\text{while } E \text{ do } S,
\]

may depend upon the number of times \( S \) is actually executed. Consider, for example, the statement

\[
\text{while } a < 0 \text{ do }
\begin{align*}
\text{begin} \\
\text{a := b; } \\
\text{b := c; } \\
\text{c := d; } \\
\text{end.}
\end{align*}
\]
The direct explicit flows that occur during the first iteration of the body of the while statement are "b ⇒ a", "c ⇒ b", and "d ⇒ c". If there is a second iteration, the indirect explicit flows "c ⇒ a" and "d ⇒ b" occur. For three or more iterations, we also have the indirect explicit flows "d ⇒ a" and "d ⇒ b". Since the number of times the body of the loop is executed is not known until run time, the algorithm must simulate the execution and accumulate class expressions across iterations. The class expressions accounting for the explicit flows to "a", "b" and "c" would be "a ⊕ b ⊕ c ⊕ d", "b ⊕ c ⊕ d", and "c ⊕ d", respectively. Note that at run time, if the loop terminates after the first execution of the body, the flows from "c" to "a", "d" to "a", and "d" to "b" do not actually occur. Thus, the mechanism may be imprecise.

After processing the while statement, GLOBAL is updated to the value

\[ \text{GLOBAL} ⊕ \text{STACK} ⊕ E \]

in order to account for the global flows from the conditional expression E. Finally, the algorithm inserts "send.probe" statements immediately after the while statement for procedures invoked inside the loop.

The algorithm consists of the following steps, where

- \( \text{CE} \) denotes the accumulated class of information in the conditional expression E, and

- \( \text{CHANGED} \) is a flag which denotes whether \( \text{SC} \) and \( \text{CE} \) have changed during the simulated execution of S1.

1. Assign \( \text{CE} := \text{LOW} \).

2. Assign \( \text{CHANGED} := \text{false} \).
3. If $CE$ is not equal to "$CE \oplus E$", assign $CE := CE \oplus E$ and $CHANGED := true$.

4. Perform $STACK.push(CE)$ (This accounts for the implicit flows to $S$ caused by $E$).

5. Create new class expressions $SC1$ and $EXP1$, initialized to $SC$ and $EXP$, respectively.

6. Apply the compile-time certification algorithm (recursively) to $S$ using $SC1$ and $EXP$.

7. If $SC$ is not equal to "$SC \oplus SC1$", assign $SC := SC \oplus SC1$ and $CHANGED := true$.

8. Perform $STACK.pop$.

9. If $CHANGED = false$ and $EXP = EXP1$, then go to step 10; otherwise go back to step 2.

10. Assign $GLOBAL \oplus := LOCAL \oplus CE$.

11. For each procedure $x$ invoked in $S$, insert "send.probe($x$)" immediately following the while statement.
The following theorem guarantees that the above algorithm terminates.

**Theorem 4.1**

Steps 2 through 9 of the algorithm are performed no more than \(((|s_1| + |s_2| + 1) + (|v| + p_1) + 1)\) times, where \(s_1\) is the set of dynamically bound variables which receive direct flows in \(S\), \(s_2\) is the set of statically bound variables which receive direct flows in \(S\), \(v\) is the set of all security variables in the procedure being processed, and \(p_1\) is the maximum path length of the security lattice.

**Proof**

Each time steps 3, 6 and 7 are performed, \(SC[x]\) for \(x\) in \(s_1\), \(EXP[x]\) for \(x\) in \(s_2\), or \(CE\) can change. If at least one of the above class expressions changes, the algorithm does not terminate on the current pass. However, suppose a class expression has reached the form:

\[
SV_1 \oplus SV_2 \oplus \ldots \oplus SV_{|v|} \oplus \text{HIGH}
\]

where \(SV_i\) denotes the \(i^{th}\) security variable. Then, the class expression cannot change further. Since each change made to a class expression must add one or more security variables and/or increase the security class, a class expression can be altered only finitely many times.

The worst case occurs when

- all the symbolic class expressions for variables which receive direct flows in \(S\) are NULL before the algorithm processes the **while** statement and eventually become
\[ SV_1 \oplus SV_2 \oplus \ldots \oplus SV_{|v|} \oplus \text{HIGH}, \text{ and} \]

- each pass alters only one of the class expressions either by adding one security variable or by increasing the security class by one level.

Then, \(((s_1| + |s_2| + 1) \times (|v| + pi) + 1)\) passes are necessary before the algorithm terminates. \[ \square \]

Note that if while statements are nested, a "send.probe" statement corresponding to a procedure invocation in an internal while statement must be inserted immediately following each surrounding while statement. Suppose that while statement B surrounds while statement A, and procedure O.g() appears in A. If the body of B is iterated ten times, the number of probes sent for O.g() is eleven (one for each termination of A within B and one for the termination of B itself). The algorithm can easily be modified so that a probe is sent only after the outermost loop terminates. This probe carries the accumulated class of all the nested conditionals. However, the major drawback of this approach is that detection of a possible information flow violation may be delayed until the outermost loop terminates, thereby wasting computing resources.

**Procedure invocation statements** We assume that the following pre-processing has been already taken place:

1. Invocations of the same external procedure from different places in the text have been renamed to make all calls distinct, and

2. An entry in the **IMPORT** category of the template has been created for each externally invoked procedure.
For an invocation of

\[ \text{RM.g}(\text{IN } x_1, \ldots, x_m, \text{OUT } y_1, \ldots, y_n) \]

the corresponding entry in the **IMPORT** category is

\[ \text{RM.g}(\text{IN } x_1, \ldots, x_m, \text{OUT } y_1, \ldots, y_n) \]
\[ \text{implicit } \equiv \text{NULL} \]
\[ x_1 \equiv \text{NULL} \]
\[ \vdots \]
\[ x_m \equiv \text{NULL} \]

When the compile-time algorithm encounters the procedure invocation statement

\[ \text{RM.g}(\text{IN } x_1, \ldots, x_m, \text{OUT } y_1, \ldots, y_n) \]

it accumulates the entries in the **IMPORT** category of the template as follows:

1. implicit \( \oplus := \text{STACK} \oplus \text{GLOBAL} \)
2. \( x_i \oplus := \text{SC}[x_i] \), for each actual IN parameter \( x_i \).

Accumulation, rather than replacement, is required since the procedure invocation may be embedded in a **while** statement.

For each actual OUT parameter \( y_i \), the algorithm does the following:

1. if \( y_i \) is dynamically bound,
   \[ \text{SC}[y_i] := \text{RM.g}.y_i \oplus \text{STACK} \oplus \text{GLOBAL} \]
2. if \( y_i \) is statically bound
   \[ \text{EXP}[y_i] \oplus := \text{RM.g}.y_i \oplus \text{STACK} \oplus \text{GLOBAL} \]

Note that if \( \text{RM.g} \) is invoked within a **while** statement, security variable \( \text{RM.g}.y_i \) must have the (*) mark.
End statement of the procedure  When the compile-time algorithm encounters the end statement of the procedure under certification, it stores symbolic flow/class equations for the statically bound variables and for the dynamically bound state variables (if these are allowed) which receive flows in the body of the procedure in the STATIC and STATE categories, respectively. It also stores symbolic class equations for the formal OUT parameters of the procedure in the EXPORT category of the template.

For the statement "end PROC", the algorithm does the following:

1. For each statically bound variable \( x \), if \( \text{EXP}[x] \) is not NULL, place the symbolic flow equation

\[
    x \equiv \text{SC}[x] \leftarrow \text{EXP}[x]
\]

in the STATIC category of the template, and check whether the flow is legal (by comparing SC\([x]\) with the fixed security class in \( \text{EXP}[x] \)).

If a flow violation is detected, abort the compilation and report the error.

2. For each formal OUT parameter \( x \) of PROC, place the symbolic class equation

\[
    x \equiv \text{SC}[x]
\]

in the EXPORT category of the template.

3. For each dynamically bound state variable \( x \) (assuming dynamically bound state variables are allowed), place the symbolic class equation

\[
    x \equiv \text{SC}[x]
\]

in the STATE category of the template.
4.4 The Run-Time Algorithm

The run-time algorithm is applied whenever the protection component of a module receives a message either from other modules or from the synchronization component.

Messages from other modules are:

1. requests to invoke procedures which are exported by the module,
2. return messages from external procedures invoked by procedures running in the access component, and
3. probe messages.

Messages from the synchronization component are:

1. requests to invoke procedures in other modules,
2. return messages from exported procedures which terminate in the access component,
3. a message reporting termination of the initialization procedure in the access component, and
4. probe messages initiated by procedures running in the access component.

When a resource module RM1 is instantiated, the run-time algorithm:

1. creates an information flow instance for the initialization procedure (i.e., a copy of the information flow template for the initialization procedure stored in the class module) in the protection component,
2. replaces the security variables corresponding to dynamically bound state variables with their current security classes (this step is performed only if dynamically bound state variables are allowed),

3. replaces the security variable init.implicit, which accounts for the class of the implicit inter-module flow from the module that instantiates RM1, with the corresponding security class carried by the instantiation message, and

4. checks for flow violations in each flow equation in the **STATIC** category and certifies the relation

\[
\text{implicit} \leq \text{SUBJECT}
\]

for each implicit in the **IMPORT** category.

If all the equations are certified to be secure, the request is forwarded to the access component via the synchronization component and a new process for the initialization procedure is initiated.

When the initialization procedure terminates, the run-time algorithm updates the classes of dynamically bound state variables based on the symbolic class equations in the **STATE** category (if dynamically bound state variables are allowed), deletes the information flow instance and notifies the protection component so that the module can receive invocation requests for its exported procedures. If the module does not define any exported procedure, the module is destroyed.

---

4 The classes of dynamically bound state variables may be LOW at instantiation time by default.
When RM1 receives a request to invoke an exported procedure PROC, the algorithm

1. creates an information flow instance (which is a copy of the information flow template for PROC) in the protection component,

2. replaces the security variables corresponding to dynamically bound state variables with their current security classes (an information flow instance at this point is referred as an “initial information flow instance”),

3. replaces (PROC.implicit) and the the security variables for formal IN parameters with the corresponding actual security classes carried by the message, and

4. checks for flow violations in each flow equation in the STATIC category and certifies the relation

\[ \text{implicit} \leq \text{SUBJECT} \]

for each implicit in the IMPORT category.

If all the equations are certified to be secure, the request is forwarded to the access component via the synchronization component and a new process for PROC is initiated.

If PROC invokes an external procedure, say “RM1.proc1”, execution of PROC is suspended in the access component and the request for invocation is forwarded to the protection component via the synchronization component. The run-time algorithm finds the entry corresponding to “RM1.proc1” in the IMPORT category of

\[ ^5 \text{This step is performed only if dynamically bound state variables are allowed.} \]
the information flow instance for PROC and determines the classes of the actual IN
parameters and implicit. Then, the algorithm attaches the security classes to the
corresponding actual IN parameters and stores implicit and the class associated with
the clearance of the subject in the message.

In general, the symbolic class equation (in the information flow instance) corre­
sponding to actual IN parameter "x" of "RM1.proc1" has the form

\[ x = S_C_0 \oplus S_C_1(SVT_1) \oplus \ldots \oplus S_C_m(SVT_m) \oplus S_V_1 \oplus \ldots \oplus S_V_n \]

where \( S_C_i \) \((0 \leq i \leq m)\) stands for the \( i^{th} \) security class, \( SVT_i \) \((1 \leq i \leq m)\) stands
for the \( i^{th} \) security variable tag, and \( SV_j \) \((1 \leq j \leq n)\) stands for the \( j^{th} \) security
variable. The security variables are temporarily ignored since they denote the security
classes of variables which, at this point, are not yet flowing into "x". Therefore, the
algorithm only uses

\[ S_C_0 \oplus S_C_1(SV_1) \oplus \ldots \oplus S_C_m(SVT_m) \]

to determine the current security class of "x". The flows to "x" from these input
variables may occur later (in the case of loops), or they have already been skipped and
will never occur in this execution (in the case of if statements or already terminated
loops).

When the module receives a return message from another (previously invoked)
module, the algorithm updates the information flow instance by replacing every occur­
rence of the security variables for the actual OUT parameters with the correspond­
ing security classes carried by the message. If a symbolic class expression contains
a security class with a security variable tag which corresponds to a returned OUT
parameter, say \( SC_1(SV(*)\), the algorithm replaces this tagged security class with
SC2(SV(*)), where SC2 is the returned security class. If a security variable SV corresponding to a returned OUT parameter is marked with (*), the algorithm either

1. adds the returned security class to the expression (if the security variable appears in the class expression of a symbolic flow equation), or

2. replaces SV(*) with SC(SV(*)) where SC is the returned security class (if the security variable appears in the class expression of a symbolic class equation).

The algorithm then checks for flow violations in each symbolic flow equation in the STATIC category and also certifies the relation

\[ \text{implicit} \leq \text{SUBJECT} \]

for each implicit in the IMPORT category. If no flow violation is detected, the message is forwarded to the access component via the synchronization component and the (blocked) calling process is resumed.

When PROC terminates in the access component, the return message is constructed and forwarded to the protection component via the synchronization component. The run-time algorithm finds the security classes corresponding to the formal OUT parameters in the EXPORT category of the information flow instance and attaches these classes to the message. The algorithm also updates the classes of dynamically bound state variables based on the symbolic class equations in the STATE category of the information flow instance (if dynamically bound state variables are allowed). After sending the return message to the calling module, the algorithm deletes the information flow instance.

When PROC performs the "send.probe(RM1.proc1)" operation in the access
component, PROC is blocked and the probe is sent to the protection component (via the synchronization component). The run-time algorithm

1. generates a unique probe identifier (probe.ID),

2. determines implicit from the “RM1.proc1” entry in the IMPORT category of the information flow instance,

3. attaches the site identifier (site.ID), the module identifier (module.ID), probe_ID and implicit to the probe, and

4. sends the probe to RM1, or if dynamically bound state variables are not allowed (in which case RM1 may be a variable), to the class of RM1, say Cl.

When instance module RM1 or class module Cl receives a probe, the run-time algorithm instantiates the information flow template corresponding to the procedure “proc1”, replacing all occurrences of proc1.implicit with implicit. If no flow violation is detected, the algorithm looks at the IMPORT category of the information flow template. If there are no entries for external procedures, the algorithm

1. updates the security class of each dynamically bound state variable to

    implicit @ its current security class (this applies only to RM1),

2. sends “probe.certified” to the sender of the probe, and

3. deletes the information flow instance.

A “probe.certified” message notifies the sender that no flow violation has been detected in this module or in subsequently called modules. If the IMPORT category
do contain entries for external procedures, copies of the probe are sent to all the listed instance modules (for RM1) or to the classes of all the listed modules (for CI). If "probe.certified" messages are received for all probes sent, the module performs the certification steps given above. If a module receives a probe which it has received before, the algorithm immediately sends a "probe.certified" message to the sender and does not need to perform steps 1 and 3. This may happen in a program with recursive invocations.

When the module that initiated the probe receives a corresponding "probe.certified" message, the algorithm notifies the access component (via the synchronization component) to resume the blocked process.

If a flow violation is detected at any point, the algorithm stops execution of the program and invokes a recovery routine.

4.5 Program Examples

In the example programs in this section and in Chapter 5, the specifications for access and synchronization components are not shown. Also, all the exported procedures shown are assumed to be instance operations. We present two examples: one with dynamically bound state variables and one without.

4.5.1 A program with no dynamically bound state variables

The first example program contains only statically bound state variables. The program and its information flow templates are shown in Figure 4.4. Note that the information flow templates do not have STATE categories. The program consists of four modules: JOB1, STACK1, CONSOLE1 and STRANGE1, which are instances of
JOB = class
state
initialize
  var i,x : integer;
  begin
    i := 0;
    while (i < 10) do
      begin
        i := i + 1;
        CONSOLE1.readInt(OUT x);
        STACK1.push(IN x);
      end;
      {send_probe(CONSOLE1.readInt);}
      {send_probe(STACK1.push);}
    i := 0;
    while (i < 10) do
      begin
        i := i + 1;
        STACK1.pop(OUT x);
        CONSOLE1.writeInt(IN x);
      end;
      {send_probe(CONSOLE1.readInt);}
      {send_probe(STACK1.push);}
  end initialize;
end JOB

Figure 4.4. An Example Program with No Dynamically Bound State Variables and Its Information Flow Templates
STACK = class
state
  S : array[1..20] of integer of security class SECRET;
  PTR : integer of security class SECRET;
procedure push(IN a : integer);
begin
  PTR := PTR + 1;
  if a >= 1000 then STRANGE1.convert(OUT a);
  {else send.probe(STRANGE1.convert);}
  S[PTR] := a;
end push;

procedure pop(OUT x : integer);
begin
  pop := S[PTR];
  PTR := PTR - 1;
  if pop <= 0 then STRANGE1.convert(OUT x);
  {else send.probe(STRANGE1.convert);}
end pop;
end STACK

STRANGE = class
state
  A : integer of security class TOPSECRET;
procedure convert(OUT a : integer);
begin
  a := A;
end convert;
end STRANGE

Figure 4.4. (continued)
CONSOLE = class
  state
  KEYBOARD : DEVICE of security class SECRET;
  DISPLAY : DEVICE of security class SECRET;
procedure readint(OUT i : integer);
  begin
    read integer from KEYBOARD device and return the value
    in formal OUT parameter i
  end readint;

procedure writeint(IN x : integer);
  begin
    output the integer value x to the DISPLAY device
  end writeint;
end CONSOLE

The information flow template for JOB.initialize
EXPORT
  initialize
IMPORT
  CONSOLE1.readint(OUT x)
    implicit = LOW @ init.implicit
  STACK1.push(IN x)
    implicit = LOW @ init.implicit
    x = LOW @ CONSOLE1.readint.x(*)
        @ init.implicit
  STACK1.pop(OUT x)
    implicit = LOW @ init.implicit
  CONSOLE1.writeint(IN x)
    implicit = LOW @ init.implicit
    x = LOW @ STACK1.pop.x(*) @ init.implicit
STATIC

Figure 4.4. (continued)
The information flow template for STACK.push

**EXPORT**
- push(IN a)

**IMPORT**
- STRANGE1.convert(OUT a)
  - implicit \(\equiv\) LOW \(\oplus\) push.implicit \(\oplus\) push.a

**STATIC**
- \(S \equiv\) SECRET \(\leftarrow\) SECRET \(\oplus\) push.implicit \(\oplus\) push.a 
  \(\oplus\) STRANGE1.convert.a
- PTR \(\equiv\) SECRET \(\leftarrow\) SECRET \(\oplus\) push.implicit \(\oplus\) push.a

The information flow template for STACK.pop

**EXPORT**
- pop(OUT x)
  - x \(\equiv\) SECRET \(\oplus\) STRANGE1.convert.x \(\oplus\) pop.implicit

**IMPORT**
- STRANGE1.convert(OUT x)
  - implicit \(\equiv\) SECRET \(\oplus\) pop.implicit

**STATIC**
- PTR \(\equiv\) SECRET \(\leftarrow\) SECRET \(\oplus\) pop.implicit

The information flow template for STRANGE.convert

**EXPORT**
- convert(OUT a)
  - a \(\equiv\) TOPSECRET \(\oplus\) convert.implicit

**IMPORT**
**STATIC**

Figure 4.4. (continued)
The information flow template for CONSOLE.readint

**EXPORT**

readint(OUT i)

\[ i \equiv \text{SECRET} \oplus \text{readint.implicit} \]

**IMPORT**

STATIC

The information flow template for CONSOLE.writeint

**EXPORT**

writeint(IN x)

**IMPORT**

STATIC

\[ \text{DISPLAY} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{writeint.x} \]
\[ \oplus \text{writeint.implicit} \]

Figure 4.4. (continued)
class JOB, STACK, CONSOLE and STRANGE, respectively.\(^6\)

Module JOB1 has only an initialization procedure. Within the first while statement, the initialization procedure calls both procedure "readint" of module "CONSOLE1" and procedure "push" of "STACK1" ten times. Within the second while statement, it calls both "pop" of "STACK1" and "writeint" of "CONSOLE1" ten times.

Class "STACK" has two state variables S and PTR, which are statically bound to SECRET. It implements a stack data structure with two exported procedures "push" and "pop". Procedure "push" pushes its IN parameter onto the stack if the value of the parameter is less than 1000. Otherwise, it pushes the value returned by procedure "STRANGE1.convert". Procedure "pop" removes the top element of the stack. It returns the value popped if this value is positive; otherwise it returns the value returned by "STRANGE1.convert".

Class "STRANGE" has an exported procedure "convert" that returns the value in its state variable A, which is statically bound to TOPSECRET. Class "CONSOLE" has two exported procedures, "readint" and "writeint". The state variables KEYBOARD and DISPLAY are assumed to be real devices, and both have classification SECRET. (Thus, only a user with clearance SECRET can use this CONSOLE, and \texttt{SUBJECT = SECRET}). The actual code for these procedures is not shown, but "readint" is assumed to return an integer value read from the device KEYBOARD along with the security class SECRET. Also, the value of the IN parameter "x" of

\(^6\)Even though literal names are used in this example, variables could be used to represent the called modules. Probes are sent to class modules rather than to instance modules whether variables or literal module names are used.
"writeint" is assumed to be output to SECRET device DISPLAY.

When the compile-time algorithm processes the while statements in the initialization procedure of class "JOB", the class expressions of the variables to which values are assigned stabilize after the first simulated execution. Since procedures "CONSOLE1.readint" and "STACK1.pop" are invoked within the while statements, the corresponding security variables "CONSOLE1.readint" and "STACK1.pop" are marked with an (*).

At run time, JOB1 is instantiated, and the information flow instance for the initialization procedure is created, in the protection component of JOB1, based on the information flow template stored in class module JOB. For this example, we assume that init.implicit is LOW. Next, the body of the initialization procedure is executed. Within the first while statement, procedure "CONSOLE1.readint" returns an integer value read from the device KEYBOARD along with the security class SECRET, and the information flow instance of the initialization procedure is updated to

**EXPORT**

initialize

**IMPORT**

CONSOLE1.readint(OUT x)

    implicit ≡ LOW

STACK1.push(IN x)

    implicit ≡ LOW

    x ≡ LOW ⊕ SECRET(CONSOLE1.readint.x(*))

STACK1.pop(OUT x)

    implicit ≡ LOW
CONSOLE1.writeint(IN x)
implicit ≡ LOW
z ≡ LOW ⊕ STACK1.pop.x(*)

Note that since the security variable CONSOLE1.readint is marked with (*), the run-
time mechanism either adds the corresponding security class SECRET to the class
expressions which contain CONSOLE1.readint.x(*) (for symbolic flow equations) or
replaces CONSOLE1.readint with SECRET(CONSOLE1.readint.x(*)) (for symbolic
class equation). Since the STATIC category is empty and

\[ \text{implicit}(= LOW) \leq \text{SUBJECT}(= \text{SECRET}) \]

for all external procedure invocations, the flow is secure and the execution resumes.

When the initialization procedure in JOB1 calls procedure “push” in “STACK1”,
the message carries the integer value just read, along with the security class SECRET,
and the class LOW for implicit. (SECRET and LOW are determined from the asso-
ciated entries in the information flow instance of the procedure).

Upon receipt of the message, the run-time mechanism creates a new information
flow instance in the protection component of STACK1 based on the information flow
template for procedure “push”. Then, the mechanism replaces every occurrence of
push.implicit or push.x with LOW or SECRET, respectively, and checks for a flow
violation in each symbolic flow equation. The mechanism also compares implicit
of STRANGE1.convert(OUT a) in the IMPORT category with SUBJECT. The
information flow instance at this point is

**EXPORT**
Since no flow violation is detected, a new process for executing the "pop" procedure is created in the access component. If the parameter value is greater than or equal to 1000, the procedure invokes "STRANGE1.convert". The message carries the class SECRET to account for the implicit inter-module flow. "STRANGE1.convert" returns an integer value in "a" along with the security class TOPSECRET. The run-time mechanism replaces all occurrences of the security variable "STRANGE1.convert.a", in the information flow instance for "push" in "STACK1", with TOPSECRET, and detects a flow violation as follows:

**EXPORT**

`push(IN a)`

**IMPORT**

`STRANGE1.convert(OUT a)`

`implicit ≡ SECRET`

**STATIC**

`$ ≡ SECRET ← SECRET ⊕ STRANGE1.convert.a`

`PTR ≡ SECRET ← SECRET.`

Since no flow violation is detected, a new process for executing the "pop" procedure is created in the access component. If the parameter value is greater than or equal to 1000, the procedure invokes "STRANGE1.convert". The message carries the class SECRET to account for the implicit inter-module flow. "STRANGE1.convert" returns an integer value in "a" along with the security class TOPSECRET. The run-time mechanism replaces all occurrences of the security variable "STRANGE1.convert.a", in the information flow instance for "push" in "STACK1", with TOPSECRET, and detects a flow violation as follows:

**EXPORT**

`push(IN a)`

**IMPORT**

`STRANGE1.convert(OUT a)`

`implicit ≡ SECRET`

**STATIC**

`$ ≡ SECRET ← TOPSECRET (*** flow violation ***)`

`PTR ≡ SECRET ← SECRET.`
The run-time mechanism then immediately stops the execution and invokes a recovery routine.

If the value of the parameter to "push" is less than 1000, the procedure sends a probe with security class SECRET to class STRANGE. Since the IMPORT and STATIC categories of the information flow template of "STRANGE.convert" are empty, the run-time mechanism immediately returns a "probe.certified" message to STACK1.push, and the execution resumes. Procedure "push" then pushes the value onto the stack, sends the return message to the initialization procedure of JOBl and terminates. The corresponding information flow instance is also deleted from the protection component of the module "STACK1". If all ten values read from the device KEYBOARD are less than 1000, the first while statement terminates normally, and probes containing security class LOW are sent to class modules CONSOLE and STACK. Since the IMPORT and STATIC categories of the information flow template for "CONSOLE.readint" are empty, class module CONSOLE immediately returns a "probe.certified" message. When STACK receives its probe, the run-time algorithm creates an information flow instance and replaces every occurrence of push.implicit with LOW. The resulting information flow instance is

**EXPORT**

push(IN a)

**IMPORT**

`STRANGE1.convert(OUT a)`

implicit ≡ LOW push.a

**STATIC**

S ≡ SECRET ← push.a ⊗ STRANGE1.convert ⊗ SECRET
Since no flow violation is detected, a probe with security class LOW is sent to class module STRANGE, where the probe is also certified to be secure. When STACK receives the "probe.certified" message from STRANGE, the run-time mechanism erases the information flow instance and returns "probe.certified" message to JOB1. Then, the second while statement in the the initialization procedure of JOB1 is executed. "STACK1.pop" and "CONSOLE1.writeint" are called ten times. The information flow instance created in the protection component of "STACK1" when "pop" is called with implicit = LOW is

**EXPORT**

```
pop(OUT x)
  pop ≡ SECRET ⊕ STRANGE1.convert.x
```

**IMPORT**

```
STRANGE1.convert(OUT x)
  implicit ≡ SECRET
```

**STATIC**

```
PTR ≡ SECRET ← SECRET.
```

If a value popped is greater than zero, the return value has the security class SECRET. If the value popped is less than or equal to zero, "pop" calls "STRANGE1.convert", and "STRANGE1.convert" returns an integer value in actual OUT parameter "x" along with the security class TOPSECRET. The information flow instance for the "pop", when STACK1 receives the return value from "STRANGE1", is

**EXPORT**
pop(OUT x)
    pop ≡ TOPSECRET

IMPORT
    Strange1.convert(OUT x)
    inter-module ≡ SECRET

STATIC
    PTR ≡ SECRET ← SECRET,

and the return value has a security class TOPSECRET.

If "STACK1.pop" returns SECRET to JOB1, the information flow instance for
the initialization procedure of JOB1 is updated to

EXPORT
    initialize

IMPORT
    Console1.readint(OUT x)
    implicit ≡ LOW
    Stack1.push(IN x)
    implicit ≡ LOW
    x ≡ SECRET(Console1.readint.x(∗)) ⊕ SECRET
    Stack1.pop(OUT x)
    implicit ≡ LOW
    Console1.writeint(IN x)
    implicit ≡ LOW
    z ≡ LOW ⊕ SECRET(Stack1.pop.x(∗))
STATIC.

If the return value is of security class TOPSECRET, the information flow instance is

**EXPORT**

initialize

**IMPORT**

STACK1.push(IN x)

\[ \text{implicit} \equiv \text{LOW} \]

\[ x \equiv \text{CONSOLE1.readint}(x) \oplus \text{SECRET} \]

CONSOLE1.readint(OUT x)

\[ \text{implicit} \equiv \text{LOW} \]

STACK1.pop(OUT x)

\[ \text{implicit} \equiv \text{LOW} \]

CONSOLE1.writeint(IN x)

\[ \text{implicit} \equiv \text{LOW} \]

\[ z \equiv \text{LOW} \oplus \text{TOPSECRET}(\text{STACK1.pop}(x)) \]

STATIC.

In the former case, the procedure invocation "CONSOLE1.writeint" carries a parameter and its associated security class SECRET. The information flow instance created in the protection component of CONSOLE1 upon the receipt of the message is

**EXPORT**

writeint(IN x)

**IMPORT**
Since the information flow is legal, the execution is carried out in the access component.

In the latter case, the actual parameter to "CONSOLE1.writeint" has security class TOPSECRET, and the information flow instance created for "writeint" is

```
EXPORT
writeint(IN x)
IMPORT
STATIC

DISPLAY ≡ SECRET ← TOPSECRET (* * * flow violation * * *).
```

Therefore, a flow violation is detected and the execution is stopped immediately.

In summary, the program terminates normally only when ten integer values read from the device KEYBOARD are greater than 0 and less than 1000. If a value read is greater than or equal to 1000, an information flow violation is detected in the protection component of module "STACK1". If a value read is less than or equal to 0, a violation is detected in module "CONSOLE1".

4.5.2 A program with dynamically bound state variables

The second example program contains both statically and dynamically bound state variables. The program and the information flow templates generated by the compile-time mechanism are shown in Figure 4.5. Assume that M1, M2, M3 and M4 are instances of class C1, C2, C3 and C4, respectively. "M1.initialize" invokes "M2.f"
C1 = class
class state
  LS1 : integer of security class SECRET;
  LS2 : integer;
initialize
  var a, b, c, d, e : integer;
begin
  a := LS1 * 2;
  b := LS1 / 2;
  c := LS2 + 3;
  if a + b > 0
    then M2.f(IN a, OUT d)
    else
      begin
        d := 0;
        {send.probe(M2.f);}
      end;
  M3.g(IN b, c, OUT e);
  LS1 := d + e;
end
end C1

C2 = class
class state
  procedure f(IN u : integer; OUT z : integer);
  var v, w : integer;
begin
  v := u / 2;
  w := u * 2;
  M4.h(IN v, w, OUT z);
end
end C2

Figure 4.5. An Example Program with Dynamically Bound State Variables and Its Information Flow Templates
C3 = class
    state
    procedure g(IN l, m : integer; OUT n : integer);
        var o, p : integer;
        begin
            o := l + m;
            p := l - 2;
            M4.h(IN o, p, OUT n);
        end
    end C3

C4 = class
    state LS3 : integer;
    procedure h(IN a, b : integer; OUT c);
        begin
            LS3 := LS3 + 2 * a + b;
            c := LS3;
        end
    end C4

The information flow template for C1.initialize

EXPORT
    initialize
IMPORT
    M2.f(IN a, OUT b)
        implicit ≡ SECRET ⊕ init.implicit
        a ≡ SECRET ⊕ init.implicit
    M3.g(IN b, c, OUT e)
        implicit ≡ LOW ⊕ init.implicit
        b ≡ SECRET ⊕ init.implicit
        c ≡ LOW ⊕ LS2 ⊕ init.implicit
STATE
STATIC
    LS1 ≡ SECRET ← SECRET ⊕ M2.f.d ⊕ M3.g.e ⊕ init.implicit

Figure 4.5. (continued)
The information flow template for C2.f

**EXPORT**

\[ M_2.f (\text{IN } u, \text{OUT } z) \]
\[ z \equiv M_4.h.z \oplus f.\text{implicit} \]

**IMPORT**

\[ M_4.h(\text{IN } v, w, \text{OUT } z) \]
\[ \text{implicit} \equiv \text{LOW} \oplus f.\text{implicit} \]
\[ v \equiv \text{LOW} \oplus f.u \oplus f.\text{implicit} \]
\[ w \equiv \text{LOW} \oplus f.u \oplus f.\text{implicit} \]

**STATE**

**STATIC**

The information flow template for C3.g

**EXPORT**

\[ M_3.g(\text{IN } l, m, \text{OUT } n) \]
\[ n \equiv M_4.h.n \oplus g.\text{implicit} \]

**IMPORT**

\[ M_4.h(\text{IN } o, p, \text{OUT } n) \]
\[ \text{implicit} \equiv \text{LOW} \oplus g.\text{implicit} \]
\[ o \equiv g.m \oplus g.n \oplus g.\text{implicit} \]
\[ p \equiv \text{LOW} \oplus g.m \oplus g.\text{implicit} \]

**STATE**

**STATIC**

The information flow template for C4.h

**EXPORT**

\[ M_4.h(\text{IN } a, b, \text{OUT } c) \]
\[ c \equiv \text{LOW} \oplus L_3 \oplus h.a \oplus h.b \oplus h.\text{implicit} \]

**IMPORT**

**STATE**

\[ L_3 \equiv \text{LOW} \oplus L_3 \oplus h.a \oplus h.b \oplus h.\text{implicit} \]

**STATIC**

Figure 4.5.  (continued)
and "M3.g". "M2.f" and "M3.g" invoke "M4.h". M1 has two state variables: LS1 and LS2. LS1 is statically bound to SECRET, and LS2 is dynamically bound. Note that since no information flows to LS2 while "M1.initialize" is executed, there is no corresponding symbolic class equation for LS2 in the STATE category of the template for "M1.initialize". M2 and M3 have no state variables, and M4 has dynamically bound state variable LS3. The run-time algorithm can determine which modules should receive probes since all modules calls must use literal modules names.

Assume that the clearance of the subject is SECRET, state variables LS1 and LS2 are initialized by the instantiation parameters, and the class of LS2 is LOW at instantiation time. When M1 is instantiated, an information flow instance for its initialization procedure is created in the protection component. We assume that $\text{init.implicit} = \text{LOW}$. The resulting information flow instance is

**EXPORT**

initialize

**IMPORT**

M2.f(IN a, OUT b)

$\text{implicit} \equiv \text{SECRET}$

a $\equiv$ SECRET

M3.g(IN b, c, OUT e)

$\text{implicit} \equiv \text{LOW}$

b $\equiv$ SECRET

c $\equiv$ LOW

**STATE**

**STATIC**
\[ \text{LS1} \equiv \text{SECRET} \leftarrow M2.f (a) \oplus M3.g (a) \oplus \text{SECRET}. \]

Since no flow violation is detected, the initialization procedure is invoked. During the execution, assume that "a + b > 0" and "M2.f" is invoked with actual IN parameter "a". The run-time algorithm determines implicit and a from the information flow instance (in this case, both are SECRET) and then attaches these classes to the message.

When M2 receives the message, the algorithm instantiates a new information flow instance and replaces all the occurrences of f.implicit and f.a with SECRET. The information flow instance at this point is

\text{EXPORT} \\
M2.f \ (\text{IN} \ a, \text{OUT} \ z) \\
z \equiv \text{SECRET} \oplus M4.h (z) \\
\text{IMPORT} \\
M4.h (\text{IN} \ v, \ w, \text{OUT} \ z) \\
\text{implicit} \equiv \text{SECRET} \\
v \equiv \text{SECRET} \\
w \equiv \text{SECRET} \\
\text{STATE} \\
\text{STATIC.} \\

Since the flows are certified to be secure, the execution of f(IN a, OUT b) is initiated in the access component. When procedure "f" invokes "M4.h" with actual IN parameters "v" and "w", the algorithm attaches implicit, v and w (all are SECRET) to the message and sends it to M4.
Assume LS3 is CONFIDENTIAL when M4 receives the message. The run-time algorithm creates an information flow instance and replaces LS3 with CONFIDENTIAL, and h.implicit, h.a and h.b with SECRET forming

\[
\text{EXPORT} \\
\text{M4.h(IN a, b, OUT c)} \\
\quad \zeta \equiv \text{SECRET} \\
\text{IMPORT} \\
\text{STATE} \\
\text{LS3 }\equiv \text{SECRET} \\
\text{STATIC}.
\]

Since the flows are secure, the execution is carried out. Note that the synchronization component must guarantee this invocation exclusive access to LS3. Otherwise, the class of LS3 may be changed by other computations after the flow is certified and before the invocation terminates. If this happens, the information flow instance above does not reflect the actual flow caused by this invocation. Upon termination, a message with return value "a" is created. The run-time algorithm attaches the class of "a" (= SECRET) to the message, updates the class of LS3 to SECRET, deletes the information flow instance, and sends the message back to M2.

When M2 receives the return message, the algorithm replaces \text{M4.h.z} with SECRET obtaining

\[
\text{EXPORT} \\
\text{M2.f (IN u, OUT z)} \\
\quad z \equiv \text{SECRET}
\]
IMPORT
M4.h(IN v, w, OUT z)
implicit ≡ SECRET
v ≡ SECRET
w ≡ SECRET

STATE
STATIC.

Since no flow violation is detected, the execution of "M2.f" resumes. Upon termination, "M2.f" returns "z" with its class SECRET to "M1.initialize".

The information flow instance for "M1.initialize" is updated to

EXPORT
initialize
IMPORT
M2.f(IN a, OUT b)
implicit ≡ SECRET
a ≡ SECRET
M3.g(IN b, c, OUT e)
implicit ≡ LOW
b ≡ SECRET
c ≡ LOW

STATE
STATIC
LS1 ≡ SECRET ← SECRET ⊕ M3.g.e.
and no flow violation is detected. When "M1.initialize" invokes "M3.g", the message carries actual IN parameters “b” (with class SECRET) and “c” (with class LOW), and implicit (= LOW). The run-time algorithm forms the following information flow instance in the protection component of M3:

\[
\text{EXPORT}
\]
\[
M3.g(\text{IN } l, m, \text{OUT } n)
\]
\[
\begin{align*}
n & \equiv \text{LOW} \oplus M4.h.n \\
\end{align*}
\]

\[
\text{IMPORT}
\]
\[
M4.h(\text{IN } o, p, \text{OUT } n)
\]
\[
\begin{align*}
\text{implicit} & \equiv \text{LOW} \\
o & \equiv \text{SECRET} \\
p & \equiv \text{SECRET}
\end{align*}
\]

\[
\text{STATE}
\]

\[
\text{STATIC.}
\]

During the execution of "M3.g", "M4.h" is invoked with actual IN parameters “o” and “p”. Since (1) the current class of LS3 is SECRET, (2) implicit is LOW, and (3) the classes of “o” and “p” are SECRET, the resulting information flow instance at M4 is

\[
\text{EXPORT}
\]
\[
M4.h(\text{IN } a, b, \text{OUT } c)
\]
\[
\begin{align*}
c & \equiv \text{SECRET}
\end{align*}
\]

\[
\text{IMPORT}
\]

\[
\text{STATE}
\]

\[
\text{STATIC.}
\]
\text{LS3} \equiv \text{SECRET} \\
\text{STATIC.}

Therefore, "M4.h" returns \text{SECRET} for \text{OUT} parameter "c", and the class of \text{LS3} remains \text{SECRET}.

Upon receipt of the return message from "M4.h", the information flow instance for "M3.g" is updated to

\text{EXPORT}
\text{M3.g(IN \ l, m, OUT \ n)} \\
\text{n} \equiv \text{SECRET}

\text{IMPORT}
\text{M4.h(IN \ o, p, OUT \ n)} \\
\text{implicit} \equiv \text{SECRET} \\
\text{p} \equiv \text{SECRET}

\text{STATE}
\text{STATIC,}

and "M3.g" returns \text{SECRET} for the \text{OUT} parameter "n".

When M1 receives the return message from M3, the algorithm updates the information flow instance to

\text{EXPORT}
\text{initialize}

\text{IMPORT}
M2.f(IN a, OUT b)
  implicit \equiv \text{SECRET}
  a \equiv \text{SECRET}
M3.g(IN b, c, OUT e)
  implicit \equiv \text{SECRET}
  b \equiv \text{SECRET}
  c \equiv \text{SECRET}

\text{STATE}

\text{STATIC}

L51 \equiv \text{SECRET} \leftarrow \text{SECRET}.

Since no flow violation is detected, the execution of "M1.initialize" resumes and terminates normally.
In Chapter 4, we proposed an information flow mechanism for a distributed object-oriented environment. The mechanism is a combination of the compile-time and run-time approaches. The compile-time mechanism generates information flow templates. When messages are passed, the run-time mechanism calculates security classes of output variables and detects information flow violations by using information flow instances generated from the information flow templates.

In order to handle implicit flows between module boundaries, the run-time mechanism sends probes. While probes are in progress, the sending process must be blocked. Thus, sending probes not only increases traffic on the network but also decreases the execution speed of a process. For each exported procedure call within a while statement or an if statement, the mechanism initiates one probe. In addition, the module that receives a probe must send this probe to all the modules listed in the IMPORT category of its corresponding information flow template. Therefore, sending probes may be expensive in some systems, such as those with high communication costs or with programs containing many modules.

In this section, we propose an alternative approach which requires no run-time certification of information flow. The mechanism combines compile-time and link-time approaches. The compile-time mechanism is the same as that proposed in the previous chapter, except that (1) inserting "send_probe" statements is not necessary
and (2) the *-mark for security variables is not necessary. The link-time algorithm is performed after all of the invoked exported procedures in associated modules have been compiled and their information flow templates established, but before execution of the program is initiated. The link-time mechanism collects all the information flow templates of the invoked exported procedures, which exist in the corresponding class modules, at one site (called the base site)\(^1\) and solves, at the base site, every symbolic flow equation in the STATIC categories. If all the symbolic flow equations are certified to be secure, the entire program is secure and the program can be executed. Since the link-time mechanism certifies all the potential information flows caused by a program before the actual execution begins, no run-time checking is necessary. Also, potential information leakage from conditional variables to a subject does not have to be considered. However, since the actual execution sequence or execution path is unknown at link time, the link-time mechanism must consider all possible information flows. Therefore, it is possible that this approach may regard a secure program to be insecure. Also, if a programmer modifies any of the exported procedures invoked within a program (which implies a modification of classes), the link-time algorithm must be performed again before the modified program is executed.

Since dynamically bound state variables may change security classes during their life-times, their classes are not known at link time. Therefore, if a program has dynamically bound state variables, link time verification becomes impossible. The link-time mechanism presented in the following three sections allows only statically

\(^1\)The base site may be any site in the system. We assume that the base site is the one on which the main module of the program is instantiated.

Note that collecting templates does require message passing.
bound state variables. As discussed in Section 4.2.3, however, this allows module invocations to use variables to specify invoked modules.

The class of the implicit inter-module flow from a module that instantiates the main module of a program (denoted by init.implicit) must be the same in every instantiation. We assume that this class is always LOW (i.e., execution of a program is not conditioned by any classified variable). In Section 5.4, we extend the above mechanism to allow dynamically bound state variables and different classes for init.implicit by combining compile-time, link-time and run-time mechanisms. As discussed in the previous chapter, since dynamically bound state variables are allowed, this approach requires that module invocations use only literal module names.

The combined compile-time and link-time approach presented in this chapter is potentially applicable to conventional languages which do not depend on message passing to implement procedure calls.

5.1 The Link-Time Mechanism

Since the mechanism handles only statically bound state variables, information flow templates generated by the compile-time mechanism do not have a STATE category. Moreover, all the security variables appearing in symbolic class expressions in the information flow template of procedure P correspond to dynamically bound formal IN parameters of P, dynamically bound actual OUT parameters of exported procedures invoked by P or incoming implicit inter-module flows to P. Thus, the symbolic class expression corresponding to a security variable may be defined in either the EXPORT or IMPORT categories of the information flow template in which the security variable is defined or in other information flow templates.
A symbolic class expression corresponding to the actual **OUT** parameter of a procedure P which invokes a procedure Q is defined only in the **EXPORT** category of the template for Q. However, since there may be several procedures which invoke P, the symbolic class expressions corresponding to the formal **IN** parameter or the incoming implicit inter-module flow of P may be defined in the **IMPORT** categories of several templates. The algorithm at first combines the symbolic class equations which are in different **IMPORT** categories but represent the same **IN** parameter or the same implicit inter-module flow into one equation by concatenating these with © operators. Thereafter, there is always only one symbolic class equation corresponding to each security variable.

The link-time algorithm is as follows:

For each symbolic flow equation, repeat the following until either a flow violation is detected or all symbolic flow equations are certified to be secure:

1. **Expansion Step:**
   
   repeat the following on the symbolic class expression part of the symbolic flow equation until the expression becomes a single security class.

   (a) Replace every security variable in the symbolic class expression with the right hand side (the symbolic class expression) of the corresponding symbolic class equation.

   (b) Apply the minimization step described in Section 4.3.1 to the expression. If the reduced expression is equivalent to an expression “SC2 ©SV1 ©…© SVn” which has appeared in some previous iteration, replace the expression with “SC2”.
2. Certification Step:

Perform the following flow certification: The equation, at this point, is of form 
"variable ≡ SC1 ← SC2" where SC1 and SC2 are security classes. If SC2 ≤ SC1 
is implied by the lattice, certify this equation to be secure. Otherwise report 
the potential flow violation.

In the algorithm, two reduced expressions are considered equivalent if they con­
tain the same security class and the same set of security variables. If an expression 
appears which is equivalent to some expression occurring in a previous iteration, we 
call this a "repetition". Repetition of an expression in the expansion step may occur if 
a program is recursive. It will be formally shown in the next section that the security 
class found in the expansion step is the value of the symbolic class expression in the 
symbolic flow equation under consideration.

We now present two examples, one with recursive invocations and another with­
out. Note that a formal parameter name does not have to match the corresponding 
actual parameter name. Therefore, the name of a security variable may be different 
from the left hand side of the corresponding symbolic class equation defined in another 
template. In our examples, we use the notation "SV (C.P)" to denote the security 
variable whose name is SV and which appears within the template for procedure P 
developed in class module C. Also, "init" denotes the initialization procedure.

Consider the example program shown in Figure 4.4 in the previous chapter. This 
program does not have recursive invocations. The program causes a flow violation if 
at least one of the values to be read from device KEYBOARD is either greater than 
or equal to 1000 or less than or equal to 0. Otherwise, the program is secure. Since 
the values to be read from KEYBOARD are unknown at link time, the link-time
algorithm detects the worst possible case and regards the program to be insecure. This illustrates the possible imprecision of the algorithm.

The information flow templates of all procedures in the program are shown in Figure 5.1. As mentioned in the previous section, the *-mark is taken out of the symbolic class expressions. There are four symbolic flow equations and the link-time mechanism solves the equations one-by-one until the mechanism encounters an insecure flow or all the equations are verified to be secure.

Since both "STACK.push" and "STACK.pop" call "STRANGE1.convert", the symbolic class equations corresponding to the security variable for the incoming implicit inter-module flow of "STRANGE.convert" are combined as follows:

\[
\text{convert.implicit (STRANGE.convert)} = \text{implicit (STACK.push)} \oplus \text{implicit (STACK.pop)} \\
= \text{LOW} \oplus \text{push.implicit (STACK.push)} \oplus \text{push.a (STACK.push)} \\
\oplus \text{SECRET} \oplus \text{pop.implicit (STACK.pop)} \\
= \text{SECRET} \oplus \text{push.implicit (STACK.push)} \oplus \text{push.a (STACK.push)} \\
\oplus \text{pop.implicit (STACK.pop)}. \\
\]

We then find the symbolic class expression corresponding to each security variable:

\begin{itemize}
  \item \text{CONSOLE1.readint.x (JOB.init)} \\
    = \text{i (CONSOLE.readint)} \\
    = \text{SECRET} \oplus \text{readint.implicit (CONSOLE.readint)}
  \item \text{STACK1.pop.x (JOB.init)} \\
    = \text{x (STACK.pop)} \\
    = \text{SECRET} \oplus \text{STRANGE1.convert.x (STACK.pop)}
\end{itemize}
The information flow template for JOB.initialize

**EXPORT**
initialize

**IMPORT**

- CONSOLE1.readint(OUT x)
  implicit \equiv LOW \oplus init.implicit
  
- STACK1.push(IN x)
  implicit \equiv LOW \oplus init.implicit
  
- STACK1.pop(OUT x)
  implicit \equiv LOW \oplus init.implicit

**STATIC**

The information flow template for STACK.push

**EXPORT**
push(IN a)

**IMPORT**

- STRANGE1.convert(OUT a)
  implicit \equiv LOW \oplus push.implicit \oplus push.a

**STATIC**

\[ S \equiv SECRET \leftarrow SECRET \oplus push.implicit \oplus push.a \]
\[ \oplus STRANGE1.convert.a \]
\[ PTR \equiv SECRET \leftarrow SECRET \oplus push.implicit \oplus push.a \]

Figure 5.1. Information Flow Templates for Program with No Recursion
The information flow template for STACK.pop

**EXPORT**

\[
\text{pop (OUT } x) \\
\quad x \equiv \text{SECRET } \oplus \text{STRANGE1.convert}.x \oplus \text{pop.implicit}
\]

**IMPORT**

\[
\text{STRANGE1.convert} \text{(OUT } x) \\
\quad \text{implicit } \equiv \text{SECRET } \oplus \text{pop.implicit}
\]

**STATIC**

\[
\text{PTR } \equiv \text{SECRET } - \text{SECRET } \oplus \text{pop.implicit}
\]

The information flow template for STRANGE.convert

**EXPORT**

\[
\text{convert} \text{(OUT } a) \\
\quad a \equiv \text{TOPSECRET } \oplus \text{convert.implicit}
\]

**IMPORT**

**STATIC**

The information flow template for CONSOLE.readint

**EXPORT**

\[
\text{readint} \text{(OUT } i) \\
\quad i \equiv \text{SECRET } \oplus \text{readint.implicit}
\]

**IMPORT**

**STATIC**

Figure 5.1. (continued)
The information flow template for CONSOLE.writeint

**EXPORT**

\[ \text{writeint(IN } x) \]

**IMPORT**

**STATIC**

\[ \text{DISPLAY } \equiv \text{SECRET } \leftarrow \text{SECRET } \oplus \text{writeint.x} \]
\[ \oplus \text{writeint.implicit} \]

Figure 5.1 (continued)

- \text{pop.implicit (STACK.pop)}
  - \text{push.a (STACK.push)}
    - \text{=} \text{x (JOB.init)}
    - \text{=} \text{LOW } \oplus \text{CONSOLE1.readint.x (JOB.init)} \oplus \text{init.implicit (JOB.init)}
  - \text{push.implicit (STACK.push)}
    - \text{=} \text{implicit (JOB.init)}
    - \text{=} \text{LOW } \oplus \text{init.implicit (JOB.init)}
  - \text{STRANGE1.convert.a (STACK.push)}
    - \text{=} \text{a (STRANGE.convert)}
    - \text{=} \text{TOPSECRET } \oplus \text{convert.implicit (STRANGE.convert)}
  - \text{STRANGE1.convert.x (STACK.pop)}
    - \text{=} \text{a (STRANGE.convert)}
    - \text{=} \text{TOPSECRET } \oplus \text{convert.implicit (STRANGE.convert)}
  - \text{pop.implicit (STACK.pop)}
As mentioned earlier, we assume $\text{init.implicit} = \text{LOW}$. For the symbolic flow equation for $\text{PTR}$ of "STACK.push", the algorithm expands the class equation as follows:

\[
PTR = \text{SECRET} \leftarrow \text{SECRET} \oplus \text{push.a} (\text{STACK.push}) \\
\quad \oplus \text{push.implicit} (\text{STACK.push}) \\
= \text{SECRET} \oplus \text{LOW} \oplus \text{CONSOLE1.readint.x} (\text{JOB.init}) \\
\quad \oplus \text{init.implicit} (\text{JOB.init}) \\
\quad \oplus \text{LOW} \oplus \text{init.implicit} (\text{JOB.init}) \\
= \text{SECRET} \oplus \text{CONSOLE1.readint.x} (\text{JOB.init}) \\
\quad \oplus \text{init.implicit} (\text{JOB.init}) \\
= \text{SECRET} \oplus \text{SECRET} \oplus \text{readint.implicit} (\text{CONSOLE.readint}) \\
\quad \oplus \text{LOW}
\]
Since SECRET ≤ SECRET, this flow is certified to be secure.

For the symbolic flow equation for **PTR** of “STACK.pop”, we have the expansion sequence:

\[
PTR \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{pop.implicit (STACK.pop)}
\]

\[
= \text{SECRET} \oplus \text{LOW} \oplus \text{init.implicit (JOB.init)}
\]

\[
= \text{SECRET} \oplus \text{init.implicit (JOB.init)}
\]

\[
= \text{SECRET} \oplus \text{LOW}
\]

\[
= \text{SECRET}.
\]

This flow is also verified to be secure.

The symbolic flow equation for **S** of “STACK.push” has the expansion sequence:

\[
S \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{push.a (STACK.push)}
\]

\[
\oplus \text{STRANGE1.convert.a (STACK.push)}
\]

\[
\oplus \text{push.implicit (STACK.push)}
\]

\[
= \text{SECRET} \oplus \text{LOW} \oplus \text{CONSOLE1.readint.x (JOB.init)}
\]

\[
\oplus \text{init.implicit (JOB.init)} \oplus \text{TOPSECRET}
\]

\[
\oplus \text{convert.implicit (STRANGE.convert)} \oplus \text{LOW}
\]

\[
\oplus \text{init.implicit (JOB.init)}
\]
= TOPSECRET ⊕ CONSOLE1.readint.x (JOB.init)
  ⊕ convert.implicit (STRANGE.convert)
  ⊕ init.implicit (JOB.init)

= TOPSECRET ⊕ SECRET
  ⊕ readint.implicit (CONSOLE.readint) ⊕ SECRET
  ⊕ push.implicit (STACK.push) ⊕ push.a (STACK.push)
  ⊕ pop.implicit (STACK.pop) ⊕ LOW

= TOPSECRET ⊕ readint.implicit (CONSOLE.readint)
  ⊕ push.implicit (STACK.push) ⊕ push.a (STACK.push)
  ⊕ pop.implicit (STACK.pop)

= TOPSECRET ⊕ LOW ⊕ init.implicit (JOB.init)
  ⊕ LOW ⊕ init.implicit (JOB.init)
  ⊕ LOW ⊕ CONSOLE1.readint.x (JOB.init)
  ⊕ init.implicit (JOB.init) ⊕ LOW
  ⊕ init.implicit (JOB.init)

= TOPSECRET ⊕ init.implicit (JOB.init)
  ⊕ CONSOLE1.readint.x (JOB.init)

= TOPSECRET ⊕ LOW ⊕ SECRET
  ⊕ readint.implicit (CONSOLE.readint)

= TOPSECRET ⊕ readint.implicit (CONSOLE.readint)

= TOPSECRET ⊕ LOW ⊕ init.implicit (JOB.init)

= TOPSECRET ⊕ init.implicit (JOB.init)

= TOPSECRET ⊕ LOW

= TOPSECRET.
Since TOPSECRET $\leq$ SECRET, this flow is found to be insecure, and therefore the whole program is regarded to be potentially insecure.

Figure 5.2 shows the program and associated information flow templates for the second example. This program involves recursion. Modules O1, O2 and O3 are instances of classes C1, C2 and C3, respectively. "C1.initialize" calls "O2.f". "C2.f" calls "O3.g", and "C3.g" calls "O2.f". Even though literal names are used in this and previous examples to represent the called modules, variables could also be used. This is because the link-time algorithm needs to know only the class of a called module (not the particular instance).

Since both "C1.initialize" and "C3.g" call "O2.f", the symbolic class equations corresponding to the security variables for the formal IN parameter and the incoming implicit inter-module flow of "C2.f" are combined as follows:

• $f.a (C2.f) = a (C1.init) \oplus g (C3.g)$
  
  $= \text{TOPSECRET} \oplus \text{init.implicit} (C1.init) \oplus \text{LOW} \oplus g.x (C3.g)$
  
  $\oplus g.implicit (C3.g)$
  
  $= \text{TOPSECRET} \oplus \text{init.implicit} (C1.init) \oplus g.x (C3.g)$
  
  $\oplus g.implicit (C3.g)$

• $f.implicit (C2.f) = \text{implicit} (C1.init) \oplus \text{implicit} (C3.g)$
  
  $= \text{LOW} \oplus \text{init.implicit} (C1.init) \oplus \text{LOW} \oplus g.x (C3.g)$
  
  $\oplus g.implicit (C3.g)$
  
  $= \text{LOW} \oplus \text{init.implicit} (C1.init) \oplus g.x (C3.g) \oplus g.implicit (C3.g)$.

For the other security variables, we have:
C1 = class
    state RS1 : integer of security class TOPSECRET;
    initialize
        var a, b : integer;
        begin
            a := RS1 / 2;
            O2.f(IN a, OUT b);
            RS1 := b + 20;
        end
    end C1

C2 = class
    state RS2 : integer of security class SECRET;
    procedure f(IN a : integer; OUT b : integer);
    var c : integer;
    begin
        c := a * 2;
        if a > 0
            then
                begin
                    O3.g(IN c, OUT b);
                    RS2 := b;
                end
            else b := RS2;
        end
    end C2

Figure 5.2. An Example Program with Recursion and Its Information Flow Templates
C3 = class
    state RS3 : integer of security class CONFIDENTIAL;
    procedure g(IN x : integer; OUT y : integer);
        var z : integer;
        begin
            z := x + 2;
            if x < 100
                then
                    begin
                        O2.f(IN z, OUT y);
                        RS3 := y;
                    end
            else y := RS3;
        end
    end
end C2

The information flow template for C1.initialize

EXPORT
    initialize

IMPORT
    O2.f(IN a, OUT b)
        implicit \equiv LOW \oplus \text{init.implicit}
    a \equiv \text{TOPSECRET} \oplus \text{init.implicit}

STATIC
    RS1 \equiv \text{TOPSECRET} \rightarrow \text{TOPSECRET} \oplus O2.f.b \oplus \text{init.implicit}

Figure 5.2. (continued)
The information flow template for C2.f

**EXPORT**

\[ f(IN a, OUT b) \]
\[ b \equiv \text{SECRET} \oplus f.a \oplus O3.g.b \oplus f.\text{implicit} \]

**IMPORT**

\[ O3.g(IN c, OUT b) \]
\[ \text{implicit} \equiv \text{LOW} \oplus f.a \oplus f.\text{implicit} \]
\[ c \equiv \text{LOW} \oplus f.a \oplus f.\text{implicit} \]

**STATIC**

\[ RS2 \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus f.a \oplus O3.g.b \oplus f.\text{implicit} \]

The information flow template for C3.g

**EXPORT**

\[ g(IN x, OUT y) \]
\[ y \equiv \text{CONFIDENTIAL} \oplus g.x \oplus O2.f.y \oplus g.\text{implicit} \]

**IMPORT**

\[ O2.f(IN z, OUT y) \]
\[ \text{implicit} \equiv \text{LOW} \oplus g.x \oplus g.\text{implicit} \]
\[ z \equiv \text{LOW} \oplus g.x \oplus g.\text{implicit} \]

**STATIC**

\[ RS3 \equiv \text{CONFIDENTIAL} \leftarrow \text{CONFIDENTIAL} \oplus g.x \oplus O2.f.y \oplus g.\text{implicit} \]

Figure 5.2. (continued)
• \( g \cdot x \ (C3 \cdot g) = \varepsilon \ (C2 \cdot f) = \text{LOW} \oplus f_{a} \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \)

• \( O2 \cdot f \cdot b \ (C1 \cdot \text{init}) = b \ (C2 \cdot f) \)
  
  \[ = \text{SECRET} \oplus f_{a} \ (C2 \cdot f) \oplus O3 \cdot g \cdot b \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \]

• \( O2 \cdot f \cdot y \ (C3 \cdot g) = b \ (C2 \cdot f) \)
  
  \[ = \text{SECRET} \oplus f_{a} \ (C2 \cdot f) \oplus O3 \cdot g \cdot b \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \]

• \( O3 \cdot g \cdot b \ (C2 \cdot f) = y \ (C3 \cdot g) \)
  
  \[ = \text{CONFIDENTIAL} \oplus g \cdot x \ (C3 \cdot g) \oplus O2 \cdot f \cdot y \ (C3 \cdot g) \oplus g_{\text{implicit}} \ (C3 \cdot g) \]

• \( g_{\text{implicit}} \ (C3 \cdot g) = \text{implicit} \ (C2 \cdot f) = \text{LOW} \oplus f_{a} \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \).

Here, init.\text{implicit} is LOW. Each module has a state variable. Therefore, there are three symbolic flow equations. The link-time algorithm solves the equation for RSI as follows:

\[
\text{RSI} \equiv \text{TOPSECRET} \leftarrow \text{TOPSECRET} \oplus O2 \cdot f \cdot b \ (C1 \cdot \text{init})
\]

\[ \oplus \text{init.\text{implicit}} \ (C1 \cdot \text{init}) \]

\[ = \text{TOPSECRET} \oplus \text{SECRET} \oplus f_{a} \ (C2 \cdot f) \oplus O3 \cdot g \cdot b \ (C2 \cdot f) \]

\[ \oplus f_{\text{implicit}} \ (C2 \cdot f) \oplus \text{LOW} \]

\[ = \text{TOPSECRET} \oplus f_{a} \ (C2 \cdot f) \oplus O3 \cdot g \cdot b \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \]

\[ = \text{TOPSECRET} \oplus \text{TOPSECRET} \oplus \text{init.\text{implicit}} \ (C1 \cdot \text{init}) \]

\[ \oplus g \cdot x \ (C3 \cdot g) \oplus g_{\text{implicit}} \ (C3 \cdot g) \oplus \text{CONFIDENTIAL} \]

\[ \oplus g \cdot x \ (C3 \cdot g) \oplus O2 \cdot f \cdot y \ (C3 \cdot g) \oplus g_{\text{implicit}} \ (C3 \cdot g) \oplus \text{LOW} \]

\[ \oplus \text{init.\text{implicit}} \ (C1 \cdot \text{init}) \oplus g \cdot x \ (C3 \cdot g) \oplus g_{\text{implicit}} \ (C3 \cdot g) \]

\[ = \text{TOPSECRET} \oplus \text{init.\text{implicit}} \ (C1 \cdot \text{init}) \oplus g \cdot x \ (C3 \cdot g) \]

\[ \oplus O2 \cdot f \cdot y \ (C3 \cdot g) \oplus g_{\text{implicit}} \ (C3 \cdot g) \]

\[ = \text{TOPSECRET} \oplus \text{LOW} \oplus \text{LOW} \oplus f_{a} \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \]

\[ = \text{TOPSECRET} \oplus \text{LOW} \oplus \text{LOW} \oplus f_{a} \ (C2 \cdot f) \oplus f_{\text{implicit}} \ (C2 \cdot f) \]
\[ \oplus \text{SECRET} \oplus f_a (C2.f) \oplus O3.g.b (C2.f) \oplus f.implicit (C2.f) \]
\[ = \text{TOPSECRET} \oplus f_a (C2.f) \oplus O3.g.b (C2.f) \oplus f.implicit (C2.f) \]

[ repetition ]

\[ = \text{TOPSECRET}. \]

Since TOPSECRET \leq TOPSECRET, this flow is certified to be secure.

The equation for RS2 is solved as follows:

\[ \text{RS2} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus f_a (C2.f) \oplus O3.g.b (C2.f) \]
\[ \oplus f.implicit (C2.f) \]
\[ = \text{SECRET} \oplus \text{TOPSECRET} \oplus \text{init.implicit} (C1.init) \]
\[ \oplus g.x (C3.g) \oplus g.implicit (C3.g) \oplus \text{CONFIDENTIAL} \]
\[ \oplus g.x (C3.g) \oplus O2.f.y (C3.g) \oplus g.implicit (C3.g) \oplus \text{LOW} \]
\[ \oplus \text{init.implicit} (C1.init) \oplus g.x (C3.g) \oplus g.implicit (C3.g) \]
\[ = \text{TOPSECRET} \oplus \text{init.implicit} (C1.init) \oplus g.x (C3.g) \]
\[ \oplus g.implicit (C3.g) \oplus O2.f.y (C3.g) \]
\[ = \text{TOPSECRET} \oplus \text{LOW} \oplus \text{LOW} \oplus f_a (C2.f) \]
\[ \oplus f.implicit (C2.f) \oplus \text{LOW} \oplus f_a (C2.f) \]
\[ \oplus f.implicit (C2.f) \oplus \text{SECRET} \oplus f_a (C2.f) \]
\[ \oplus O3.g.b (C2.f) \oplus f.implicit (C2.f) \]
\[ = \text{TOPSECRET} \oplus f_a (C2.f) \oplus f.implicit (C2.f) \oplus O3.g.b (C2.f) \]
\[ = \text{TOPSECRET} \oplus \text{TOPSECRET} \oplus \text{init.implicit} (C1.init) \]
\[ \oplus g.x (C3.g) \oplus g.implicit (C3.g) \oplus \text{CONFIDENTIAL} \]
\[ \oplus g.x (C3.g) \oplus O2.f.y (C3.g) \oplus g.implicit (C3.g) \]
\[ \oplus \text{LOW} \oplus \text{init.implicit} (C1.init) \oplus g.x (C3.g) \]
Since TOPSECRET $\not\leq$ SECRET, this flow is found to be insecure and the whole program is regarded to be insecure.

### 5.2 Termination and Correctness of the Link-Time Algorithm

In the previous section, we described the link-time algorithm. In this section, we show that the algorithm is correct in the sense that it does not certify an insecure program to be secure.

**Theorem 5.1**

The link-time algorithm eventually terminates.

**Proof**

Since there are only finite number of symbolic flow equations within a given program, if the expansion step of the algorithm terminates on each symbolic flow equation, the entire algorithm eventually terminates.

Since there exist only a finite number of security classes in the system and only a finite number of security variables within a program, only a finite number of non-equivalent symbolic class expressions can be generated. Each application of the expansion step changes one symbolic class expression to another reduced expression. Therefore, a sequence of expansions will eventually produce a repetition unless the expression becomes a single security class, and so the expansion step terminates.
Theorem 5.2

If the expansion step in the link-time algorithm is applied to $SC_2 \oplus SV_1 \oplus \ldots \oplus SV_n$, and terminates with the value $SC_3$, then $SC_2 \oplus SV_1 \oplus \ldots \oplus SV_n = SC_3$.

Proof

Step (a) in the expansion step replaces a security variable with the right hand side of the symbolic class equation for that variable. The reduction in step (b) replaces two or more security classes with their least upper bound. Neither of these replacements changes the value of the symbolic class expression. Therefore, if a sequence of the replacement steps results in a single security class, this security class is the value of the expression.

Note that if steps (a) and (b) are applied to an expression $SC_1 \oplus SV_1 \oplus \ldots \oplus SV_m$ and result in $SC_2 \oplus SV'_1 \oplus \ldots \oplus SV'_k$, then $SC_1 \leq SC_2$. The security class part of a reduced expression never decreases.

Suppose the reduction step (b) produces an expression $SC_3 \oplus SV''_1 \oplus \ldots \oplus SV''_m$ which is equivalent to an expression which has appeared in some previous iteration.

Assume the expansion sequence:

\[ X = \overline{SC_1} \leftarrow \overline{SC_2} \oplus \overline{SV_1} \oplus \ldots \oplus \overline{SV_n} \quad (1) \]

\[ \vdots \]

\[ = \overline{SC_3} \oplus \overline{SV'_1} \oplus \ldots \oplus \overline{SV'_k} \quad (2) \]

\[ \vdots \]

\[ = \overline{SC_3} \oplus \overline{SV''_1} \oplus \ldots \oplus \overline{SV''_m} \quad (3) \]

where (3) is equivalent to (2).
Suppose that $SV_1^I \oplus \ldots \oplus SV_m^I$ in (2) is expanded to $SC_4 \oplus SV_1^II \oplus \ldots \oplus SV_l^II$ before (3) is reached, where $SC_4$ is a security class which is not less than or equal to $SC_3$. Because $SC_3 \oplus SC_4 \not\leq SC_3$, the resulting expression can never become (3). Thus,

$$SV_1^I \oplus \ldots \oplus SV_m^I = SV_1^II \oplus \ldots \oplus SV_m^II \leq SC_3$$

and $SC_3 \oplus SV_1^II \oplus \ldots \oplus SV_m^II = SC_3$. □

**Theorem 5.3**

Suppose all state variables in a program are statically bound. If the link-time algorithm certifies the program to be secure, the program is secure.

**Proof**

The compile-time algorithm creates a template for each procedure in the system. A template reflects all information flows, from input parameters and statically bound variables to output parameters and statically bound variables, that potentially occur within a procedure. We will call such information flows "direct flows". The direct flows define a binary relation on the set of input and output parameters and statically bound variables.

In order to combine the direct flows of different templates, we identify the output parameter of a procedure call with the input parameter of the called procedure. Now, within a program, an information flow from one statically bound variable to another is either a direct flow or is the result of a sequence of direct flows which defines a path (with respect to the direct flow relation) through the set of input parameters and
statically bound variables. All such flows are captured by the transitive closure of the relation defined by direct flows. We call this transitive closure the "flow relation".

For each statically bound variable, X, the expansion step in the link-time algorithm finds the least upper bound of the security classes of the set of statically bound variables that are related to X by the flow relation. Therefore, the link-time algorithm finds all possible flows from one statically bound variable to another. Hence, if the algorithm certifies the program, no flow violation can occur within the program. □

5.3 The Modified Link-Time Mechanism

In this section, we consider the imprecision of the link-time mechanism defined in the previous sections and present a modified link-time mechanism which avoids some of this over-classification.

Over-classification may occur because the algorithm at first combines all the symbolic class equations in IMPORT categories representing either the same IN parameters or the same outgoing implicit flows. Consider the example shown in Figure 5.3. Assume O0, O1, O2 and O3 are instances of classes C0, C1, C2 and C3, respectively. The program does not cause a flow violation, since when O1 and O2 call "O3.f", "O3.f" returns SECRET information to local variable "y" of procedure "h", and TOPSECRET data to variable "b" of procedure "g". However, if the proposed link-time algorithm is applied, this program is regarded to be insecure as follows.

The symbolic class equations

\[ x (C1.h) \equiv \text{SECRET} \oplus \text{h.implicit} (C1.h) \]

in the information flow template of procedure "f" and
C0 = class
  Initialize
  begin
    01.h();
    02.g();
  end
end

C1 = class
  state SV1: integer of security class SECRET;
  procedure h();
    var x, y: integer;
    begin
      x := SV1 * 2;
      O3.f(IN x, OUT y);
      SV1 := y + 1;
    end
end

C2 = class
  state SV2: integer of security class TOPSECRET;
  procedure g();
    var a, b: integer;
    begin
      a := SV2 / 3;
      O3.f(IN a, OUT b);
      SV2 := b + 3;
    end
end

Figure 5.3. Over-Classification Caused by the Link-Time Algorithm
C3 = class
  procedure f(IN m : integer; OUT n : integer);
    begin
      n := m;
    end
end

The information flow template for C0.initialize

**EXPORT**
  initialize

**IMPORT**
  O1.h()
    implicit \equiv LOW \oplus init.implicit
  O2.g()
    implicit \equiv LOW \oplus init.implicit

**STATIC**

The information flow template for C1.h

**EXPORT**
  h()

**IMPORT**
  O3.f(IN x, OUT y)
    implicit \equiv LOW \oplus h.implicit
    x \equiv SECRET \oplus h.implicit

**STATIC**
  SV1 \equiv SECRET \leftarrow SECRET \oplus O3.f.y \oplus h.implicit

Figure 5.3. (continued)
The information flow template for C2.g

EXPORT
\hspace{1cm}g()

IMPORT
\hspace{1cm}O3.f(IN a, OUT b)
\hspace*{.7cm}\text{implicit} \equiv \text{LOW} \oplus g.\text{implicit}
\hspace*{1cm}a \equiv \text{TOPSECRET} \oplus g.\text{implicit}

STATIC
\hspace{1cm}SV2 \equiv \text{TOPSECRET} \leftarrow \text{TOPSECRET} \oplus O3.f.b \oplus g.\text{implicit}

The information flow template for C3.f

EXPORT
\hspace{1cm}f(IN m; OUT n)
\hspace*{.7cm}n \equiv f.m \oplus f.\text{implicit}

IMPORT

STATIC

Figure 5.3. (continued)
\( a \ (C2.g) \equiv \text{TOPSECRET} \oplus g.\text{implicit} \ (C2.g) \)

in the information flow template of procedure "g" represent the same IN parameter "m" of procedure "f". Thus, the algorithm combines these equations into

- \( f.m \ (C3.f) = x \ (C1.h) \oplus a \ (C2.g) \)

  \[= \text{SECRET} \oplus h.\text{implicit} \ (C1.h) \oplus \text{TOPSECRET} \oplus g.\text{implicit} \ (C2.g)\]

  \[= \text{TOPSECRET} \oplus h.\text{implicit} \ (C1.h) \oplus h.\text{implicit} \ (C2.g).\]

Also, \( \text{implicit} \ (C1.h) \) and \( \text{implicit} \ (C2.g) \) are combined to form the expression for \( f.\text{implicit} \ (C3.f) \) as follows:

- \( f.\text{implicit} \ (C3.f) = \text{implicit} \ (C1.h) \oplus \text{implicit} \ (C2.g) \)

  \[= \text{LOW} \oplus h.\text{implicit} \ (C1.h) \oplus \text{LOW} \oplus g.\text{implicit} \ (C2.g)\]

  \[= \text{LOW} \oplus h.\text{implicit} \ (C1.h) \oplus g.\text{implicit} \ (C2.g).\]

The symbolic class equations corresponding to other security variables are

- \( h.\text{implicit} \ (C1.h) = \text{implicit} \ (C0.\text{init}) = \text{LOW} \oplus \text{init.implicit} \ (C0.\text{init}) \)
- \( g.\text{implicit} \ (C2.g) = \text{implicit} \ (C0.\text{init}) = \text{LOW} \oplus \text{init.implicit} \ (C0.\text{init}) \)
- \( O3.f.y \ (C1.h) = n \ (C3.f) = f.m \ (C3.f) \oplus f.\text{implicit} \ (C3.f) \)
- \( O3.f.b \ (C3.f) = n \ (C3.f) = f.m \ (C3.f) \oplus f.\text{implicit} \ (C3.f) \).

Note that \( \text{init.implicit} \ (C0.\text{init}) \equiv \text{LOW} \). Symbolic flow equation

\[ SV2 \equiv \text{TOPSECRET} \rightarrow \text{TOPSECRET} \oplus O3.f.b \oplus g.\text{implicit} \]

in the information flow template of "C2.g" is transformed as follows:
SV2 = TOPSECRET ← TOPSECRET ⊕ O3.f.b (C2.g) ⊕ g.implicit (C2.g)
= TOPSECRET ⊕ f.m (C3.f) ⊕ f.implicit (C3.f) ⊕ LOW
⊕ init.implicit (C0.init)
= TOPSECRET ⊕ f.m (C3.f) ⊕ f.implicit (C3.f)
⊕ init.implicit (C0.init)
= TOPSECRET ⊕ TOPSECRET ⊕ h.implicit (C1.h)
⊕ g.implicit (C2.g) ⊕ LOW ⊕ h.implicit (C1.h)
⊕ g.implicit (C2.g) ⊕ LOW
= TOPSECRET ⊕ h.implicit (C1.h) ⊕ g.implicit (C2.g)
= TOPSECRET ⊕ LOW ⊕ init.implicit (C0.init) ⊕ LOW
⊕ init.implicit (C0.init)
= TOPSECRET ⊕ init.implicit (C0.init)
= TOPSECRET ⊕ LOW
= TOPSECRET.

The flow to SV2 is therefore regarded to be secure. However, symbolic flow equation

SV1 ≡ SECRET ← SECRET ⊕ O3.f.y ⊕ h.implicit

of "h" is regarded to be insecure because of the following transformation:

SV1 ≡ SECRET ← SECRET ⊕ O3.f.y ⊕ h.implicit

= SECRET ⊕ f.m (C3.f) ⊕ f.implicit (C3.f) ⊕ LOW
⊕ init.implicit (C0.init)
= SECRET ⊕ f.m (C3.f) ⊕ f.implicit (C3.f) ⊕ init.implicit (C0.init)
= SECRET ⊕ TOPSECRET ⊕ h.implicit (C1.h) ⊕ g.implicit (C2.g)
We now present a modification of the link-time algorithm which avoids this type of over-classification. The example above shows that over-classification may result from combining class equations representing either the same actual IN parameters or the same outgoing implicit inter-module flows. The algorithm needs to distinguish between situations in which combining class equations is legitimate and those in which it is not. The following discussion is based on the examples in Figure 5.4 and Figure 5.5. We assume the algorithm is solving a symbolic flow equation of procedure A. Also, we will not consider security variables for incoming implicit inter-module flows and symbolic class equations for outgoing implicit inter-module flows. Incoming and outgoing implicit inter-module flows can be treated in the same manner as the flows for formal and actual IN parameters.

Figure 5.4 represents the case in which B calls C with actual IN parameter “a” and actual OUT parameter “b”, then B calls A with actual IN parameter “c” and actual OUT parameter “d”. Formal IN parameters of A, B and C are “x”, “y” and “z”, and formal OUT parameters of A, B and C are “u”, “v” and “w”, respectively. Beside these invocations, A is called from other procedures with actual IN parameters “i” and “j”; B is called with actual IN parameters “k”, “l” and “m”; and C is called
procedure A

procedure B

procedure C

Figure 5.4. Information Flow Paths Due to Procedure Invocations (Case 1)
Figure 5.5. Information Flow Paths Due to Procedure Invocations (Case 2)
with actual IN parameters “n” and “o”.

Assume the symbolic flow equation of A on which the mechanism is working contains security variable xi. Since A may be called from many procedures and the algorithm needs to consider all possible information flows, all the symbolic class equations representing the IN parameter “x” of A, which are stored in IMPORT categories of the calling procedures, must be combined. The symbolic class equations corresponding to actual IN parameters “i”, “j” and “c” must be combined to make up one equation. This combined equation may contain security variables corresponding to the actual OUT parameter “b” and formal IN parameter “y” of B. After replacing xi with this combined equation, the security variables corresponding to “y” and “b” must in turn be replaced with the symbolic class equations corresponding to these variables.

Consider security variable b first. There is exactly one equation which corresponds to an actual OUT parameter, and it is found in the EXPORT category of the called procedure’s template. The symbolic class expression for “w” found in C’s template corresponds to security variable b. Thus, the mechanism replaces b with the expression for “w”. If the expression for “w” contains security variable z, z must now be replaced by the expression corresponding to the actual IN parameter to C. According to the link-time algorithm shown in Section 2, the symbolic class equations for “a”, “n” and “o” are all combined to replace z. The input variable to this specific invocation of C, however, is only the actual IN parameter “a”, and the other actual IN parameters “n” and “o” are not involved in this invocation. Therefore, combining the equations for “a”, “n” and “o” in this situation causes over-classification. The algorithm must be modified so that z is replaced only by the equation corresponding
Now consider security variable $y$. Information from actual IN parameters "k", "l" and "m" flows to B. Unlike the preceding case, information from any of these parameters may eventually flow to A as a result of other invocations. Thus, $y$ must be replaced by the combined expressions of "k", "l" and "m".

Figure 5.5 represents the case in which A calls B with actual IN parameter "a" and actual OUT parameter "d"; then B calls C with actual IN parameter "b" and actual OUT parameter "c". Formal IN parameters of A, B and C are "x", "y" and "z", and formal OUT parameters of A, B and C are "u", "v" and "w", respectively. A is called with actual IN parameters "i", "j" and "k". B is called with actual IN parameters "l" and "m" as well as "a". C is called with actual IN parameters "n" and "o"; as well as "b". Assume the symbolic flow equation under evaluation contains both security variables $d$ and $x$.

Consider security variable $x$ first. Procedure A may be called with each of the actual IN parameters "i", "j" and "k". Therefore, the symbolic class equations corresponding to these parameters, which are found in the IMPORT categories of the calling procedures, must be combined into one equation in order to replace $x$.

Now consider security variable $d$. The algorithm replaces $d$ with the symbolic class equation for "v" of B. The resulting flow equation may contain security variables $c$ and $y$. There is only one class equation corresponding to $c$ — that is, the equation for "w" found in the EXPORT category of procedure C. After the replacement, the flow equation may contain security variable $z$. Even though information from the three actual IN parameters "n", "o" and "b" eventually flows to C, only information from "b" flows to C as the result of this specific invocation. Therefore, the algorithm
should replace z with the equation for "b" and should not consider "n" and "o". Similarly, security variable y should be replaced with the equation for "a", and the equations for "l" and "m" should be ignored.

This leads us to the following conclusions:

1. If a security variable to be replaced in an original symbolic flow equation corresponds to a formal IN parameter or an incoming implicit inter-module flow, the symbolic class equations for the corresponding actual IN parameters or outgoing implicit inter-module flows (found in the IMPORT categories of the calling procedures) must be combined.

   For example, in Figure 5.4, security variable x corresponds to formal IN parameter "x". Therefore, the equations for "i", "j" and "c" are combined before replacing x. Similarly, in Figure 5.5, the equations for "i", "j" and "k" are combined into one equation to replace x.

2. If a security variable corresponding to a formal IN parameter or an incoming implicit inter-module flow appears in a symbolic class expression during an expansion step as the result of a sequence of replacements of security variables for formal IN parameters or incoming implicit inter-module flows, that security variable must be replaced by the combined equation of the corresponding actual IN parameters or outgoing implicit inter-module flows.

   For example, in Figure 5.4, since y appears as the result of replacing security variable x, which corresponds to formal IN parameter "x", with the equation for "c", the equations for "k", "l" and "m" must be combined to replace y.

3. If a security variable to be replaced is for an actual OUT parameter, there is
exactly one symbolic class equation for the corresponding formal OUT parameter (found in the EXPORT category of the called procedure), and the security variable is replaced by the equation.

4. If a security variable corresponding to a formal IN parameter or an incoming implicit inter-module flow appears as the result of replacement of a security variable for an actual OUT parameter, that security variable must be replaced only by the equation corresponding to the actual IN parameter or the outgoing implicit inter-module flow of the caller.

For example, in Figure 5.4, since \( z \) appears as the result of replacing \( b \) (which corresponds to actual OUT parameter "b" of B) with the equation for "w", \( z \) must be replaced by the equation for the corresponding actual IN parameter found in the template of the caller (in this case, the equation for "a" in B), and the equations for "n" and "o" should be ignored. In Figure 5.5, \( z \) appears as the result of replacing \( c \) (which is the security variable for actual OUT parameter of B) with the equation for "w". Thus, \( z \) must be replaced only with the equation for "b". The equations for "n" and "o" must be ignored.

5. As the result of the replacement described in 4, a security variable corresponding to a formal IN parameter may appear in the equation. In this case, the security variable should be replaced only by the equation for the corresponding actual IN parameter or outgoing implicit inter-module flow of the caller. If a security variable for a formal IN parameter or an incoming implicit inter-module flow appears as the result of this replacement, that security variable must also be replaced only by the equation corresponding to the actual IN parameter or the
outgoing implicit inter-module flow of the caller.

For example, in Figure 5.5, \( y \) may appear as the result of replacing \( z \). Even though \( z \) corresponds to the formal IN parameter of C, \( y \) should be replaced only by the equation for "a" and the equations for "l" and "m" must be ignored.

In order to keep track of a calling sequence, the algorithm could maintain multiple stacks. Each stack corresponds to a security variable for an actual OUT parameter in the original symbolic flow equation under certification. At first, the name of a procedure which contains the flow equation is pushed onto each stack. For each replacement of a security variable for an actual OUT parameter, the name of the procedure which contains the equation replacing the security variable is pushed onto the corresponding stack. If the replacement generates more than one security variable, a new stack is created for each security variable. Each newly created stack is an exact copy of the stack on which the name of the procedure has just been pushed. For each replacement described in steps 4 and 5, the name of the procedure which contains the corresponding equation appears second from the top in the corresponding stack. The algorithm finds the equation for the corresponding actual IN parameter or the outgoing implicit-inter module flow in that procedure and pops the top of the stack.

For example, in Figure 5.5, the algorithm creates a stack for \( d \) (which appears in the original flow equation) and pushes A onto the stack. When \( d \) is replaced by the equation for "v", the algorithm pushes B onto the stack. When replacing \( c \) with the equation for "w", C is pushed onto the stack. In order to replace \( z \), the algorithm looks for the template for B, whose name appears second from the top in the stack. Therefore, the corresponding equation in this case is the one
for "b". After replacing z with the equation for "b", the algorithm pops C from the stack. If y appears in the flow equation as the result of replacing z with the equation for "b", y should be replaced by the equation for the corresponding actual \textbf{IN} parameter of procedure A, whose name appears second from the top in the stack.

In summary, the equations for actual \textbf{IN} parameters or outgoing implicit inter-module flows should be combined to replace the security variable for a formal \textbf{IN} parameter or an incoming implicit inter-module flow only when that security variable ether is in the original flow equation or appears as the result of consecutive replacements of only formal \textbf{IN} parameters or incoming implicit inter-module flows.

Applying the modified link-time mechanism, the example program shown above is verified to be secure. For the symbolic flow equation of "h", the following expansion is applied.

\begin{align*}
SV1 & \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{O3.f.y (C1.h)} \oplus \text{h.implicit (C1.h)} \\
& = \text{SECRET} \oplus \text{f.m (C3.f)} \oplus \text{f.implicit (C3.f)} \oplus \text{LOW} \\
& \quad \oplus \text{init.implicit (C0.init)} \\
& = \text{SECRET} \oplus \text{f.m (C3.f)} \oplus \text{f.implicit (C3.f)} \oplus \text{init.implicit (C0.init)} \\
& = \text{SECRET} \oplus \text{SECRET} \oplus \text{h.implicit (C1.h)} \oplus \text{LOW} \\
& \quad \oplus \text{h.implicit (C1.h)} \oplus \text{LOW} \\
& = \text{SECRET} \oplus \text{h.implicit (C1.h)} \\
& = \text{SECRET} \oplus \text{LOW} \oplus \text{init.implicit (C0.init)} \\
& = \text{SECRET} \oplus \text{LOW} \\
& = \text{SECRET}
\end{align*}
Even though security variable \( f.m \) in the template for "C3.f" corresponds to the formal IN parameter, it appears as the result of replacement of security variable \( 03.f.y \) in the template for "C1.h", which corresponds to the actual OUT parameter "y" of procedure invocation "O3.f". Therefore, \( f.m \) must be replaced only by the equation corresponding to the actual IN parameter of the caller, which is "x" of "C1.h". Thus, \( f.m \) must be replaced by the equation

\[
x \equiv \text{SECRET } \oplus \text{h.implicit}(C1.h),
\]

found in the entry for \( 03.f(\text{IN } x, \text{OUT } y) \) in the IMPORT category of the template for "C1.h".

Similarly, \( f.\text{implicit} \) is replaced by the equation

\[
\text{implicit}(C1.h) \equiv \text{LOW } \oplus \text{h.implicit}(C1.h).
\]

For the symbolic flow equation of "g", the expansion sequence is as follows:

\[
SV_2 = \text{TOPSECRET } \rightarrow \text{TOPSECRET } \oplus \text{O3.f.b (C2.g) } \oplus \text{g.implicit (C2.g)}
\]

\[
= \text{TOPSECRET } \oplus \text{f.m (C3.f) } \oplus \text{f.implicit (C3.f) } \oplus \text{LOW}
\]

\[
\oplus \text{init.implicit (C0.init)}
\]

\[
= \text{TOPSECRET } \oplus \text{f.m (C3.f) } \oplus \text{f.implicit (C3.f)}
\]

\[
\oplus \text{init.implicit (C0.init)}
\]

\[
= \text{TOPSECRET } \oplus \text{TOPSECRET } \oplus \text{g.implicit (C2.g) } \oplus \text{LOW}
\]

\[
\oplus \text{g.implicit (C2.g) } \oplus \text{LOW}
\]

\[
= \text{TOPSECRET } \oplus \text{g.implicit (C2.g)}
\]

\[
= \text{TOPSECRET } \oplus \text{LOW } \oplus \text{init.implicit (C0.init)}
\]

\[
= \text{TOPSECRET } \oplus \text{LOW}
\]

\[
= \text{TOPSECRET}
\]
Figure 5.6 shows a more complicated example. A calls C with actual IN parameter “a” and actual OUT parameter “h”. C calls D and B with actual IN parameters “b” and “d”, respectively, and actual OUT parameters “c” and “e”, respectively. D calls E with actual IN and OUT parameters “f” and “g”, respectively. A, B, C, D and E have formal IN parameters “v”, “w”, “x”, “y” and “z”, and formal OUT parameters “k”, “l”, “m”, “n” and “o”, respectively. These modules are also called with actual IN parameters “i_j” as shown in the figure. Assume that the modified link-time algorithm is certifying a symbolic flow equation in the template for B. Since security variable w in the equation corresponds to the formal IN parameter of B, the variable is replaced by the combined equation for actual IN parameters “i_3”, “i_4”, and “d”.

Suppose that the resulting equation has security variables x and c. Since x corresponds to formal IN parameter “x” of C and appears as the result of replacing the security variable corresponding to formal IN parameter “w” of B, it must be replaced by the combined equation of actual IN parameters “i_5”, “i_6” and “a”. Security variable c corresponds to an actual OUT parameter of C. Therefore, c must be replaced by the equation corresponding to formal OUT parameter “n” of D. As the result of these replacements, security variables y, y and g may appear in the equation. Since y appears as the result of a sequence of replacement only of formal IN parameters, it must be replaced by the combined equation of “i_1” and “i_2”. On the other hand, y appears as the result of replacing the security variable corresponding to the actual OUT parameter. Thus, y must be replaced only by the equation corresponding to the actual OUT parameter of the caller, in this case, “b” of C, and the equations for “i_7” and “i_8” should be ignored. Since g corresponds to
Figure 5.6. Another Example for the Modified Link-Time Algorithm
the actual OUT parameter of D, it is replaced by the equation for the corresponding formal OUT parameter "o" of E.

After replacing g with this equation, security variable z may appear in the equation. Then, z must be replaced by the equation corresponding to "f" of D, and the equations for "i_9" and "i_10" should be ignored.

If the replacement generates security variable y in the symbolic flow equation under certification, even though y corresponds to the formal IN parameter of D and the previous replacement is for the formal IN parameter "z" of E, y must be replaced by the equation for actual OUT parameter "b" of the calling procedure C. The reason for this is that the replacement sequence to generate y includes the replacement for actual OUT parameters "c" of C and "g" of D.

5.4 The Link-time/Run-time Algorithm Allowing Dynamically Bound State Variables

In this section, we extend the approach presented in the previous sections to allow dynamically bound state variables and conditioning of the execution of a program by classified variables. Module invocations, however, will be required to use only literal module names. The extended algorithm combines the compile-time, link-time and run-time approaches. The link-time mechanism solves symbolic class equations in STATE categories and symbolic flow equations in STATIC categories in the same manner as the link-time algorithm given in Section 5.1. However, the expansion step of the new algorithm does not replace init.implicit or security variables corresponding to dynamically bound state variables and thus may not reduce a symbolic class expression to a single security class. Therefore, in general, the symbolic class expression part of a
resulting symbolic class equation or symbolic flow equation contains a single security class and security variables. These partially-solved equations are stored in a new template, called a reduced template, that has only STAT\textsc{e} and STAT\textsc{ic} categories. This template is stored in the class of the main module.

Immediately before each execution of a program (i.e., before instantiation of the main module), the run-time mechanism solves each expression in the corresponding reduced template by replacing

1. init.implicit with the class of the implicit inter-module flow from the module which initiates the program, and

2. security variables for the dynamically bound state variables with their current security classes.

In order to determine current security classes of the dynamically bound state variables, the run-time mechanism needs to send messages, called tokens, to invoked modules which contain dynamically bound state variables. The run-time mechanism can determine which modules are invoked, since only literal module names are used to specify called modules. When a module receives a token, the module must freeze itself and return the security classes of its dynamically bound state variables to the main module. If all the symbolic flow equations are verified to be secure, the execution can be carried out. The number of tokens required is the number of invoked modules (excluding the main module) which contain dynamically bound state variables. No further run-time checking is necessary.

While a module is in a frozen state it can accept only messages sent from within the execution which froze it. A module remains in a frozen state until the run-time
mechanism sends a message to unfreeze it. If the mechanism detects a potential flow violation, it immediately unfreezes all frozen modules and the program is not executed. If no potential flow violation is detected, execution of the program is begun. When the execution terminates, the classes of the dynamically bound state variables are updated and the modules are unfrozen.

The freezing mechanism is necessary because certification of a program P is performed before execution starts using the current security classes of the dynamically bound state variables appearing in P. The certification assumes these classes do not change until the execution terminates. However, they may change if during execution another program invokes a module invoked by P. This can happen even if P and the other computation invoke different procedures within the module. Thus, the mechanism must guarantee P exclusive access to all invoked modules which contain dynamically bound state variables until the execution terminates. Note that it is not necessary to freeze a module if it does not contain dynamically bound state variables or if information does not flow to or from its dynamically bound state variables during execution of a program.

The certification process performed by the link-time and run-time mechanisms is demonstrated by using the example program shown in Figure 4.5 in the previous chapter. The information flow templates generated by the compile-time mechanism are shown in Figure 5.7. The link-time mechanism solves the symbolic class equation for LS3 and the symbolic flow equation for LS1.

We first find the symbolic class expression corresponding to each security varia-
The information flow template for C1.initialize

**EXPORT**
initialize

**IMPORT**
M2.f(IN a, OUT b)
   implicit \equiv\text{SECRET} \oplus \text{init.implicit}
   a \equiv\text{SECRET} \oplus \text{init.implicit}
M3.g(IN b, c, OUT e)
   implicit \equiv\text{LOW} \oplus \text{init.implicit}
   b \equiv\text{SECRET} \oplus \text{init.implicit}
   c \equiv\text{LOW} \oplus\text{LS2} \oplus \text{init.implicit}

**STATE**

**STATIC**
LS1 \equiv\text{SECRET} \leftarrow \text{SECRET} \oplus M2.f.d \oplus M3.g.e \oplus \text{init.implicit}

The information flow template for C2.f

**EXPORT**
M2.f(IN u, OUT z)
   z \equiv M4.h.z \oplus f.implicit

**IMPORT**
M4.h(IN v, w, OUT z)
   implicit \equiv\text{LOW} \oplus f.implicit
   v \equiv\text{LOW} \oplus f.u \oplus f.implicit
   w \equiv\text{LOW} \oplus f.u \oplus f.implicit

**STATE**

**STATIC**

---

Figure 5.7. Information Flow Templates for Program with Dynamically Bound State Variables
The information flow template for C3.g

**EXPORT**

M3.g(IN l, m, OUT n)

\[ n \equiv M4.h.n \oplus g.implicit \]

**IMPORT**

M4.h(IN o, p, OUT n)

\[ implicit \equiv LOW \oplus g.implicit \]
\[ o \equiv g.m \oplus g.n \oplus g.implicit \]
\[ p \equiv LOW \oplus g.m \oplus g.implicit \]

**STATE**

**STATIC**

The information flow template for C4.h

**EXPORT**

M4.h(IN a, b, OUT c)

\[ c \equiv LOW \oplus LS3 \oplus h.a \oplus h.b \oplus h.implicit \]

**IMPORT**

**STATE**

LS3 \equiv LOW \oplus LS3 \oplus h.a \oplus h.b \oplus h.implicit

**STATIC**

Figure 5.7. (continued)
ble: \(^2\)

- \(M_{2.f.d} (C1.init) = z (M2.f) = M_{4.h.z} (M2.f) \oplus f.implicit (M2.f)\)
- \(M_{3.g.e} (C1.init) = n (M3.g) = M_{4.h.n} (M3.g) \oplus g.implicit (M3.g)\)
- \(M_{4.h.z} (M2.f) = z (M4.h)\)
  \[= \text{LOW} \oplus \text{LS3} (M4.h) \oplus h.a (M4.h) \oplus h.b (M4.h) \oplus h.implicit (M4.h)\]
- \(f.implicit (M2.f) = \text{implicit} (C1.init) = \text{SECRET} \oplus \text{init.implicit}\)
- \(f.u (M2.f) = a (C1.init) = \text{SECRET} \oplus \text{init.implicit}\)
- \(g.m (M3.g) = b (C1.init) = \text{SECRET} \oplus \text{init.implicit}\)
- \(g.n (M3.g) = c (C1.init) = \text{LOW} \oplus \text{LS2} (C1.init) \oplus \text{init.implicit}\)
- \(g.implicit (M3.g) = \text{implicit} (C1.init) = \text{LOW} \oplus \text{init.implicit}\)
- \(h.a (M4.h \text{ when called from } M2.f) = v (M2.f)\)
  \[= \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f)\]
- \(h.b (M4.h \text{ when called from } M2.f) = w (M2.f)\)
  \[= \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f)\]
- \(h.implicit (M4.h \text{ when called from } M2.f) = \text{implicit} (M2.f)\)
  \[= \text{LOW} \oplus f.implicit (M2.f)\]
- \(h.a (M4.h \text{ when called from } M3.g) = o (M3.g)\)
  \[= g.m (M3.g) \oplus g.n (M3.g) \oplus g.implicit (M3.g)\]
- \(h.b (M4.h \text{ when called from } M3.g) = p (M3.g)\)
  \[= \text{LOW} \oplus g.m (M3.g) \oplus g.implicit (M3.g)\]
- \(h.implicit (M4.h \text{ when called from } M3.g) = \text{implicit} (M3.g)\)
  \[= \text{LOW} \oplus g.implicit (M3.g)\]

\(^2\)In this example, we use \(SV(M_P)\) to denote the security variable \(SV\) appearing in procedure \(P\) of instance module \(M\) for instance modules \(M2, M3\) and \(M4\).
• h.a (in STATE of M4.h)
  \[= \text{v} (M2.f) \oplus o (M3.g)\]
  \[= \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f) \oplus g.m (M3.g)\]
  \[\oplus g.n (M3.g) \oplus g.implicit (M3.g)\]

• h.b (in STATE of M4.h) = w (M2.f) \oplus p (M3.g)
  \[= \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f) \oplus \text{LOW} \oplus g.m (M3.g)\]
  \[\oplus g.implicit (M3.g)\]
  \[= \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f) \oplus g.m (M3.g) \oplus g.implicit (M3.g)\]

• h.implicit (in STATE of M4.h)
  \[= \text{implicit} (M2.f) \oplus \text{implicit} (M3.g)\]
  \[= \text{LOW} \oplus f.implicit (M2.f) \oplus \text{LOW} \oplus g.implicit (M3.g)\]
  \[= \text{LOW} \oplus f.implicit (M2.f) \oplus g.implicit (M3.g)\]

For LSI, the link-time algorithm expands the equation as follows:

\[\text{LS1 (C1.init)} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus M2.f.d \oplus M3.g.e \oplus \text{init.implicit} \]

For M2.f.d (C1.init),

\[M2.f.d \ (C1.init) = M4.h.z (M2.f) \oplus f.implicit (M2.f)\]
  \[= \text{LOW} \oplus \text{LS3} (M4.h) \oplus h.a (M4.h) \oplus h.b (M4.h)\]
  \[\oplus h.implicit (M4.h) \oplus \text{SECRET} \oplus \text{init.implicit}\]
  \[= \text{SECRET} \oplus \text{LS3} (M4.h) \oplus h.a (M4.h) \oplus h.b (M4.h)\]
  \[\oplus h.implicit (M4.h) \oplus \text{init.implicit}\]
  \[= \text{SECRET} \oplus \text{LS3} (M4.h) \oplus \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f)\]
  \[\oplus \text{LOW} \oplus f.u (M2.f) \oplus f.implicit (M2.f) \oplus \text{LOW}\]
\[ \oplus f.\text{implicit} \quad (M2.f) \oplus \text{init.implicit} \]
\[ = \text{SECRET} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit} \]
\[ \oplus f.u (M2.f) \oplus f.\text{implicit} \quad (M2.f) \]
\[ = \text{SECRET} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit} \oplus \text{SECRET} \]
\[ \oplus \text{init.implicit} \oplus \text{SECRET} \oplus \text{init.implicit} \]
\[ = \text{SECRET} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit}. \]

For \( M3.g.e \quad (C1.\text{init}) \):
\[
M3.g.e \quad (C1.\text{init}) = M4.h.n \quad (M3.g) \oplus g.\text{implicit} \quad (M3.g) \\
= \text{LOW} \oplus LS3 \quad (M4.h) \oplus h.a \quad (M4.h) \oplus h.b \quad (M4.h) \\
\oplus h.\text{implicit} \quad (M4.h) \oplus \text{LOW} \oplus \text{init.implicit} \\
= \text{LOW} \oplus LS3 \quad (M4.h) \oplus h.a \quad (M4.h) \oplus h.b \quad (M4.h) \\
\oplus h.\text{implicit} \quad (M4.h) \oplus \text{init.implicit} \\
= \text{LOW} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit} \oplus g.m \quad (M3.g) \oplus g.n \quad (M3.g) \\
\oplus g.\text{implicit} \quad (M3.g) \oplus \text{LOW} \oplus g.m \quad (M3.g) \\
\oplus g.\text{implicit} \quad (M3.g) \oplus \text{LOW} \oplus g.\text{implicit} \quad (M3.g) \\
= \text{LOW} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit} \oplus g.m \quad (M3.g) \oplus g.n \quad (M3.g) \\
\oplus g.\text{implicit} \quad (M3.g) \\
= \text{LOW} \oplus LS3 \quad (M4.h) \oplus \text{init.implicit} \oplus \text{SECRET} \oplus \text{init.implicit} \\
\oplus \text{LOW} \oplus LS2 \quad (C1.\text{init}) \oplus \text{init.implicit} \oplus \text{LOW} \oplus \text{init.implicit} \\
= \text{SECRET} \oplus LS2 \quad (C1.\text{init}) \oplus LS3 \quad (M4.h) \oplus \text{init.implicit}. \]

Therefore,
\[ LS1 \quad (C1.\text{init}) \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus LS2 \quad (C1.\text{init}) \oplus LS3 \quad (M4.h) \]
\[ \oplus \text{init.implicit}. \]
For LS3, the algorithm expands the equation as follows:

\[
\text{LS3 (M4.h)} = \text{LOW} \oplus \text{LS3 (M4.h)} \oplus h.a \oplus h.b \oplus h.\text{implicit}.
\]

For \( h.a \) (M4.h),

\[
h.a (M4.h) = \text{LOW} \oplus f.u (M2.f) \oplus f.\text{implicit} (M2.f) \oplus g.m (M3.g) \oplus g.n (M3.g) \\
\quad \oplus g.\text{implicit} (M3.g) \\
= \text{LOW} \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{SECRET} \oplus \text{init.implicit} \\
\quad \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{LOW} \oplus \text{LS2 (C1.init)} \\
\quad \oplus \text{init.implicit} \oplus \text{LOW} \oplus \text{init.implicit} \\
= \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{init.implicit}.
\]

For \( h.b \) (M4.h):

\[
h.b (M4.h) = \text{LOW} \oplus f.u (M2.f) \oplus f.\text{implicit} (M2.f) \oplus g.m (M3.g) \\
\quad \oplus g.\text{implicit} (M3.g) \\
= \text{LOW} \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{SECRET} \oplus \text{init.implicit} \\
\quad \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{LOW} \oplus \text{init.implicit} \\
\quad \oplus \text{LOW} \oplus \text{init.implicit} \\
= \text{SECRET} \oplus \text{init.implicit}.
\]

For \( h.\text{implicit} \) (M4.h):

\[
h.\text{implicit} (M4.h) = \text{LOW} \oplus f.\text{implicit} (M2.f) \oplus g.\text{implicit} (M3.g) \\
= \text{LOW} \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{LOW} \oplus \text{init.implicit} \\
= \text{SECRET} \oplus \text{init.implicit}.
\]
Therefore,

\[ \text{LS3 (M4.h)} \equiv \text{LOW} \oplus \text{LS3 (M4.h)} \oplus \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{init.implicit} \]
\[ \quad \oplus \text{SECRET} \oplus \text{init.implicit} \oplus \text{SECRET} \oplus \text{init.implicit} \]
\[ = \text{SECRET} \oplus \text{LS3 (M4.h)} \oplus \text{LS2 (C1.init)} \oplus \text{init.implicit}. \]

Thus, the reduced template in class C1 is

**STATE**

\[ \text{LS3 (M4.h)} \equiv \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \oplus \text{init.implicit} \]

**STATIC**

\[ \text{LS1 (C1.init)} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \]
\[ \quad \oplus \text{init.implicit}. \]

Assume that \text{init.implicit} and \text{LS2} are \text{LOW} at instantiation time. Immediately before instantiation of M1, the run-time mechanism sends a token to M4 to determine the class of LS3 and freezes the module. First, consider the case in which the current class of LS3 is \text{TOPSECRET}. Then, M4 returns \text{TOPSECRET}, and the resulting reduced template becomes:

**STATE**

\[ \text{LS3 (M4.h)} \equiv \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \oplus \text{init.implicit} \]
\[ = \text{SECRET} \oplus \text{LOW} \oplus \text{TOPSECRET} \oplus \text{LOW} \]
\[ = \text{TOPSECRET} \]

**STATIC**

\[ \text{LS1 (C1.init)} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \]
\[ \quad \oplus \text{init.implicit} \]
Since there is a potential flow violation, execution of the program is prevented and the mechanism unfreezes M4 by sending a message to the module.

Next, assume that the class of LS3 is less than TOPSECRET, say CONFIDENTIAL. The resulting template becomes:

**STATE**

\[ \text{LS3 (M4.h)} \equiv \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \oplus \text{init.implicit} \]

\[ \equiv \text{SECRET} \oplus \text{LOW} \oplus \text{CONFIDENTIAL} \oplus \text{LOW} \]

\[ \equiv \text{SECRET} \]

**STATIC**

\[ \text{LS1 (C1.init)} \equiv \text{SECRET} \leftarrow \text{SECRET} \oplus \text{LS2 (C1.init)} \oplus \text{LS3 (M4.h)} \]

\[ \oplus \text{init.implicit} \]

\[ \equiv \text{SECRET} \oplus \text{LOW} \oplus \text{CONFIDENTIAL} \oplus \text{LOW} \]

\[ \equiv \text{SECRET}. \]

Since this execution is certified to be secure, M1 is instantiated and the execution is carried out. Upon termination, the run-time mechanism sends a message to M4 to update LS3 to SECRET and unfreeze the module. If M1 does not have any exported procedures, M1 is deleted.

An advantage of this approach is that sending probes is not necessary at execution time and only \( n \) tokens and \( n \) responses need to be sent before each execution, where \( n \) is the number of modules within the computation (excluding the main module)
which contain dynamically bound state variables. A disadvantage is the potential for over-classification of dynamically bound state variables. For example, consider the program segment

\[
\text{if } x = 0 \text{ then } \text{R1.P(OUT LSI)} \text{ else } \text{LS1 := 0},
\]

where LSI is a dynamically bound state variable, the classes of "x" and LS1 are CONFIDENTIAL, and "R1.P" returns a TOPSECRET value in LSI. If "x" is nonzero, the class of LS1 stays CONFIDENTIAL. However, the algorithm increases the class of LS1 to TOPSECRET regardless of the value of "x".

The correctness of this algorithm follows by arguments similar to those used in the proof of Theorem 5.3.

5.5 Comparisons of the Information Flow Mechanisms

An information flow certification method combining compile-time and run-time approaches was proposed in Chapter 4. A combination of the compile-time and link-time approaches was proposed earlier in this chapter.

In this section, we compare our proposed information flow mechanisms with previously proposed methods — Fenton's run-time mechanism, Denning's run-time mechanism, Denning's compile-time mechanism and Andrews and Reitman's compile-time mechanism. We will discuss the advantages and disadvantages of each mechanism.

We will base our comparison on several properties which we consider essential for real applications:

\[^3\text{Also, } n \text{ messages must be sent after each execution to update the classes of dynamically bound state variables and unfreeze modules.}\]
1. The ability to handle both dynamically and statically bound variables:

If a mechanism handles only statically bound variables, programmers must specify a security class for every variable in a program. This is very inconvenient. In addition, many versions of functionally equivalent procedures for parameters of different security classes must be created, thereby reducing the possibility of resource sharing. If a mechanism can handle only dynamically bound variables, it cannot be applied in real systems since it requires even classes of output devices such as terminals and printers to be dynamically bound. Secure systems cannot be constructed using such a mechanism. Some mechanisms require state variables to be statically bound in order to allow invoked modules to be specified by variables, even though other variables can be either dynamically or statically bound. Since this requirement reflects the case in most actual implementations, we do not consider this to be a significant disadvantage. Note that if a mechanism allows dynamically bound state variables, exclusive write access to those variables must be guaranteed by the mechanism.\(^4\)

2. Separate compilation of modules or procedures:

If modules are constructed in different locations or at different points in time, it may be inconvenient and costly to require that all modules be brought together for compilation at one time. Therefore, it is preferable that each procedure or module can be compiled independently of the others. Also, if a program is very large, separate compilation is a virtual necessity.

3. The ability to handle module invocations that use variables instead of literal

\(^4\)This would be normally controlled by the synchronization specification.
module names:

In object-oriented systems, invoked modules are often dynamically instantiated, passed as parameters or determined by interaction with a subject at run time. Such modules are represented by variables within a program. Mechanisms allowing dynamically bound state variables may not be able to handle such dynamically determined modules invocations.

4. Efficiency:

Since information flow checking is considered overhead by system's users, the mechanism must be efficient. Efficiency at compile time, link time and run time must be considered. However, run-time efficiency is most important since programs are run far more frequently than they are compiled and linked.

5. Precision:

It may be impossible to develop a mechanism which implements a given information flow policy precisely, and may even be impossible to determine which of two imprecise algorithms is better. However, by comparing behaviors of algorithms in particular cases, we can sometimes conclude that one algorithm is preferable to another. For example, consider the program segment

```
if a = 0 then x := y else x := z.
```

Since the execution path cannot be determined at compile time, a compile-time algorithm considers \(x\) to be \("a \oplus y \oplus z\"\) even though the class of \("x\"\) after the execution is either \("a \oplus y\"\) or \("a \oplus z\"\). On the other hand, a run-time based algorithm considers \(x\) to be either \("a \oplus y\"\) or \("a \oplus z\"\) based on the actual
execution path. In general, a run-time algorithm is better than a compile-time or link-time algorithm in the sense that it will not over-classify in the case of conditional statements.

We will refer to

1. the combination compile-time/run-time approaches proposed in Chapter 4 as "Algorithm 1", and

2. the combination compile-time/link-time approaches proposed in this chapter as "Algorithm 2".

Fenton's run-time mechanism allows separate compilation of each procedure but depends totally on run-time certification. This is inefficient and can handle only statically bound variables. Furthermore, this mechanism has a significant restriction. If an information flow violation is detected, the mechanism must skip the execution of the statement which causes the violation and continue running as though no violation had occurred. The error must be kept hidden from the user. This may put the system into an inconsistent state. Therefore, this approach may not be practical.

Denning's run-time mechanism also allows separate compilation but is very inefficient. This approach was developed for a nondistributed environment and Denning gave no suggestions as to how to expand the mechanism to work in a distributed environment. In order to adjust the mechanism for a distributed environment, the update mechanism should be replaced by sending probes as in Algorithm 1. The mechanism can handle only dynamically bound variables. At first glance, it appears that the the mechanism can easily be extended to handle statically bound variables. However, there is a problem. Consider the program segment
if \( x = 0 \) then \( y := z \),

where \( y \) is statically bound and the classes of a user, "x", "y" and "z" are UNCLASSIFIED, CONFIDENTIAL, CONFIDENTIAL and SECRET, respectively. Even if the flow from "x" to "y" is certified, the user can deduce whether "x" is zero or nonzero by observing the existence of an error message, since a flow violation occurs only if \( x \) is zero and \( y := z \) is executed. Therefore, extension of the mechanism to handle statically bound variables is not as easy as it looks and this mechanism too may be impractical.

Denning's compile-time mechanism allows separate compilation, and no certification is required at run time. However, the mechanism can handle only statically bound variables, and we consider this to be an unrealistic restriction. In her dissertation, Denning presented a possible method for handling dynamically bound parameters. However, her method does not allow a procedure to have access to long-term storage such as state variables, and this is too restrictive.

Andrews and Reitman's compile-time mechanism does not allow dynamically bound state variables. Other variables can be either statically or dynamically bound, and no certification is required at run time. We do not consider the prohibition of dynamically bound state variables to be a significant disadvantage. In order to avoid over-classification, the mechanism considers values of variables whenever possible in attempt to determine the execution paths of conditional statements. Andrews and Reitman do not discuss how well the mechanism avoids over-classification. The biggest problem of their approach is that the mechanism does not allow separate compilation of procedures. If a procedure under certification invokes other procedures, those procedures must have been previously compiled and the flow information of these
procedures must be available to the mechanism. There are difficulties in applying this mechanism to a distributed environment and constructing a large program. Also, Andrews and Reitman did not show how to handle recursive invocations under this restriction. Even though they did not suggest it in their paper, it may be easy to expand the algorithm to allow invoked modules to be determined dynamically at run time since all the state variables must be statically bound.

Algorithm 1 can handle both statically bound and dynamically bound variables, and allows separate compilation of procedures. Two approaches are possible:

1. allowing module invocations to use variables instead of literal module names and requiring all state variables to be statically bound, and

2. allowing dynamically bound state variables and requiring module invocations to use literal module names.

This mechanism may require sending many probes at run time and therefore may be inefficient. Since each procedure is certified at compile time, the mechanism is imprecise. In particular, the class of a subject (a user) is involved in the verification algorithm.

Algorithm 2 allows separate compilation and requires no certification at run time. It can use either of the approaches mentioned in Algorithm 1. The mechanism tends to over-classify variables, and this may be critical, especially if dynamically bound state variables are allowed, since it not only prevents the execution of some secure programs but may unnecessarily upgrade the classes of dynamically bound state variables.

In summary, Andrews and Reitman’s compile-time mechanism, algorithm 1 and algorithm 2 seem more useful for practical applications. Andrews and Reitman’s
mechanism is suitable when

1. no recursive invocations are involved,

2. the procedures can always be compiled in a bottom up manner on the invocation
   dependency graph.

With these restrictions, application to large programming projects may be difficult. Also, since the mechanism frequently needs to access flow information of other procedures while compiling, it is not practical unless communication costs are low.

Algorithm 1 and Algorithm 2 allow separate compilation of procedures. If dynamically bound state variables are disallowed, the algorithms allow invoked modules to be dynamically determined at run time. If invoked modules are specified only by literal names, they allow dynamically bound state variables. Algorithm 1 requires sending probes at run time. Consequently, networks in which this algorithm is applied should have low communication costs. Algorithm 2 needs no run-time overhead and is suitable for distributed computing environments in general. It is easily applied to conventional language environment. However, Algorithm 2 requires that the link-time algorithm be performed every time a class is modified. Algorithm 1 is suitable for program development. When program development is completed, Algorithm 2 is more suitable. Both Algorithm 1 and Algorithm 2 are imprecise. However, as shown in the example program in Figure 4.4, there are programs which Algorithm 1 determines to be secure and Algorithm 2 regards as insecure. The converse does not appear to be true.
6 CONCLUSION

6.1 Summary

The major thrust of this research was to investigate the problem of security within the context of a distributed object-oriented system. In Chapter 2, we proposed a "resource module" as a basic unit for structuring distributed network application software. Resource sharing is assumed to be natural and desirable and concurrency is the rule rather than the exception. Protection and synchronization are separated from the access component, leading to very-high level specification languages. Software can be constructed using modularization techniques. The system incorporates common features of object-oriented systems such as instances, classes, METACLASS and inheritance. The role of each component, the naming convention, inter-/intra-module message flows and a prototype implementation model are presented.

In Chapter 3, we introduced the concept of "four-tuple access control lists" and access-rights expressions as the basis for the specification language for a protection component of a resource module. The four-tuple access control list realizes the principle of least privilege at a procedure invocation level. By combining access-rights expressions with the four-tuple access control list, the system provides the ability to effectively enforce the use of a resource module in a way defined by the programmer and the owner of the module.
Chapter 4 described the information flow control mechanism of the protection component. The mechanism combines compile-time and run-time approaches. For each exported procedure and the initialization procedure, the compile-time algorithm creates an "information flow template" which stores "symbolic class equations" and "symbolic flow equations". The information flow template is stored in the class module which defines the procedure. At run time, an "information flow instance" is created in the protection component of the instance module for each procedure invocation. An information flow instance is a copy of the information flow template. The run-time algorithm solves symbolic class equations and symbolic flow equations on the information flow instance. This is done by replacing "security variables" appearing in the equations with the corresponding security classes carried by messages. Symbolic class equations are used to determine classes of output variables. Symbolic flow equations are used to detect flow violations.

The mechanism incorporates the following features:

1. Run-time information flow security checks are done only at message passing time.

2. Program variables can be either dynamically or statically bound to security classes. (If dynamically bound state variables are allowed, invoked modules must be specified by literal names.)

3. Each procedure can be compiled and its "internal" security established independently of other procedures. And

4. Invoked modules can be instantiated or determined dynamically at run time. (In this case, dynamically bound state variables are not allowed.)
Compile-time certification algorithms were presented for selected programming constructs.

The mechanism presented in Chapter 4 requires the sending of "probes" in order to account for implicit flows between module boundaries which occur when procedure invocations are skipped. The run-time overhead due to sending probes may be expensive in certain systems where communication costs are very high.

In Chapter 5, we presented an alternative approach which requires no run-time certification. The mechanism relies on compile-time and link-time certifications. The link-time algorithm is performed after all of the invoked exported procedures in associated modules have been compiled and their information flow templates established by the compile-time algorithm. To certify a program, the algorithm solves symbolic flow equations in the templates of procedures which are invoked within the program. This approach does not allow dynamically bound state variables, but module invocations may use variables to specify invoked modules. Since actual execution paths of a program are not known at link time, the link-time algorithm must account for all possible information flows. Hence, the algorithm sometimes over-classifies variables. In Section 5.3, we proposed a modified link-time algorithm which considers, to some extent, the execution paths of a program in order to reduce the over-classification.

For a system with dynamically bound state variables, a mechanism which is based on a combination of compile-time, link-time and run-time certifications was presented. The link-time algorithm solves symbolic class and flow equations in the same manner as the link-time mechanism given in Section 5.1. However, the security classes of the incoming implicit flow from a module which instantiates the main module (denoted by \texttt{init.implicit}) and of dynamically bound state variables are left as security variables.
These partially-solved equations are stored in the reduced template. Immediately before each execution of a program, the run-time algorithm solves each equation in the corresponding reduced template by replacing `init.implicit` with the actual implicit flow from the instantiating module and replacing the security variables for dynamically bound state variables with the current security classes of those variables. It then certifies the program. In order to determine the security classes of dynamically bound state variables in remote modules, the run-time algorithm needs to send "tokens". The number of tokens required is the number of invoked modules (excluding the main module) which contain dynamically bound state variables and no further run-time checking is necessary. This approach requires that all module invocations use literal module names.

Finally, we compared the proposed information flow mechanisms with the previous work done by Fenton, Denning, and Andrews and Reitman.

### 6.2 Areas of Future Research

The final goal of our research is to implement the system we have described and to evaluate its usefulness. Before implementation, however, we need to solve some theoretical problems.

#### 6.2.1 Access component

The final version of the language for specifying an access component has not been defined. In order to allow exploitation of concurrency at all levels, the final language may be modeled after functional languages or data flow languages [1,18,20].

The information flow mechanism assumes the existence of a recovery routine in
order to maintain system consistencies when an information flow violation is detected. Our future research plans include an investigation of an efficient recovery mechanism for distributed object-oriented environments.

6.2.2 Protection component

Since the language for the access component has not been defined, we assumed a PASCAL-like conventional structured language constructs and developed compile-time information flow certification algorithms for several statements including the assignment statement, if statement, and while statement. After a specification language for an access component is defined, the run-time certification algorithms may need to be expanded.

As suggested by Andrews and Reitman, a synchronization statement may cause an information leakage. Future research plans include an investigation of a possible information leakage due to extended open predicate path expressions in the synchronization component.

Also, we would like to apply our results to the development of certification semantics for other synchronization primitives such as synchronizing resources [5] and ADA-like selective message passing [48].
7 BIBLIOGRAPHY


8 ACKNOWLEDGMENTS

I would like to express my deep appreciation to my thesis advisor, Dr. Arthur E. Oldehoeft, for providing skillful encouragement and guidance throughout my stay at Iowa State University. I am also grateful to the other members of my committee, Dr. Arthur V. Pohm, Dr. Deepinder Sidhu, Dr. Robert M. Stewart and Dr. George O. Strawn for their suggestions and support.

Discussions with Mr. John H. Leuchner proved valuable in formalizing my arguments. Earlier conversations with Dr. Giora Slutzki and Dr. Rattikorn Hewett helped to clarify the logic behind the link-time information flow certification algorithm described in Chapter 5. I am also thankful to Mr. Brian C. Justice for his comments and encouragement.