INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9110540

Multilevel security and concurrency control for distributed computer systems

Moukaddam, Samir, Ph.D.

Iowa State University, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Multilevel security and concurrency control for distributed computer systems

by

Samir Moukaddam

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

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1990

TABLE OF CONTENTS

ACKNOWLEDGEMENTS viii

1. INTRODUCTION 1

1.1 Goals 1

1.2 Assumed Model 3

1.3 Outline of Dissertation 4

2. MULTILEVEL SECURITY AND INFORMATION FLOW CONTROLS 6

2.1 Introduction and Definitions 6

2.2 The Bell and LaPadula Model 8

2.3 Information Flow Controls Based on the Lattice Model 11

2.3.1 Explicit and implicit flows 13

2.4 Various Existing Information Flow Mechanisms 14

2.4.1 Fenton's run-time mechanism 16

2.4.2 Denning's compile-time certification mechanism 17

2.4.3 Denning's run-time mechanism 19

2.4.4 Andrews' and Reitman's compile-time mechanism 20

3. TRANSACTIONS 22

3.1 Introduction and Definitions 22

3.2 Recovery Techniques 30
3.2.1 Shadowing 31
3.2.2 Logging 32
3.2.3 Distributed atomic commit 32
3.3 Concurrency Control 33
3.3.1 Two-phase locking 33
3.4 Moss' Nested Transactions 35
3.4.1 Implementing remote procedure calls 37
4. A MODEL FOR MULTILEVEL SECURITY FOR DISTRIBUTED SYSTEMS 39
4.1 The Resource Module Model 40
4.2 The Information Flow Control Mechanism 43
4.2.1 Overview 43
4.2.2 Discussion on checking implicit flows 45
4.2.3 The compile-time mechanism 60
4.2.4 The run-time mechanism 78
4.3 The Protected Resource Module Model 85
4.4 Shortcomings of the Flow Model 86
4.4.1 Restrictions on concurrency 86
4.4.2 Overclassification of dynamically-bound state variables 89
5. A CONCURRENCY CONTROL MECHANISM FOR THE PROTECTED RESOURCE MODULE SYSTEM 91
5.1 Introduction 91
5.2 Overview 94

5.3 An Initial Locking Mechanism 97
5.3.1 Lock usage at run time 101
5.3.2 The flow control mechanism revisited 112
5.3.3 Lock compatibility 116
5.4 Reducing the Number of Lock Types 120
5.4.1 Combining the write-lock types 121
5.4.2 Combining the read-lock types 122
5.4.3 The 3-type lock scheme 123
5.4.4 Usage of the three lock types 124
5.4.5 Locking and recovery rules for nested transactions 125
6. PRECISE CALCULATIONS OF CLASSES OF DYNAMIC VARIABLES 127
6.1 Introduction 128
6.2 Basic Blocks and Flow Graphs in the PRM System 131
6.2.1 Procedure syntax 131
6.2.2 Basic blocks 131
6.2.3 Flow graphs 136
6.2.4 Summary of notations 142
6.3 Determining Implicit Flows into Basic Blocks 142
6.4 Generating Flow Graph S-Definitions 147
6.4.1 Compound statements 147
6.4.2 If-then statements 149
6.4.3 If-then-else statements 151
6.4.4 While statements 155

6.4.5 Run-time support for evaluating s-definitions 167
6.5 Examples 171
6.5.1 Example 1 171
6.5.2 Example 2 178
6.6 Summary 186
6.7 Precision of the Modified Flow Control Mechanism 189
6.7.1 Precision of the class updating scheme 190
6.7.2 Precision of the security mechanism 191
7. PROBE IMPLEMENTATION ISSUES 196
7.1 Analysis of Probe Performance 196
7.2 Parallel Sending of Probes 197
7.3 Probe Tree Computation 197
7.4 Example 201
8. CONCLUSION 205
8.1 Summary 205
8.2 Areas for Future Research 208
8.2.1 Improvement of the precision of the security mechanisms . . . 208
8.2.2 The locking mechanism 209
8.2.3 Different concurrency control methods 209
BIBLIOGRAPHY 210

LIST OF FIGURES

Figure 3.1: Compatibility table for read and write locks 34

Figure 4.1: An example representation of an RM object 42

Figure 4.2: Implicit flow checking across modules 56

Figure 4.3: Violation in the absence of checking SUBJECT 57

Figure 4.4: Input and output variables of a procedure 61

Figure 4.5: An example illustrating module-recursive calls 70

Figure 4.6: An example illustrating module-recursive calls 73

Figure 4.7: Summary of notations 74

Figure 4.8: An example information template 76

Figure 4.9: Code for the example template 77

Figure 4.10: An example flow instance 79

Figure 4.11: Example of a probe tree 82

Figure 4.12: Components of a resource module object 85

Figure 4.13: Undetected illegal flows in the absence of concurrency control 88

Figure 5.1: An RM object with a concurrency control component 92

Figure 5.2: Tree showing subtransactions handling probes and RPCs . . 96

Figure 5.3: A general template 102

Figure 5.4: Compatibility matrix for class and value locks 120

Figure 5.5: Compatibility matrix for class and value locks 123

Figure 6.1: A flow graph 130

Figure 6.2: Representations of basic blocks 133

Figure 6.3: The four types of dsv basic blocks 136

Figure 6.4: Flow graphs of the various statements 139

Figure 6.5: An example flow graph 140

Figure 6.6: Summary of flow graph notations 143

Figure 6.7: A while statement as a succession of if-then statements . . . 157

Figure 6.8: Notations for while loops 158

Figure 6.9: Example 1 172

Figure 6.10: Flow graphs for example 1 173

Figure 6.11: Templates for example 1 174

Figure 6.12: Example 2 180

Figure 6.13: Example 2 templates (omitted parts are empty) 181

Figure 6.14: Flow graphs for example 2 182

Figure 6.15: Precision of our security mechanism 191

Figure 7.1: Example for computing a probe tree 202

Figure 7.2: Example probe tree information 203

ACKNOWLEDGEMENTS

I am indebted to Dr. Arthur E. Oldeheoft, my thesis supervisor, for his guidance during our always enlightening discussions. I also thank him for his patience, encouragement, and assistance.

I am also grateful to the members of my program-of-study committee: Dr. Don Pigozzi, Dr. Arthur Pohm, Dr. Gurpur Prabhu, and Dr. Johnny Wong.

I would also like to thank the participants in the various seminar classes I took. I have greatly benefited from the, sometimes heated, discussions that used to take place during our meetings.

1. INTRODUCTION

1.1 Goals

Securing information from abuse is a problem that nearly all computer systems try to tackle. The problem is aggravated by the building of networks connecting different computer systems. As computer system networks become larger, computer security becomes harder to design and implement. At the same time, computer security becomes more important as more users share and get access to the systems. Computer security is crucial to prevent the users from abusing any component system in a networked system.

Computer security has many aspects. In our work, we study *multilevel security.* Multilevel security deals with the problem of controlling the flow of classified information. This type of *information flow control* involves associating *security classes* with all users and stored information. To implement multilevel security, an information flow control *mechanism* prevents the leakage of information from a certain security class level into a lower security class level.

We will present multilevel information flow control mechanisms for distributed systems. In addition to checking the security of computational activities within individual sites in a network, they are also designed to provide multilevel security for communications among the sites. This is done by preventing the circulated messages

from causing illegal flows.

The distributed systems augmented by our security mechanisms are multilevel secure. To render them more practical, an important property we require of them is to allow different processes to concurrently access shared data. If left uncontrolled, such concurrent accesses may introduce intolerable inconsistencies in the shared data. In particular, inconsistencies may appear in the security information which the security mechanisms rely upon. As a result, uncontrolled concurrency may jeopardize the correct behavior of the security mechanisms and allow undetected illegal information flows.

We will present *concurrency control mechanisms* that ensure the consistency of the security information and shared data. The concurrency control techniques are to prevent unsafe interferences among parallel processes.

Finally, we deal with the precision of our mechanisms. If a system employs a security mechanism that prevents all illegal flows, then it is said to be *secure*. If, in addition, in no case does it reject any legal flows, it is said to be *precise*. Because of existing theoretically proven facts, it is impossible to build a secure system that is also precise. However, we can always seek to increase the precision of our mechanisms. We will present schemes to achieve that goal. Those schemes are needed to improve upon the practicality of our security mechanisms.

In the rest of this chapter, we present our assumed computational model. Then, we briefly outline the rest of the dissertation.

1.2 Assumed Model

We work under the assumption that a distributed system exists. The networked components of the systems may reside at different locations within short or long distances from each other. In addition, different networks can communicate through gateways giving rise to even larger systems.

The network may be controlled by a *network operating system* [33]. Each computer may be running a different local operating system and different protocols are adopted to allow those systems to communicate. The user on such a network is generally not shielded from the details allowing him the use of services of the different machines.

On the other hand, *distributed operating systems* [33] are designed to render the existence of multiple machines transparent. To the user of such a system, the networked systems run as one. For example, a distributed operating system will support a single system-wide file system accessible from all component machines. The user need not concern himself about specific machine details.

For our purposes, either systems are considered to support computations that may involve many different machines and software components. This type of distributed computing creates a powerful environment for software development and other programming needs. We expect that such an environment provides a programming language with the following characteristics and features.

A *data abstraction* mechanism and facilities to define *classes* (or *modules*) should be available. A class is a data type that describes the behavior of a collection of *objects* or *class instances*. We assume that all physical and program objects in the system and their characteristics are modeled by instance objects. The objects encapsulate

instances of the data defined in their classes. They respond to requests to execute the exported procedures which specify the only permitted ways of access to the encapsulated data. The state of an object is the state of its encapsulated data.

Objects can reside anywhere on the system. However, any one object cannot be split between different machines. Objects communicate through exported procedure requests or *remote procedure calls.* Communication between different objects via shared memory is excluded. Different objects can be active simultaneously. Each of them can have different procedure calls being serviced concurrently.

We are lead to the conclusion that our purposes are served by a *distributed object-oriented* language and environment. Such an environment allows us to abstract the aspects of the underlining system and to focus on the behavior of the classes and their instances in the system. The system is viewed as a collection or network of concurrently running distributed interdependent objects communicating via remote procedure calls.

In summary, for our purposes, we assume the existence of a distributed object-oriented programming environment.

1.3 Outline of Dissertation

First, in Chapters 2 and 3, we briefly survey some of the relevant existing works on multilevel security and on concurrency control and recovery techniques.

In Chapter 4, we present our information flow control mechanisms. Those mechanisms provide multilevel security for distributed computer systems. The mechanisms are modifications and enhancements of the work found in [27]. For efficiency reasons, instead of using a pure run-time security checking approach, the mechanisms combine

compile-time and run-time checking.

The flow control mechanisms lack a mechanism for the preservation of the consistency of data in the presence of concurrency. In Chapter 5, we present a concurrency control mechanism that uses a locking scheme to prevent any undesirable interferences among parallel processes.

In Chapter 6, we continue to enhance the information flow control mechanism. We present a scheme that uses flow graphs to compute the precise security classes for dynamically-bound variables. Such a scheme is needed because we avoid using a pure run-time approach in which classes are computed at every operation updating the variable. The class computing scheme avoids overclassifying variables and also renders the mechanism more precise in accepting secure programs.

In Chapter 7, we briefly discuss an important implementation issue pertaining to the proposed mechanism.

Finally, we conclude our work in Chapter 8 by summarizing the features of our system.

2. MULTILEVEL SECURITY AND INFORMATION FLOW CONTROLS

In this chapter, we present the concept of multilevel security. Various existing information flow control mechanisms are described.

2.1 Introduction and Definitions

A program that can retain information can potentially cause security violations. It can leak confidential data to which it has access. For example, a service program (e.g., a compiler) can retain confidential information passed to it as parameters by its caller process. Then, it can leak this information to another untrusted process. This is called the *confinement problem* [21, 23].

One conceptually simple way to deal with this problem is by producing programs that are prevented from leaking and retaining any of the information contained in their parameters, confidential or not, and thus, they are made *memoryless* and *confined.* This way a user of such service programs is assured that none of the confidential information that is made available to them can be leaked during their execution or after their completion. Note that in this solution, confined programs can only make calls to other programs that must also be confined. A system consisting of confined service programs is very rigid in allowing modifications of existing services or addi-

tions of new ones. The situation is worse when dealing with a distributed system. Message passing in such systems needs to be severely restricted if programs are to be confined and prevented from leaking information through messages.

More importantly, the scheme suffers from a problem of trust. It is not possible for a suspicious user, by relying completely on the above technique, to be assured that the service programs he uses are confined and secure. A designer can produce a "leaky" program while still advertising it as being confined. The user cannot be trustful of programs that are presented to him. At the same time, the owner of a service program cannot completely trust the user. He cannot afford to always be confident that a user will not steal confidential information contained in the program. This is the mutual suspicion problem mentioned in [9].

A more flexible solution to the above security problems can be achieved by introducing a *flow policy* which specifies the allowed transfers of information or *information flows*. Specifically, the policy should not allow the leakage of confidential data to untrusted processes satisfying the security concerns of both users and designers of programs. However, a flow policy should not place any unnecessary restrictions on the transfers of nonconfidential data. Specifying such a policy is called the *selective confinement problem* [11, 9]. A flow-policy solution of this problem still allows programs in a system to retain or transfer public or nonconfidential data. So, programs are not strictly confined.

Most *flow control mechanisms* that enforce flow policies use the concept of a *security class* (as used for military security). A security class associated with a data object reflects the sensitivity level of the information it contains [22]. A flow control policy can specify a finite set of *multilevel* security classes that correspond to the

different sensitivity levels that can be attached to the various types of information in a system. The security classes are used to regulate the flow of information at the different levels of sensitivity. For example, information with a security class corresponding to a certain sensitivity level should flow only to processes that are trusted at that level or at a higher level.

A simple flow policy [11] may use two security classes: confidential (C) and nonconfidential (N). The only prohibited flows in such a policy consist of the flows from class C to class N. All other flows are allowed. This simple policy can be used by a system to selectively confine service programs: the output channels of the program that are not directed to the user of the service are assigned security class N. This way the program can output or return confidential data of class C only to the user that called the service. Only the leakage of confidential information to other processes through the output channels of the program is prevented by the flow policy. Transfer of information with class N is not affected.

2.2 The Bell and LaPadula Model

Access control policies are used to protect data objects by regulating accesses to them [9]. The active entities of a system, the *subjects,* are given different access *rights* to the protected entities of the system, the *objects.* Subjects can be protected by also classifying them as objects. Access rights typically include read, write, and execute rights. An access control policy specifies that subjects can only access objects to which they have access rights. Also, they may access these objects only in the way the specific rights, which they have, allow. For example, subject S may read file F only if it has a read access right for F.

Access controls alone cannot regulate the manipulation of the information contained in the objects. As a result, they do not protect against any type of leakage of information and cannot solve the selective confinement problem.

The Bell and LaPadula flow control mechanism is an extension of access control mechanisms that can be used to prevent information leaks over *legitimate* and *storage* channels [21]. (Leaks over *covert* channels are not handled.) This security model designed by Bell and LaPadula is the main model used to build secure military and government systems [3, 13].

The Bell and LaPadula model assigns a *clearance* to each subject (such as an active process) and a *classification* to each object (such as a file). Clearances and classifications are represented by security classes.

Military systems [13] employ a multilevel security policy in which security classes consist of pairs of the form (A, C) where A is an *authority level* and C is a *category.* The authority levels are UNCLASSIFIED (U), CONFIDENTIAL (C), SECRET (S), and TOPSECRET (T). An order is defined on the authority levels:

$$U < C < S < T.$$

Given two authority levels, A and B, we write

$$A \leq B$$

if and only if $A < B$ or A is the same level as B.

Categories are subsets of a set of military *compartments* such as Nuclear, Atomic and Nato. The subsets Atomic and Nato, Atomic are examples of categories.

The set of security classes is partially ordered. For two security classes, (A1, C1) and (A2, C2), we write

$$(A1, C1) \leq (A2, C2)$$

if and only if $A1 \leq A2$ and $C1 \subseteq C2$. For example,

$$(S, \{Atomic\}) \leq (T, \{Nato, Atomic, Nuclear\})$$

is true, but (S, {Atomic}) and (T, {Nato, Nuclear}) are not comparable.

The Bell and LaPadula model restricts accesses to objects by subjects using the following two rules.

- The *simple security rule* states that no subject with clearance C1 may read an object with classification C2 unless $C2 \leq C1$.
- The **-property* states that no subject may transfer information from an object with classification C2 to an object with classification C3 unless $C2 \leq C3$.

The above rules ensure that reading information is allowed only from lower classes ("no read up") and writing information is allowed only to higher classes ("no write down"). The rules are enforced by giving the subjects the appropriate restricted access rights to the objects in the system.

The Bell and LaPadula Model was found to have problems [22, 11, 27] some of which are listed below.

1. The protection rules do not deal with multilevel objects. They are designed to deal with large objects (such as files) as single-level objects.
2. Many programs that are in fact secure are disallowed by the rules of the model. This lead to the addition of the trusted process concept to the model. Trusted processes are created to deal with special situation that call for violating the

rigid rules. However, the rules determining which processes can be trusted and which cannot are not specified by the model.

3. The model does not take into consideration the internals of individual programs and processes. Instead, it relies on external information such as the clearance of the subjects on behalf of which processes are running.

2.3 Information Flow Controls Based on the Lattice Model

In [7], it was recognized that the security classes along with the imposed partial order on them form a multilevel lattice structure that can be exploited to construct an information flow policy. An important property of such a policy is that it does not impose a set of rules on the access rights a user may have. Instead, the flows are restricted solely on the basis of the information transfers that the lattice structure allows. The clearance of a subject is not made a factor in deciding which flows are permissible; the focus is on the actual flow of information among objects rather than on the specific access rights that a subject may acquire.

The lattice model allows the information flow policy to be more precise. It allows for mechanisms that regulate information transfers by directly examining the flows among objects at the statement and variable level in the programs. This property is lacking in the Bell and LaPadula model which is based on a model that uses access rights to protect single-level objects.

An information flow policy based on the lattice structure can formally be defined as a pair, (SC, $\leq$), where SC is a finite set of *(security) classes* and $\leq$ is a partial order binary relation on SC [7]. (SC, $\leq$) must also have the property that any two security classes in SC have a unique *greatest lower bound* and a unique *least upper*

bound. Thus, the flow policy forms a *lattice.* (It is argued in [9] that any flow policy can be transformed into a lattice).

We say that there is an information flow from an object x to an object y whenever information in x is transferred directly to y or used to derive information transferred to y. Basically, after a flow from x to y, new information about x can be derived by examining the information in y.

The partial order $\leq$ in a flow policy is called a *flow relation.* It is defined as the following relation

$$\{(A, B) \mid A, B \text{ are security classes and class A information is allowed to flow into class B objects}\}.$$

All information flows in a system must obey the flow relation of a flow policy. This is the only requirement that should be imposed to render a system *secure.* A flow control mechanism can enforce the policy by rejecting insecure flows. However, it is impossible for a mechanism to completely avoid rejecting some secure flows also; in this sense, all mechanisms are said to be imprecise. Building a *precise* mechanism that ensures security has been proven to be theoretically impossible [18].

Objects in a system are assigned security classes in one of two ways. An object can be *statically bound* to a class. In this case, the class is constant over the lifetime of the object.

On the other hand, the class of an object can vary depending on the contents of the object. Thus, allowing objects to be *dynamically bound* to a variable class.

In what follows, we introduce some useful notation.

- The lowest and highest classes in a policy (SC, $\leq$) are denoted by LOW and

HIGH, respectively. LOW information may flow into any other class. HIGH information may only flow into HIGH.

- The class of an object x is denoted as $\underline{x}$.
- For any two classes A and B, A $\oplus$ B and A $\otimes$ B denote the least upper and greatest lower bounds of A and B, respectively.
- A flow from object x to object y is denoted as x $\Rightarrow$ y.

2.3.1 Explicit and implicit flows

There are two types of information flows. A flow x $\Rightarrow$ y is *explicit* if the operations causing it do not depend on the information flowing from x. The simplest example of an explicit flow is the flow caused by an assignment statement, such as `y = x` where an obvious transfer of information occurs independent of the value of x.

To illustrate the other type of flow, consider the following statements that include a conditional assignment:

```
y = 1;
if (x == 0)  then y = 0.
```

After execution of the above statements, some information about x can be deduced by looking at the value of y. For example, if y has value 0, then the value of x must be 0. So, obviously, there is a flow from x to y. This flow exists even if the assignment `y = 0` is not executed (in which case, x is not 0 and y must be 1). This flow, x $\Rightarrow$ y, exists because the statement that specifies a flow to y is conditioned on x. Flows of this type are called *implicit*.

2.4 Various Existing Information Flow Mechanisms

The lattice structure can be exploited in many ways to simplify the flow control mechanisms. We present some examples to illustrate how the properties of lattices can be used in such mechanisms [9].

Example: Consider the following two assignment statements:

```
z = x;
y = z;
```

The statements specify two direct flows, x $\Rightarrow$ z and z $\Rightarrow$ y, and an indirect flow, x $\Rightarrow$ y. To check if the flows are to be permitted, a mechanism need only check that $\underline{x} \leq \underline{z}$ and $\underline{z} \leq \underline{y}$, which imply $\underline{x} \leq \underline{y}$, by transitivity of $\leq$. In general, flow mechanisms can take advantage of the transitivity of the flow relation, to check the security of a sequence of statements by checking individually the security of the statements in the sequence.

The lattice structure guarantees the existence of greatest lower and least upper bounds for any group of security classes. This property can be exploited to simplify a flow mechanism based on the lattice model.

Example: The statement

```
y = x1 * x2 + x3
```

specifies the flows x1 $\Rightarrow$ y, x2 $\Rightarrow$ y, and x3 $\Rightarrow$ y which are permissible if $\underline{x1} \leq \underline{y}$ and $\underline{x2} \leq \underline{y}$ and $\underline{x3} \leq \underline{y}$ all hold. However, it suffices for a mechanism to check if

$$\underline{x1} \oplus \underline{x2} \oplus \underline{x3} \leq \underline{y}$$

where $\underline{x1} \oplus \underline{x2} \oplus \underline{x3}$ is the unique least upper bound of the security classes $\underline{x1}$, $\underline{x2}$ and $\underline{x3}$.

Example: Finally, consider the conditional statement

```
if (x == 0) then
   begin
     y1 = 0;
     y2 = 0;
     y3 = 0;
   end
```

which specifies the implicit flows $x \Rightarrow y1$, $x \Rightarrow y2$, and $x \Rightarrow y3$. It suffices to check if

$$\underline{x} \leq \underline{y1} \otimes \underline{y2} \otimes \underline{y3}$$

holds, where $\underline{y1} \otimes \underline{y2} \otimes \underline{y3}$ is the unique greatest lower bound of the classes $\underline{y1}$, $\underline{y2}$, and $\underline{y3}$.

Several information flow mechanisms, which are considered to follow and exploit the properties of the lattice flow model, have been suggested. Such mechanisms are designed to enforce security either at run time or at compile time. Furthermore, they can choose to handle program variables that are statically or dynamically bound to security classes. In run-time mechanisms, security of the flows is checked as the statements that cause them are executed. Statements that result in insecure flows are not executed. In compile-time mechanisms, security of the flows is checked before the programs execute. Programs that are found to contain flows that may cause security violations are rejected.

2.4.1 Fenton's run-time mechanism

The mechanism presented here deals with statically bound variables only. Each memory location is assumed to include a tag field that holds a security class. In addition, a security class, $\underline{pc}$, is associated with the program counter to allow for run-time checking of implicit flows. Initially, $\underline{pc}$ is LOW.

In the case of static binding of variables to security classes, an assignment statement causing an explicit flow can be verified at run time by making sure that the least upper bound class of the right-hand side is allowed to flow into the left-hand side. For example,

```
y = x1 + x2 - x3
```

is secure if

$$\underline{x1} \oplus \underline{x2} \oplus \underline{x3} \leq \underline{y}$$

holds when the assignment is executed. The assignment is skipped or aborted if it causes an insecure flow.

The class of the program counter is computed as the least upper bound of all the classes implicitly flowing from conditionals. Implicit flows are handled by using a stack to hold the successive classes of the program counter. Whenever a conditional statement is executed, the class of the program counter is saved and then updated. For example, executing

```
if (x == 0) then y = 1
```

causes the mechanism to push $\underline{pc}$ on the stack and update it to hold $\underline{pc} \oplus \underline{x}$. If the value of x is 0, in addition to checking the explicit flow into y, the mechanism verifies that $\underline{pc} \leq \underline{y}$.

On the other hand, if the value of x is not zero, then the implicit flow need not be verified. This check is not necessary, for it is proved in [15] that in the static binding case, it is sufficient to check the security of implicit flows at the time of explicit assignments only[1].

A major restriction of the mechanism is that attempted security violations may not be reported to the user even if the insecure statement is skipped; producing an error message may result in leaking classified information [9]. It is extremely impractical to keep the user uninformed about unexecuted statements in his program. Among other things, this makes his debugging task nearly impossible.

2.4.2 Denning's compile-time certification mechanism

In [10], a program certification mechanism is presented. The mechanism assumes static binding and can be integrated into a compiler. Programs submitted to such a compiler are either certified as secure or rejected. Rejected programs do not always result in insecure flows when run; the mechanism is not precise.

Certification semantic rules are presented for each syntactical language construct. Those rules are checked at compile time by examining the flows that a program specifies. For example, an assignment statement of the form

$$b = f(a_1, ..., a_n),$$

where $f(a_1, ..., a_n)$ represents an expression involving the variables a_1 through a_n, is certified as secure if the compiler finds that

$$\underline{a_1} \oplus ... \oplus \underline{a_n} \leq \underline{b}$$

[1] A more complete discussion on checking implicit flows is included in Chapter 4.

holds. Otherwise, the whole program is rejected.

Additional security conditions are presented for compound, conditional, iterative and procedure call statements.

For procedure calls of the form

$$q(a_1,...,a_m,b_1,...,b_n),$$

where $a_1,...,a_m$ are actual input arguments and b_1, ...,b_n are actual output parameters, the security conditions require that

1. the body of q (as a compound statement) is secure and
2. $\underline{a_i} \leq \underline{x_i}$ where x_i is the formal parameter corresponding to a_i and
3. $\underline{y_i} \leq \underline{b_i}$ where y_i is the formal parameter corresponding to b_i.

The mechanism is efficient but it has a major drawback. Since all variables are statically bound and because of the above procedure certification rule, one copy of some procedure can only handle calls with parameters at a specific security level. To be generally accessible, any system library function must be duplicated to accommodate calls with parameters of different classes. This scheme becomes an impediment to code sharing (via inheritance, for example).

To avoid having multiple copies of the same code, another solution is to always assume that each output parameter of a library procedure is a function of all its input parameters. Then, the security of procedure calls can be verified independently of the classes of the formal parameters. That is by replacing the above second and third security conditions by

$$\underline{a_1} \oplus ... \oplus \underline{a_n} \leq \underline{b_1} \otimes ... \otimes \underline{b_n}.$$

However, this solution results in the rejection of many otherwise secure procedure calls [27] since the security condition is too strong for most procedure calls.

2.4.3 Denning's run-time mechanism

This run-time mechanism allows dynamically bound variables only. It is an extension of Fenton's work to handle checking implicit flows in the presence of variables with varying security classes.

The idea of using dynamic binding is to allow the class of a variable to vary according to the class of its contents. For example, if y is dynamically bound, then executing the assignment

$$y = f(a_1, ..., a_n)$$

also updates the class of y to

$$\underline{y} = \underline{a_1} \oplus ... \oplus \underline{a_n} \oplus \underline{pc}.$$

Here, $\underline{pc}$ is the class of the program counter that represents implicit flows. However, this updating mechanism is not sufficient to correctly compute classes of variables in the presence of implicit flows, as shown in examples in [9, 15]. Contrary to the case when only statically bound variables are allowed, in the presence of dynamic binding, it is not sufficient to deal with implicit flows only at explicit assignment time. The implicit flows must be included when updating classes of variables even if assignments are skipped[2].

In [7], Fenton's run-time mechanism is augmented by a compile-time mechanism which starts by analyzing the flows in a program. Then, it inserts instructions to

[2]Again, refer to Chapter 4 for a more elaborate discussion.

update the class of variables that receive implicit flows. The update consists of increasing the class of the variables by pc. The update instructions are executed in all cases even if the explicit flow into the variable does not occur. This scheme handles the implicit flows even in the case the explicit flows are skipped at run time.

2.4.4 Andrews' and Reitman's compile-time mechanism

Andrews and Reitman [2] developed a mechanism based on proof rules in which the flow requirements are written as assertions about the values and classes of the program variables [9].

They also divided implicit flows into two types. *Local* implicit flows are ones that occur as a result of an if-statement or a loop. For example, in the following statements

```
if (x == 0) then y = 0 else z = 1;
while (a > 0) do b = b + 1;
```

there is are local implicit flows from x to y and z and from a to b.

Global implicit flows occur as a result of a loop. The global flow is from the condition of the loop to the statements that logically may follow the body of the loop. For example, in the following statements

```
y = 1;
while (a > 0) do b = b + 1;
if (x == 0) then y = 0;
```

in addition to the the local implicit flows, there is a global flow from a into y. That is because, we can deduce some information about the value of a by checking the value of y. For instance, if we know that y is 0, we can deduce that the value of a is less than or equal to 0, since otherwise, the control cannot reach the if-statement.

3. TRANSACTIONS

In this chapter, we present the notion of data consistency. We motivate the general problem of controlling concurrency to preserve the consistency of data. We present the concepts transactions and of nested transactions.

3.1 Introduction and Definitions

In our proposed distributed model, modules export procedures that may be called by procedures in other modules. Procedure invocations are serviced by independent processes within a module. All the processes have shared access to state variables in a module since they can run in parallel. As a result of allowing this type of internal concurrency, we achieve better performance since invoking exported procedures is the only way to access the information the modules encapsulate.

Two types of information are connected with the state variables of a module: their values and their security classifications. Being allowed concurrent access to the encapsulated information, different processes may interfere with each other whenever they try to access the same state variables. Such interference, if left uncontrolled, can cause problems and unexpected behavior with respect to both the values and classifications of state variables. We present a classical example to illustrate such problems.

Assume we have a variable, *saving,* that can be accessed by the following operation:

```
Deposit(real sum) {
        real t;

        t := Read(saving);
        t := t + sum;
        Write(saving, t);
}
```

Suppose that *saving* is 1000 before two calls to Deposit, Deposit(500) and Deposit(200), are made. Two concurrent processes are started to service the calls.

It is possible that both processes read the value of *saving* as 1000 before either has a chance to update it. Then, after both are finished, the final value of *saving* will be either 1500 or 1200. In either case, the result of only one of the two deposits is reflected in the final value. This is obviously an intolerable unexpected result which is directly brought about by the uncontrolled sharing of the variable *saving* between two concurrent processes. This type of problem is called the *lost updates anomaly* [4].

The *inconsistent retrievals anomaly* [4] is another problem which we illustrate. Assume now that there is another shared variable, *checking,* and two other operations:

```
Transfer (real sum) {
        real t1, t2;
```

```
        t1 := Read(checking);
        t2 := Read(saving);

        t1 := t1 - sum;
        t2 := t2 + sum;

        Write(checking, t1);
        Write(saving, t2);
}

PrintTotal () {
        real t1, t2;

        t1 := Read(checking);
        t2 := Read(saving);

        Print(t1 + t2);
}
```

Assume that initially *saving* is 1000 and *checking* is 500. Suppose a call Transfer(200) is interrupted right after *checking* is updated to 300 but before *saving* shows a value of 1200. Meanwhile, a call PrintTotal() is made and runs until completion, outputing a value of 1300.

Again, this is another obvious incorrect result produced by operations on shared

data. In this case, reading the information before all the relevant updates were made results in an inconsistent output.

One obvious solution to the interference problem is to disallow concurrency by allowing no more than one active process within a module. We already rejected this solution on the basis of poor performance.

As another possible solution, one might naively suggest employing any usual mutual exclusion technique to individually protect variables from concurrent access. For example, use semaphores to protect each single state variable from interference by different processes. As the following example shows, this is still an inadequate solution.

By using two binary semaphores, checking-sem and saving-sem, one can rewrite Transfer() and PrintTotal() as follows:

```
Transfer (real sum) {
        real t1, t2;

        Wait(checking-sem);
        t1 := Read(checking);
        t1 := t1 - sum;
        Write(checking, t1);
        Signal(checking-sem);

        Wait(saving-sem);
        t2 := Read(saving);
        t2 := t2 + sum;
```

```
        Write(saving, t2);
        Signal(saving-sem);
}

PrintTotal () {
        real t1, t2;

        Wait(checking-sem);
        t1 := Read(checking);
        Signal(checking-sem);

        Wait(saving-sem);
        t2 := Read(saving);
        Signal(saving-sem);

        Print(t1 + t2);
}
```

If the PrintTotal() and Transfer() operations interleave in a certain way, the problem of inconsistent retrievals shows up again. It is still possible that PrintTotal() computes a total by adding up a new value for *checking* to an old one for *saving*, or vice versa.

One problem with the above attempted solution is that Transfer() signals the checking semaphore too early allowing PrintTotal() to access *checking* before *saving*

is credited with the appropriate amount. The early signal causes the connection between the values of the two accounts to be lost and thus, leads to the unexpected results.

The connection between the two variables can be stated by a *consistency constraint* that the users who share access to those variables expect to hold. Consistency constraints are assertions that the data accessed by users must satisfy [12, 34]. Those assertions are usually only implicitly stated since in real systems, they are too numerous.

For example, the consistency constraint for the above example is that the sum of the two account balances is equal to a specific value. In our discussion, we rejected solutions to the interference problem on the basis that we implicitly assume that this constraint holds. When the constraint is violated we get unexpected and wrong results. This kind of erroneous behavior in the example can be fixed with the proper placement of the semaphore operations, a topic to be discussed later.

To produce correct results, Transfer() must be run in a way that does not leave the variables in a state violating the consistency constraint, i.e., in an *inconsistent* state. By looking at the code of Transfer(), we see that this is accomplished since the sum of *checking* and *saving* is left intact. However, during the execution of Transfer(), there is a time where the state of the variables is temporarily inconsistent. This *temporary inconsistency* is unavoidable while not harmful [12]. The wrong output is actually the result of the early signal operation allowing PrintTotal() access to an inconsistent state which is corrected but only when Transfer() ends.

In summary, we need to require that (completed) operations preserve consistent states while tolerating temporary inconsistency during the execution of the opera-

tions. However, we need to make sure concurrent operations run without ever accessing inconsistent data even if they are temporarily so. In real systems, it is not possible to satisfy those requirements by explicitly listing and checking all consistency constraints. Instead, we run the operations as transactions.

A *transaction* is a sequence of *actions* (operations) manipulating data objects. When executed alone and to completion, each transaction takes a consistent data state to another consistent state [12, 34, 5].

The *concurrency control problem* concerns the coordination (or synchronization) of the actions of different concurrent transactions so as to preserve consistency. Concurrency control prevents the intermediate results of actions within a single transaction from being disclosed and used by other transactions. Such property of transactions is called *concurrency atomicity* or *indivisibility* [4]. For all purposes, transactions can be treated as executing atomically with no interruption from or interleaving with other system events.

A sequence of interleaved actions from a set of concurrent transactions is called a *schedule* [5]. A schedule is *consistent* if it specifies the interleaving of actions in a way that gives each transaction a consistent view of the state of the data [12].

For example, the sequences of operations formed by running the transactions serially in any order are all consistent schedules (since each transaction individually forms a consistent schedule.) Such sequences are called *serial schedules*.

Schedules are used to represent the sequence in which a system executes the actions of a set of transactions. If we impose on the system the requirement that the effect of following a schedule of actions is equivalent to following some serial schedule of the same actions, then consistency is preserved. A schedule which is equivalent

to a serial is said to be *serializable* [12]. The concept of equivalence between two schedules is not treated formally here.

So far in our discussion, we have assumed a failure-free environment and we required just the concurrency atomicity property of transactions. Actually, concurrency control algorithms cannot guarantee consistency in the presence of failures since they may prevent the completion of transactions in execution. Different types of failures may occur. For example, going back to the Transfer() operation, if the system crashes between the updates of the two accounts preventing only one of the balances from being written into, the state of the data is left inconsistent. Also, after a flow violation is detected, execution of a transaction could be halted in the middle of a temporarily inconsistent state, a situation which should be prevented from becoming permanent.

Some *error recovery* technique must be employed to render the system and concurrency control resilient in the presence of failures. An all-or-nothing policy should be enforced for transaction execution: either all the the effects of the actions of a transaction are observed or none of them are, in either case consistency is preserved. This is the *failure atomicity* or *recoverability* requirement [5].

Error recovery techniques are concerned with bringing the system to a consistent state after a failure causes the interruption of the execution of a transaction, possibly leaving inconsistent intermediate results. As one option, they can roll back to the consistent state existing previous to the start of the transaction, i.e., employing *backward recovery* [19]. Alternatively, *forward error recovery* takes the system to a consistent state by modifying and correcting the intermediate inconsistent state resulting from the incomplete transaction [19]; it makes use of the partial results

already computed. Forward recovery requires exact knowledge about the damage the failure caused and the semantics of the interrupted transactions. Exception handling is an example technique that recovers forward from software failures.

Systems can be designed to tolerate certain failures. Failures are divided into two categories: *tolerable failures* and *intolerable failures* [19]. In short, tolerable failures are ones that the recovery system is able to handle. All other failures are intolerable. We assume that intolerable failures do not occur or that otherwise, consistency is not guaranteed.

Combining concurrency atomicity with failure atomicity of transactions allows the building of systems that maintain the consistency of their data in the presence of concurrency and (tolerable) failures. Actually, by preventing intermediate results of transactions from being viewed by others, concurrency control halts the propagation of results of failed transaction, and thus simplifies the recovery task. Transactions that are both concurrency and failure atomic are also called *atomic actions.*

Next, we present some well-known concurrency control algorithms and error recovery techniques. There is an impressive amount of published material on both subjects. We present only the basic works that directly or indirectly influenced our thinking and results.

3.2 Recovery Techniques

Recovery techniques guarantee the failure atomicity property of transactions. Given a started transaction, either all its effects are completely reflected on the data, in which case the transaction is said to *commit,* or all of them are erased, in which case the transaction is said to *abort.*

Implementing transactions requires the availability of *permanent* or *stable* storage media that can survive crashes. Permanent storage provides the usual read and write operations. The write operation is always performed atomically so as not to leave partially updated and corrupted data. Disks can be used to implement permanent storage [19]. We only assume that some kind of stable storage exists and that it is an intolerable failure for it to behave erroneously or lose its data.

Permanent storage always contains a consistent state of the objects. However, intermediate states can be stored on volatile storage while being manipulated by an active transaction. There are basically two general techniques to make use of permanent storage for recovery: shadowing and logging.

We also mention a protocol that allows transactions to commit atomically on distributed sites.

3.2.1 Shadowing

This technique involves maintaining consistent versions of object states on permanent storage. These versions represent the most recent data that was written by committed transactions. Any intermediate updates are made to different temporary versions of the states; the previous consistent versions are not affected.

When a transaction is aborted, the temporary inconsistent states are discarded. When a transaction is committed, its effects are made permanent by atomically replacing the previous consistent state with the new versions.

3.2.2 Logging

Instead of creating new versions of the data, a logging technique makes all updates to the original consistent copy. However, an *incremental log* of all updates to the states is kept. Each log entry represents an update operation and contains enough information to allow for the recovery of the previous state. The log is maintained on permanent storage.

When a transaction aborts, the recovery is made by going through the log and performing an *undo* operation for each update entry. Eventually, the original consistent state is recovered. To commit a transaction, the log is simply discarded since the changes to the data are already made.

3.2.3 Distributed atomic commit

A distributed transaction may manipulate data at different nodes called its *cohorts*. Once a decision has been made to abort or commit a transaction, the recovery system must guarantee that all cohorts will take the same coordinated actions: either discarding all computations made by the transaction or committing them. The *two-phase commit* protocol coordinates the commit action among the cohorts [19]. The protocol works even if crashes happen while it is being executed. It is implemented by a *commit coordinator* that resides on a single site and can communicate with all the cohorts.

3.3 Concurrency Control

3.3.1 Two-phase locking

Locking is a way to synchronize concurrent access to shared resources. Before a transaction is allowed to access a data item, it must obtain a lock on the item. Commonly, a transaction can request either read access or write access on a data item. Different transactions can concurrently read an item but only one transaction should be allowed to write to an item. Accordingly, a common lock convention divides locks on items into two types: read locks and write locks. Before reading (writing) an item, a transaction must request a read lock (write lock). Only if it obtains the requested lock that a transaction is allowed to access the item.

We denote the operations of reading and writing a data item x by r[x] and w[x], respectively. We then can assume that with each data item x, two locks are associated: a read lock and a write lock, denoted by rl[x] and wl[x], respectively. Write locks are exclusive; if a transaction requests and obtains wl[x] for an item x, then no other transaction can obtain or hold a lock (of any type) on x. On the other hand, read locks can be shared by reader transactions; any number of transactions can hold rl[x] (as long as no transaction is holding wl[x].)

The above lock properties ensure the one-writer, multiple-reader rules. Lock types that are not allowed to be concurrently held on the same item by different transactions are said to *conflict*; otherwise, they are said to be *compatible*. A transaction that requests a lock that conflicts with a lock held by another transaction is blocked until the conflicting lock is released (i.e., until the item is unlocked.)

A *compatibility table* can be used to summarize lock properties. In Figure 3.1,

	rl	wl
rl	yes	no
wl	no	no

Figure 3.1: Compatibility table for read and write locks

we show the compatibility table for read and write locks. A "yes" entry indicates that the lock types do not conflict or that they are compatible.

An important property that needs to be assumed for any locking scheme is that all basic operations are implemented atomically. Namely, it is assumed that the operations of acquiring and releasing locks and of reading and writing a data item are all atomic. The atomicity of the basic operations can be implemented using any known mutual exclusion technique such as semaphores.

A *two-phase locking (2PL)* transaction is one that obtains all the locks it needs before it starts releasing any of them. Such transactions are divided into two phases: a growing phase in which locks are requested but none are released and a shrinking phase in which locks are released but none are requested. If only 2PL transactions are allowed to run in a system, then any resulting schedule is serializable and thus, maintains consistency.

A *strict* 2PL mechanism requires transactions to release all the locks together when the transaction ends. One obvious advantage of such transactions is that there is no need to perform any steps to figure out when the shrinking phase should start; it always starts when the transaction is done.

3.3.1.1 Adding new locks We have assumed that the only operations that can be performed on a data item are reads and writes. However, new types of

operations and corresponding lock types can be added if one follows the following rules [5].

1. Each new operation is implemented to ensure its atomicity.

2. A new lock type is defined for *each* new operation.

3. A compatibility table is defined that includes all locks.

The above rules will be used later when we present our locking scheme for our system.

3.4 Moss' Nested Transactions

So far, we have talked about only single-level atomic transactions. Moss [30, 31] developed a nested transaction model in which any transaction can spawn subtransactions. The model is based on locking. Subtransactions run as part of a parent transaction but keep their atomicity properties on their own. They introduce concurrency within a transaction as well as independent recovery properties.

Any transaction can have child subtransactions which in turn, can have their own child subtransactions, and so on. The top-level transaction and its subtransactions form a rooted transaction tree. In this context, we can talk of transactions or subtransactions being children, parents, ancestors, descendants, siblings....

Subtransactions are allowed to run concurrently but not with their parents. They either abort or commit. If a subtransaction aborts, its parent need not also abort. However, even if a subtransaction commits, it can still be aborted if its parent aborts. In fact, the effects of a multilevel transaction are not made permanent and are hidden from other transactions until the top-level (the root) transaction commits. On the

other hand, the effects of a committed nested subtransaction are visible to other related subtransactions.

The locking rules for nested transactions are an extension of the usual single-level locking protocol. They are simplified by the fact that parent transactions are not allowed to run concurrently with their children. Any (sub)transaction is assumed to be 2PL. The rules are stated as follows.

1. A transaction may read (write) an item only if it holds the corresponding read (write) lock on the item.

2. A transaction may obtain a read lock on an item only if all holders of write locks (if any) on the item are its ancestors.

3. A transaction may obtain a write lock on an item only if all holders of any (read or write) locks (if any) on the item are its ancestors.

4. If a subtransaction commits, its locks are inherited by the parent. The parent becomes the holder of those locks.

5. If a subtransaction aborts, its locks are discarded.

These rules ensure serializability within as well as among transactions.

The rules for recovery are also extensions of the shadow versions technique. They are stated as follows.

1. When a transaction obtains a write lock on an item, a version of the item is made. All updates to the item are made to this shadow version.

2. When a transaction commits, the shadow version is inherited by its parent. If the parent already has its own version of the same item, the child's version takes

precedence. When the top-level transaction commits, the shadow versions it holds are installed on stable storage.

3. When a transaction fails, its shadow versions are discarded.

In the first rule, the shadow version is made from the original stable storage copy if no other versions of the same item are held within the transaction tree. Otherwise, the shadow version is copied from the version held by the youngest (most deeply nested) transaction.

Committing a top-level transaction involves communicating with all the sites where a subtransaction run. The committing process must be made atomic. A two-phase commit protocol is used. The top-level transaction acts as coordinator of the protocol. Its committed subtransactions are the participants.

3.4.1 Implementing remote procedure calls

One of the interesting applications [26, 24] of the nested transaction model is in implementing remote procedure calls with at-most-once semantics in a distributed system[1]. We heavily rely on the following scheme in our model.

The main idea is to run the RPC at the remote node as a subtransaction of the caller. Assume that transaction T at node A issues an RPC to run on node B.

1. The caller T creates a local child subtransaction T_1 at node A which is to manage the call on behalf of T. T_1 is called the *local call subtransaction* of T.

2. In turn, T_1 creates a remote child subtransaction T_2, a grandchild of T. T_2

[1] RPCs with at at-most-once semantics are guaranteed to either run exactly once or not at all.

runs on B and executes the RPC; it is called the *remote call subtransaction* of T.

An advantage of having the extra local call subtransaction is to be able to abort the remote RPC subtransaction without affecting the caller transaction. To abort T_2, all is needed is to locally abort T_1. Then, T can resume without having to worry about T_2 since by aborting T_1, an eventual abort of T_2 is certain. Another advantage of having T_1 is to allow the caller to concurrently start a number of RPCs.

Of course, if the RPC subtransaction commits, it must inform its parent, the call subtransaction. In this case, the caller is informed of a successful call. If for any reason the call is unsuccessful, the caller can always try another RPC or retry the same one; it does not have to abort.

4. A MODEL FOR MULTILEVEL SECURITY FOR DISTRIBUTED SYSTEMS

The information flow control mechanisms presented in Chapter 2 are designed for centralized systems and are not readily suited for a distributed environment. Also, each approach handles either dynamic or static class binding but not both.

On one hand, the run-time mechanisms require special hardware and incur substantial overhead at run time. On the other hand, the compile-time mechanisms require that any service procedures be verified before they can be called. This is not practical in a distributed system where objects and their exported procedures can exist on different sites. For our purposes, we need a mechanism in which the verification of the software programs can proceed as much as possible locally, independent of the software on other sites until remote services are requested. This supports the modular approach to software development.

For efficiency reasons, the security mechanism should rely mostly on compile-time verification. However, in our environment, it is not possible to avoid run-time checking of flows altogether because of the existence of messages passed among the sites.

Finally, supporting both static binding and dynamic binding is a need and allows more flexibility. For example, dynamic binding eliminates the need for more than one

version of a single procedures with differently classified parameters. On the other hand, allowing static binding becomes necessary to classify the users' output devices. For example, a terminal may be treated as statically-bound to a class corresponding to the user's clearance, as long as the login shell is running.

In this chapter, we present a mechanism that includes the above features for distributed object-oriented systems. First, we introduce a computational model that we will assume henceforth. In the last part of the chapter, we discuss some major restrictions of the security mechanism. The restrictions will be removed in later chapters.

4.1 The Resource Module Model

In this section, we introduce a programming model which we will call the *Resource Module* (RM) model. It basically serves as a computational and syntactical basis for the rest of this dissertation. However, we note that the RM model is general enough to allow any of the subsequently suggested mechanisms to apply to any other distributed model.

In the RM model, it is assumed that data types and their operations are defined by *RM classes* that can be instantiated to produce *RM objects* or simply *RMs*. An RM is a class instance that encapsulates some data objects, its *state variables*. The *state* of an RM consists of the current data stored in its state variables. Access to the state variables of an RM is allowed only through the exported operations defined in the class of that RM (including the inherited operations).

There are two types of state variables. A *dynamic state variable* is one whose security class is dynamically-bound. A *static state variable* has a constant statically-

bound class. Similarly, we define two types of local variables in procedures: *dynamic local variables* and *static local variables.* Each state variable has two pieces of data associated with it: its value and its security class. Accordingly, we can talk about the *value state* and the *security state* of an RM object.

In Figure 4.1, we show an example of how RMs are represented. The example shows an RM encapsulating four state variables. Dsv1 and dsv2 are dynamically-bound. Ssv1 and ssv2 are statically-bound to security classes SECRET and LOW, respectively.

The operation init is a special procedure that runs when the object is created. The exported procedure P has three parameters. We use the symbol '|' to separate IN parameters which are passed by value from OUT parameters which are by-result.

Communication of service requests between RMs is made through blocking *remote procedure calls* (RPCs). An RPC, like a usual procedure call, may include IN and OUT actual parameters. In the example, P sends an RPC request to RM M2 for procedure Q. When an object receives an RPC it creates a process to run the corresponding procedure code. Internal object concurrency is allowed. So, different RPCs that are received at an RM result in the creation of concurrent processes and multiple threads of control within the RM.

In addition to RPCs, two other types of messages are used for RM communication: *return* messages which carry results back to the caller of a procedure and *probe* messages which will be explained in the next section. A return message is sent when a procedure ends and causes the termination of the process sending it. A probe, like an RPC, causes the suspension of the process sending it.

```
M1 = {
        int dsv1, ssv1(SECRET), dsv2;  ! STATE VARIABLES
        real ssv2(LOW);

        init()                  ! INITIALIZATION PROCEDURE
        {
                dsv1 = 0;
                dsv2 = 0;
        }

        P(a,b | c)              ! EXPORTED PROCEDURE
        int a, b(CONFIDENTIAL);
        real c;
        {       int index(LOW), dlv;

                ...
                M2.Q(dlv | ssv1);
                ...
        }

        R(a | b)              ! EXPORTED PROCEDURE
        int a, b;
        {
                ...
        }
}
```

Figure 4.1: An example representation of an RM object

4.2 The Information Flow Control Mechanism

4.2.1 Overview

The information flow control mechanism is developed in [27]. It is a combined compile-time run-time mechanism with the following features.

1. At compile time, the internals of individual procedures are certified independently. Also, flow and security information that is needed at run time is saved in appropriate structures.

2. At run time, certification of the communications between modules is performed. For improved efficiency, run-time certification is invoked only at message passing time.

3. The mechanism allows procedures to have statically-bound and dynamically-bound local variables and formal parameters.

4. The mechanism supports statically-bound state variables.

5. The mechanism can support dynamically-bound variables if concurrent processes within a module are not allowed.

The main task of the compile-time mechanism is to build an *information flow template* for each exported procedure in a class. Templates are instantiated when an RPC is received. A template consists of two main parts: *security definitions* and *security checks.*

The security definitions are used by the run-time mechanism to compute the security classes of various dynamically-bound variables. For example, the class of

a dynamically-bound state variable must be computed at run time using a security definition. The class of this state variable cannot be known at compile time since it may depend on, among other things, the class of some dynamically-bound IN parameters.

The security checks of a template contain all the information needed at run time or message passing time to verify the security of the possible flows that may be caused by an invocation of the procedure. Specifically, for each statically-bound variable that is visible and accessed inside the procedure code, there is a security check to express the flow into that variable. For instance, before starting the execution of a procedure, the security checks are evaluated to determine the possibility of security violations. This may require evaluating the classes of the dynamically bound variables that are involved in the flow.

The problem of inter-module implicit flows is handled nicely by including with each RPC message the security class on which the call is conditioned. That security class is computed at run time using a security definition in the template instance. However, since, among other reasons, our model allows the presence of dynamically-bound variables, implicit flows need to be checked even if the RPC is skipped as a result of a conditional. (This is similar to the update operation in Denning's run-time approach.) This situation is handled by sending a *probe* message to the called module. A probe contains the class on which the call is conditioned to allow the called module to verify the security of the implicit flows. It results in the update of the classes of dynamic state variables.

In this section, we present the details of the information flow run-time and compile-time mechanisms for distributed systems. The mechanisms are presented

with enhancements, clarifications, and notational modifications from the original mechanisms found in [27]. The enhancements remove the original restrictions of not allowing internal concurrency and of overclassification of the classes of dynamic variables. Further elaborations on the implication of these enhancements will be presented later in this section.

Next, we discuss implicit flows at length. This will help in clarifying the subsequent presentation of the run-time and compile-time mechanisms.

4.2.2 Discussion on checking implicit flows

The information flow control mechanisms that have been presented in Chapter 2 follow either of two approaches: the run-time approach where checks are exclusively made at execution time and the compile-time approach where checks are made when programs are compiled. Our approach is a combined compile-time run-time[1] approach. We chose that approach because we wanted to have as much of the flow checking done at compile-time for efficiency. However, because of the distributed nature of our system we could not completely avoid run-time checking.

In a mechanism following a compile-time approach, flow violation error messages do not convey any information about the values of any variable to a user (subject) since offending programs are rejected even before they are run or any values are accessed. However, a compile-time approach mechanism may reject programs that if run would only cause authorized flows and would not lead to a flow violation. We say that such a mechanism is not *precise* [9]. The imprecision comes from the fact that

[1] More specifically, the run-time checks in our approach are made at message passing time.

the actual execution path of the program is not taken into consideration. We present an example using Denning's compile-time certification mechanism for programs with static variables.

```
int x(SECRET), y(LOW);
...

P()
{   ...
    if (x == 0) then y = 0;
    ...
}
```

The above code will not be certified at compile time since it shows an implicit flow from x to y which is illegal because it is a flow from SECRET to LOW. Regardless of what the value of x would have been at run time, the program is rejected with a flow violation error message which does not leak any classified information.

Now, we consider the same example with a run-time approach. Specifically, we consider Fenton's run-time mechanism with static variables. Fenton showed that such a mechanism, can be made more precise than the compile-time approach if at run time, the actual execution path of the program is considered. He showed that, in the absence of dynamic variables, implicit flows need only be checked at explicit assignment time. In other words, implicit flows into a static variable are not checked if the assignment statement to that variable is skipped. For example, the security of the above program is checked as follows.

```
int x(SECRET), y(LOW);
...

P()
{   ...
    if (x == 0) then
       ! security check done at run time
       IF (class(x) <= class(y)) THEN
            y = 0;
       ELSE FLOW VIOLATION IS DETECTED;
    ...
}
```

Note how the extra code (if-statement shown in capital letters) for checking the implicit flow from x to y is only executed right before executing the assignment statement and that it is not executed if x is not 0 (i.e., if the assignment statement y = 0 is skipped). This results in a more precise mechanism since if x is not 0, no checking is done and the program is not rejected as in the compile-time approach mechanism.

However, with the increased precision, another problem is introduced. Fenton discovered that in this mechanism if a flow violation is reported then the error message can actually leak classified information to the subject. In the above example, if x is 0, then the mechanism detects a flow violation since $\underline{x}$ is strictly greater than $\underline{y}$. If the user is informed about the flow violation then he can deduce the value of x since the security check is only made if x is 0. Moreover, if x is not 0, the absence of a flow

error message reveals the class of x.

To solve the problem, in case a flow violation is detected, Fenton's mechanism requires that it is not reported and even, that the offending program is allowed to continue running so that no leakage to the user takes place. For example, the above program is compiled into the following code.

```
int x(SECRET), y(LOW);
...

P()
{   int x(SECRET), y(LOW);

    ...
    if (x == 0) then
       IF (class(x) <= class(y)) THEN
            y = 0;
       ELSE SKIP; ! skip if flow error is detected
    ELSE SKIP; ! also skip if x is not 0
    ...
}
```

In this case, if x is 0 and a flow violation is detected before an assignment statement is executed, then the program is not allowed to perform the offending statement. However, the program continues running by executing the SKIP statement. On the other hand, if x is not 0 and no flow violation is detected, SKIP is also executed. This way if the user subsequently tests the value of y and finds it not to be 0, then

he cannot deduce whether that is because x is not 0 or because a flow violation is detected. On the other hand, if y is found to be 0, then the user may deduce that x is 0. However, this is not a problem since in that case the user must have enough clearance to know the value of y and therefore must have enough clearance to know the value of x since $\underline{x} \leq \underline{y}$.

Note that the code indeed violates the lattice no matter what the value of x is. Once the test `if x == 0` is reached and evaluated, the implicit flow to y automatically occurs. However, the security check is only made when x is 0. The question arises on whether this program should be allowed to run at all despite the presence of the lattice violating flow. Fenton argues that since the user is left in the dark about the way the program runs then no actual harmful flow occurs. However, the program is an erroneous one and its practicality is very doubtful.

As a matter of fact, the major drawback of Fenton's method is that users are never sure of the results of their programs and whether they can rely on them. Reliable programming becomes impossible in this case. For that reason, we reject this type of solution for the run-time part of our mechanism. We do need to inform users of any flow violations to have any practical and meaningful programming environment. Of course, this should not jeopardies security.

In [27], Mizuno presents a preferable solution for the problem of the leakage of information through flow violation error messages. The idea is not to allow the execution of a conditional statement unless it is certified that the user is cleared to read the implicit flow caused by the statement. In other words, the user's clearance, represented by SUBJECT, is greater or equal to the class of the implicit flow. If it is not the case then a flow violation can be reported without leaking information about

the implicit flow since the conditional statement does not even get to execute and no implicit flow takes place. We present an example showing how the security of the program is checked[2].

```
int x(SECRET), y(LOW);
...

P()
{
    ...
    ! check clearance of the subject first
    IF (class(x) <= clearance(SUBJECT)) THEN
    {  if (x == 0) then
          IF (class(x) <= class(y)) THEN
             y = 0;
          ELSE REPORT FLOW VIOLATION AND ABORT;
    }
    ELSE REPORT NOT ENOUGH CLEARANCE AND ABORT;
    ...
}
```

The idea behind introducing the clearance of the user, SUBJECT, is that any

[2]The example is only used to explain the mechanism since actually, the above program will be rejected first by our compile-time mechanism. The run-time mechanism is not called upon since no messages are passed. However, for simplicity, let us assume for now that only a run-time approach is employed. Later, we will generalize the example to show RPCs being made causing the run-time mechanism to execute at message passing time.

output device to which a user has access must be statically classified at his clearance level to prevent any higher classified information from being written onto that device. For example, a user's terminal can be classified at the user's clearance at login time. The terminal classification is static and is not allowed to vary for the duration of the login session.

So, if we think of the flow error message as actually carrying an implicit flow from x, then we can 'classify' the error message at the level of x (SECRET), even before testing x. Then, the error message must not be allowed to flow to the user's output device unless he is cleared to read x itself. If the user is not cleared for x then he should not be able to even test its value and he should be informed that he has lower clearance than needed to run the program.

By going through all possible execution paths in the above code, we show that no illegal flow of information can occur.

Case 1: Assume SUBJECT is strictly less than x. Then, a potential violation is detected and the user is informed that he does not have enough clearance to run the program. No information about x is compromised and the implicit flow from x to y is prevented since the test `if x == 0` is not performed.

Case 2: Assume x is less than or equal to SUBJECT.

1. If x is 0 then a flow violation is detected when the classes of the variables are compared. The user receives a flow violation error message. This message actually carries information about x which is SECRET. So, the user is able to infer that the value of x is 0. This does not constitute an illegal leakage since it has already been established that the user is cleared

to read x.

2. If x is not 0 then no flow violation is reported and the program may continue executing. Note however that, as in Fenton's mechanism, since the test on x was actually performed, here also the implicit flow to y has occurred but again ignored. Later, we will show how this is a problem which raises questions about the practicality of allowing the program to continue running in such cases.

Note that in case 2, even a normal termination message can carry information about x. For example, assume another user has a clearance, SUBJECT2, lower than x, and he knows SUBJECT, x, and y. In other words, the user knows the results of the security tests. Then, if he has access to the normal termination message, then he can deduce that the value of x is not 0. This is similar to this other user being able to deduce that x is 0 if he gets access to the flow violation message. So, it must be assumed that no other user with a lower clearance may have access to the output device of the user on whose behalf the program is running. In this way no message of any type can cause undetected illegal flows. The lower clearance user is prevented from having access to the program's termination status.

Although this scheme in no case produces insecure programs, but as mentioned above, this mechanism also violates the lattice structure by ignoring the implicit flow from x to y in case x is not 0. This may lead to unexpected results as shown by the following code which is a slight modification of our running example.

```
int x(SECRET), y(LOW);
int z(LOW);
```

```
...

P()
{
    ...
    ! check clearance of the subject first
    IF (class(x) <= clearance(SUBJECT)) THEN
    {  if (x == 0) then
          IF (class(x) <= class(y)) THEN
             y = 0;
          ELSE REPORT FLOW VIOLATION AND ABORT
    }
    ELSE REPORT NOT ENOUGH CLEARANCE AND ABORT

    IF (class(y) <= class(z)) THEN
       z = y;
    ...
}
```

In the example, we introduced another global variable, z, of class LOW. If x is not 0 and there are no flow violations (i.e., $\underline{x} \leq \underline{\text{SUBJECT}}$ and $\underline{x} \leq \underline{y}$), then the variable z gets the value of y. Assigning the value of y to z is not detected as a violation since they are both LOW. However, y carries a SECRET implicit flow since the test `x == 0` was indeed reached and executed. It should be expected that the assignment does not take place.

Although, we have shown that no leakage of information can result since the termination status of the program is protected, but here we claim that the program is erroneous since it actually allows an illegal flow from x to y and then to z. For that reason, we conclude that the implicit flow must be checked even if the assignment does not take place. The security checkings should be performed as shown below.

```
int x(SECRET), y(LOW);
int z(LOW);
...

P()
{
    ...
    IF (class(x) <= clearance(SUBJECT)) THEN
       if (x == 0) then
          IF (class(x) <= class(y)) THEN
             y = 0;
          ELSE REPORT FLOW VIOLATION AND ABORT
    ! check flow even when skipping the assignment
       ELSE IF NOT (class(x) <= class(y)) THEN
                REPORT FLOW VIOLATION AND ABORT
    ELSE REPORT NOT ENOUGH CLEARANCE AND ABORT

    IF (class(y) <= class(z)) THEN
       z = y;
```

```
    ...
}
```

The implicit flow is checked under both branches of the if statement. The flow violation will be caught regardless of the value of x and z will not receive the value of y, as the user would expect. So, the above scheme is not more precise than Denning's compile-time approach but a degree of preciseness has to be sacrificed if we want to allow a practical programming environment where users can rely on the results from their programs.

Before we continue with the discussion, we need to show, in our system, the equivalent code that will actually call upon the run-time flow checking mechanism. In Figure 4.2, we show two modules where x, y, and z are static state variables. There are no local assignment statements in P, but when it is called the implicit flows to M2.W are checked regardless of the value of x. This is done by having the RPC carry the class of x into M2.W. If the RPC is skipped, sending a probe carrying the same class allows the checking of the implicit flow that occurs even though the assignment in M2.W is skipped. In all cases, z is prevented from receiving the value of y from the other module since the flow from x to y is not allowed.

As previously mentioned, probes must update the classes of dynamically-bound state variables, in the absence of the actual RPC. As justified by the above example, probes need to be sent even in case where state variables are statically bound. Probes do need to perform checks of implicit flows into static variables, in addition to their task of class updating.

Since we concluded that implicit flows must be checked in both branches of a conditional, one might question the need for still checking the clearance of the

```
M1 = { int x(SECRET);
       int z(LOW);
       ...

! no assignment statement exists
! implicit flows are checked at
! the procedures that are called
 P()
 {
 ...
 IF class(x) <= clearance(SUBJECT)
 THEN
    if (x == 0) then
         M2.W()
    ELSE
     ! check implicit flow
     ! in skipped call
      send-probe(M2.W);
 ELSE
   REPORT NOT ENOUGH CLEARANCE
   AND ABORT;

 M2.R(z);
 ...
 }
}
```

```
M2 = { int y(LOW);
       ...

 ! either the probe or the
 ! RPC carry in implicit
 ! flows (e.g., class(x));
 ! in either case it must
 ! be checked
 W()
 {
  IF "implicit" <= class(y)
  THEN
     y = 0;
  ELSE
     REPORT FLOW VIOLATION
     AND ABORT;
 }

 ! b is dynamic;
 ! at run time it gets the
 ! class of y (LOW)
 R( |b)
 {
  b = y;
 }
}
```

Figure 4.2: Implicit flow checking across modules

```
M1 = { int x(SECRET);
       int z(SECRET);
       ...

   ! x may flow to subject
   ! if clearance(subject)
   ! is not tested
   P()
   {
       ...
       if (x == 0) then
            M2.R( |z)
       ELSE
          send-probe(M2.R);
       ...
   }
}
```

```
M2 = { int y(TOPSECRET);
       ...

   ! b gets class of y
   R( |b)
   {
      b = y;
   }
}
```

Figure 4.3: Violation in the absence of checking SUBJECT

subject. In Figure 4.3, we show an example in which not making that check results in an illegal flow.

In the example, assume that the subject knows x, y, and z. So, he knows that that if x is 0, a flow violation occurs when a return is sent back from M2.R and the flow from the formal OUT parameter b to z is checked [3]. However, if x is not 0, then since a value is not returned from M2.R through z, a flow violation is not reported and the value of x can be deduced. As a consequence, if the subject is not cleared to read y, security is violated unless we prevent the execution from reaching the test x == 0. We conclude that checking the subject's clearance is still necessary.

[3] As will be explained later in the description of the run-time mechanism, if M2.R is called, when it returns, it also sends back the class of its OUT parameter, b. In this case, b is TOPSECRET and it flows to the actual OUT parameter, z, which is classified SECRET.

As the last topic in this section, we discuss the placement of the subject's clearance checking. Obviously, the earliest it could be performed is right when an RPC is received at run time since SUBJECT is not known before then. This may result in wrongly rejecting secure programs, as the following code shows.

```
! perform all subject clearance checks
! before execution starts
IF NOT (class(x) <= class(SUBJECT)) THEN
        REPORT NOT ENOUGH CLEARANCE
IF NOT (class(y) <= class(SUBJECT)) THEN
        REPORT NOT ENOUGH CLEARANCE


...
if (x == 0) then
    if (y == 0) then
         z = 0;
...
```

If x is not 0, there is actually no need to check the subject's clearance against y since the test `y == 0` is not reached.

As a consequence, to make the mechanism more precise, we can move the test until right before a conditional is executed. That is the latest it can be performed since in no way can we afford to have the test after the condition is evaluated. Then, the code is transformed as follows.

```
! delay subject clearance checks as long as possible
```

```
...
IF (class(x) <= class(SUBJECT)) THEN
{   if (x == 0) then
        IF (class(y) <= class(SUBJECT)) THEN
        {  if (y == 0) then
               z = 0;
        }
        ELSE REPORT NOT ENOUGH CLEARANCE
}
ELSE REPORT NOT ENOUGH CLEARANCE
...
```

However, it must be noted that the gain in precision is made at the expense of violating our requirement that run-time checks are only made at message passing time. However, we don't view this as a major deviation from our philosophy since it is a relatively simple test to perform during the execution of a procedure and only when conditions are checked. Moreover, the gain in preciseness may be substantial.

Finally, we summarize our conclusions.

- The subject's clearance must be checked against the implicit flows to avoid leaking information through flow violation error message. However, if more precision is desired, the system designer may elect to delay the check until right before the conditional is executed.

- The user's output devices are classified at his clearance level at login time. No other user with a lower level may read information shown on those devices. This,

in turn, has the result of protecting the information regarding the termination status of a program (flow error and normal termination errors) by 'classifying' them at the level of the clearance of the subject running the program.

- For users to expect reliable results from their programs, the system must check implicit flows even if assignments are skipped. This applies even if only statically-bound variables are used. Therefore, in addition to updating the classes of dynamically-bound state variables, probes must be given the task of checking implicit flows into statically-bound ones.

Now, we are ready to present in detail both the compile-time and the run-time mechanisms.

4.2.3 The compile-time mechanism

The compile-time mechanism checks the internal security of procedures. It also builds information flow templates, structures that include information necessary for the run-time certification mechanism.

4.2.3.1 Information flow templates Information can flow in and out of procedures through different types of variables. Variables that carry information into a procedure are called *input variables,* while *output variables* are the ones that carry information out.

For a procedure P in a module M (see Figure 4.4), input variables are identified as being the actual IN parameters sent to P by its callers, the state variables in M whose values are read in P, and the formal OUT parameters of procedures called by P. (Constants used by P are always assumed to have LOW classification, so they are

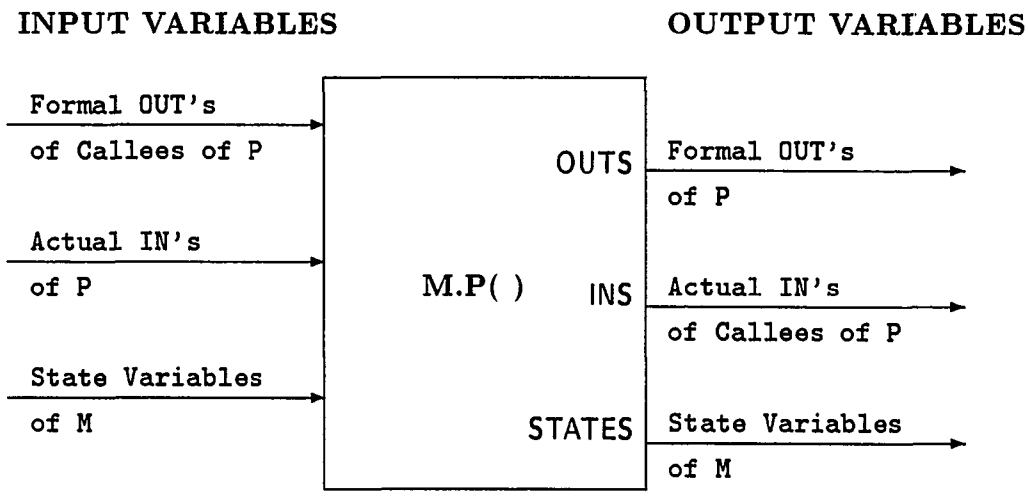

Figure 4.4: Input and output variables of a procedure

not considered.) Output variables are the formal OUT parameters of P, the state variables in M that are written in P, and actual IN parameters sent to procedures called by P.

Since, in general, the classes of input variables are not known until run time, the compile-time mechanism represents them with *security variables* or *s-variables*, for short. The classes of static variables are represented by their constant classes. The following example illustrates the notation that is used for the different s-variables.

Assume a procedure P in a module M1 is declared with the heading P(x | y) (the symbol '|' separates IN parameters from OUT parameters). Also, assume that P makes a call M2.Q(a | b) for a procedure Q in M2. Then, P.x denotes the s-variable representing the class of the input variable corresponding to the actual parameter for x. M2.Q.b denotes the s-variable for the class of the value returned to P by the RPC to Q. Finally, the class of a dynamically-bound state variable dsv in M is denoted as

dsv.

Note that P.x and M2.Q.b are used to represent the actual class of the information carried into the procedure through the corresponding input variables which are considered to be the recipients of that information flow. The variables themselves, x and b, may actually be statically bound. In that case, we will see later that the flows from P.x to x and from M2.Q.b to b must be checked. Otherwise, if x and b are dynamically-bound, then they are assigned the classes P.x and M2.Q.b, respectively.

In order, to compute the classes of output variables in terms of the s-variables of the input variables, *security expressions* or *s-expressions* are generated. S-expressions represent the least upper bound of some relevant security classes and s-variables. For example,

$$\oplus(\text{SECRET, TOPSECRET, dsv, z, a})$$

is an s-expression that represents the least upper bound of the classes within the parentheses, where dsv, z, and a represent s-variables. Note that the example expression can be simplified and written as

$$\oplus(\text{TOPSECRET, dsv, z, a})$$

since the least upper bound of the constant classes can be computed at compile time and since it is obvious (from the $\oplus$-operator) that the operands represent security classes. We will usually show only the simplified form of a s-expression.

The compile-time mechanism uses s-expressions to build *information flow templates* for each procedure in a module (including the initialization procedure.) The flow templates are formed of two parts. The first part is used to compute (an upper

bound on) the security classes of output variables at run time and contains *security definitions* or *s-definitions*. S-definitions are of the form

$$\underline{\langle \textit{ output variable } \rangle} = \langle \textit{ s-expression } \rangle.$$

The flow template contains an s-definition for each of the output variables. An s-definition corresponding to a static variable is simply its security class and will not be shown in templates.

As an example, consider the following piece of code.

```
int dsv;
...

P( a | b);
int a, b;
{  int c(SECRET);

   ...
   if a then
     b = c;
   else
     b = dsv + 1;
}
```

Then, an s-definition for the output variable b can be written as

$$\underline{b} \equiv \oplus(\underline{dsv}, \underline{P.a}, \text{SECRET}, \text{LOW})$$

or simply,

$$\underline{b} \equiv \oplus(\text{SECRET, dsv, P.a}).$$

The dynamically-bound output variable b receives flows from a, c (SECRET), dsv and 1 (LOW). The s-definition shows an upper bound on the class of b when the code is executed. Since the values of some shown s-variables, P.a and dsv, cannot be known until run time when P gets called, $\underline{b}$ cannot be computed before then.

Since there are three different types of output variables, the s-definitions in a template can be further divided into three categories. We refer to those categories as OUTS, INS, and STATES s-definitions corresponding to the formal OUT parameters, actual IN parameters and state variables, respectively. The above example s-definition would be listed under the OUTS category of a template for P.

The second part of an information flow template is used to verify the security of the flows into the static variables that are written into by a procedure; it contains *security checks* or *s-checks*. For each static variable that is written into by a procedure, the compile-time mechanism creates an s-check of the following form

$$\underline{\langle\ \textit{static variable}\ \rangle}\ (\langle\ \textit{constant security class}\ \rangle) \leftarrow \langle\ \textit{s-expression}\ \rangle.$$

The s-checks show the flow of information into the static variables that may occur during the execution of the procedure.

For example, from the following piece of code

```
int sv(SECRET);
...
```

```
if a then
   M.P( x | sv);
else
   sv = 0;
```

we can generate the following s-check for sv

$$\underline{sv}(\text{SECRET}) \leftarrow \oplus(\text{LOW, a, M.P.sv}).$$

The s-check is evaluated at run time to verify the flows into sv. Note how the example shows how sv and M.P.sv represent different classes. The former is the static class SECRET. The latter is the class of the information returned by the call to M.P.

4.2.3.2 Generating s-expressions for loops To generate s-expressions for s-checks or s-definitions for variables that receive flow in a loop, we need some new special notations. Here, we will present the way loops are handled in [27]. In Chapter 6, we present our new scheme to generate s-definitions. However, we note that s-checks will still be generated in the way presented here.

Consider the following piece of code containing a while loop.

```
a = 0;
while (exp) do {
   a = b;
   b = c;
}
```

At execution time, during the first iteration of the loop, b flows into a. During the second iteration, c also flows into a since at the end of the previous iteration c flows into b.

When templates are being generated at run-time, it is not known how many times the loop will iterate. The s-expressions must be generated so as to allow for all possible flows. In [27], a flag (*) is used with certain s-variables in s-expressions to indicate that the flow from that variable occur within loops.

For example, in the above code, if a is a SECRET static variable, then the following s-check is generated.

$$\underline{a}(\text{SECRET}) \leftarrow \oplus(\text{LOW}, b(*))$$

The flag associated with b indicates that b flows into a in a loop, that is, more than once.

Assume that during execution time, the initial classes of b and c are SECRET and CONFIDENTIAL, respectively. Then, after the first iteration, the flagged s-variable causes the class SECRET to appear in the s-expression as follows.

$$\underline{a}(\text{SECRET}) \leftarrow \oplus(\text{LOW}, \text{SECRET}, b(*))$$

Note that $\underline{b}(*)$ does not disappear from the s-expression, indicating that b may flow into a again.

If the loop iterates one more time, then again the flagged s-variable causes another class to be added to the s-expression. This time the added class is CONFIDENTIAL since c flows into b before the second flow from b to a. Then, we have the following s-check.

$$\underline{a}(\text{SECRET}) \leftarrow \oplus(\text{LOW}, \text{SECRET}, \text{CONFIDENTIAL}, b(*))$$

Note how the use of the flag allows the mechanism to check all flows into a that are caused by any iteration of the loop.

If a flagged s-variable appears in an s-definition, it has a slightly different meaning. For example, assume that a is a dynamically-bound variable. Then, from the above code, we get the following s-definition

$$\underline{a} \equiv \oplus(\text{LOW, b(*)}),$$

which indicates that b flows into a within a loop. In this case, if the loop iterates more than once, there is no need to accumulate the flows. It is enough to just replace the previous flows. If b and c have the same initial classes as above, then during the first iteration, we have

$$\underline{a} \equiv \oplus(\text{LOW, SECRET(b(*))}),$$

which indicates that the new class of a is SECRET. However, $\underline{b}(*)$ does not disappear from the s-expression because $\underline{a}$ may still change inside the loop. As a matter of fact, during the second iteration of the loop, the class of a becomes CONFIDENTIAL and we get the following s-definition.

$$\underline{a} \equiv \oplus(\text{LOW, CONFIDENTIAL(b(*))}),$$

Note how the class of a after the first iteration is changed since the SECRET information is overwritten by the CONFIDENTIAL information in c.

4.2.3.3 Inter-module implicit flows Inter-module implicit flows are handled by associating a special s-variable, implicit, with each procedure call. An s-definition for implicit in a template for a procedure P takes the following form

$$\underline{\text{implicit}} = \oplus(\text{ EXP, IMP})$$

where EXP is an s-expression representing the class on which the call is conditioned and IMP is the class of the incoming implicit flow sent by the caller to P. Implicit represents the class of the implicit flow into to the called procedure and is sent along with the call message.

Since implicit flows occur even if the procedure call is skipped at run time, the compile-time mechanism must generate statements to send implicit to the skipped procedure. This is achieved by adding *send-probe* statements to conditional procedure calls. For example, a statement like

```
if (EXP) then M.Q()
```

is compiled into

```
if (EXP) then M.Q() else send-probe(M.Q).
```

The send-probe statement, if executed, handles sending implicit when an RPC is skipped. At the receiving RM the class of the incoming implicit flow is denoted by the special s-variable IMP.

Send-probe statements are also inserted as the first statements following the loop. They are needed to handle inter-module global implicit flows.

4.2.3.4 Updating classes of dynamic state variables Potentially, the class of each dynamic state variable in a module must be updated every time a message (RPC, probe, or return) is sent by the module. This is done so as to keep the class of the dynamic state variables up-to-date while the process that is sending the message is suspended (or terminated). This updating is a necessary expense since

another process may need to access the class of the state variable[4]. This enhancement provides the first step[5] in removing the restriction of the original mechanism in [27] of not allowing internal concurrency in RMs.

For each procedure call, the compile-time mechanism generates an s-definition for each dynamic state variable that has possibly been written into before a request for sending an RPC or return message. This s-definition is used to update the class of the state variables before sending that message. We use the s-variable dsv⇑M.F to denote the class of the dynamic state variable dsv *before* the call (or probe) to M.F. The s-definitions in STATES are used to update the classes of the dynamic state variables upon returning.

Similarly, we introduce the s-variable dsv⇓M.F to denote the class of a dynamic state variable dsv *after* the call (or probe) to M.F. This type of s-variable is used in the s-expressions in s-definitions of variables that receive flow from a dynamic state variable after an RPC or probe request. Note that if the dynamic state variable is actually an actual OUT parameter than dsv⇓M.F is still simply denoted by M.F.dsv (which represents the class after the call). However, the new notation is needed since the class of a dynamic state variable may be modified by another concurrent process starting in the same RM while the caller process is suspended. Therefore, dsv⇓M.F may actually be different than dsv⇑M.F even if dsv is not an actual OUT parameter.

We present two examples to illustrate the above ideas. In the first example shown in Figure 4.5, we assume that a call to procedure M1.P is received by M1 which in turn causes a chain of RPCs to be made. The chain can be represented as

[4]We will see in the next chapter that this other process must belong to the same transaction as the suspended or terminated process.

[5]The concurrency control mechanism of the next chapter provides the other step.

```
M1 ={ int dsv1;
      int ssv1(LOW);
      int ssv2(SECRET);

   init()
   {   ...
       dsv1 = ssv1;
   }

   P()
   {   dsv1 = ssv2;
       M2.R();
   }

   Q()
   {   ...
       ssv1 = dsv1;
   }
}
```

```
M2 = {
   ...

   R()
   {   ...
       M3.S();
   }
}

M3 = {
   ...

   S()
   {   ...
       M1.Q();
   }
}
```

Figure 4.5: An example illustrating module-recursive calls

$$M1.P() \rightarrow M2.R() \rightarrow M3.S() \rightarrow M1.Q().$$

Note how the chain starts and ends in the same module, M1. The RPC to M1.P (indirectly) results in a call back to a procedure in M1. Such RPCs, which directly or indirectly cause the same module to be called more than once in the RPC chain, are called *module-recursive* RPCs.

In the example, the class of dsv1 is initially LOW when the module is created. When the module-recursive call M1.P is received, dsv1 gets SECRET information from ssv2. Then, the call to M2.R occurs. Eventually, M1.Q is called and the s-check

$$\underline{ssv1}(LOW) \leftarrow \oplus(dsv1, IMP)$$

from its template is used to check the flow into ssv1.

Assume the current class of dsv1 is not updated before sending the RPC message to M2.R from M1.P (i.e., $\underline{dsv1}$ remains LOW). Then, substituting LOW for the s-variable $\underline{dsv1}$ in the s-check detects no flow violation as the s-check becomes

$$\underline{ssv1}(LOW) \leftarrow \oplus(LOW, IMP)$$

which shows a flow from LOW to LOW. However, it can obviously be seen that actually dsv1 holds SECRET information (since ssv2 is assigned to dsv1 before the call to M2.R) and that M1.Q should not be executed since it would result in an illegal flow.

On the other hand, if the class of dsv1 is updated before the RPC is sent then the flow violation will be caught. With the current $\underline{dsv1}$ set to properly to SECRET, when M1.Q is received, its s-check is evaluated to

$$\underline{ssv1}(LOW) \leftarrow \oplus(SECRET, IMP)$$

resulting in an obvious illegal flow. Therefore, we conclude the necessity to update classes of dynamic state variables before a message is sent out if module recursive calls are permitted.

The second example is similar to the first one and is shown in Figure 4.6. The chain of calls is the same and the module-recursive call is still M1.P. Here a flow violation occurs after the call to M2.R returns and dsv1 is assigned to ssv2. This is a result of having M1.Q assign TOPSECRET information to dsv1 whereas ssv2 is a SECRET static variable.

By using the new notation and generating the s-check

$$\underline{\text{ssv1}}(\text{LOW}) \leftarrow \oplus(\text{dsv1}\Downarrow\text{M2.R}, \text{IMP})$$

the violation is caught when M2.R returns and $\underline{\text{dsv1}}\Downarrow$M2.R is replaced with the class of dsv1 *after* the call, which is TOPSECRET.

The s-check

$$\underline{\text{ssv1}}(\text{LOW}) \leftarrow \oplus(\text{dsv1}, \text{IMP})$$

would not catch the violation since the s-variable $\underline{\text{dsv1}}$ would be replaced by a class of dsv1 before the call to M2.R. Therefore, we see the need for the new notation.

4.2.3.5 Ambiguous and unambiguous assignments Statements, which assign or may assign a new class to a dynamic state variable dsv, are called *security assignments* or *s-assignments*. We will say that such statements *s-assign* dsv. Examples are statements that read dsv, procedure calls with dsv as an actual OUT parameter and of course, assignment statements. Those are called *unambiguous s-assignments* since they with certainty define the class of dsv.

```
M1 ={ int dsv1;                          M2 = {
      int ssv1(LOW);                        ...
      int ssv2(SECRET);
      int ssv3(TOPSECRET);                  R()
      ...                                   {   ...
                                                M3.S();
   P()                                      }
   {   dsv1 = ssv1;                      }
       M2.R();
       ssv2 = dsv1;                      M3 = {
   }                                        ...

   Q()                                      S()
   {   ...                                  {   ...
       dsv1 = ssv3;                             M1.Q();
   }                                        }
}                                        }
```

Figure 4.6: An example illustrating module-recursive calls

As explained earlier, a procedure call in which dsv does not appear as an actual OUT parameter may still result in a change in dsv. This type of s-assignments that only possibly modify the class of a dynamic state variable are *ambiguous.* As another example, an assignment through a pointer is also considered an ambiguous s-assignment of any dynamic variable dsv if it is possible that the pointer points to dsv since at execution time the pointer may not point to dsv at all.

The example of Figure 4.6 shows how a procedure call, M1.R, can ambiguously assign a new class to a dynamic state variable, dsv1. The use of the s-variable dsv1⇓M2.R implies an ambiguous s-assignment. On the other hand, M2.R.dsv1 would imply an unambiguous s-assignment caused by dsv1 being an actual OUT parameter.

Finally we note that statements that cause an implicit flow into a dynamic variable dsv are said to *increment* its class. For example, since probes carry only

SUBJECT	Clearance of the user on whose behalf the program is running
dsv	a dynamically-bound state variable
ssv	a statically-bound state variable
x	Class of the variable x, dynamic or static
P.a	Input variable carrying information into P through the formal IN parameter, a, of P
P.a	Class of the information in P.a
M.Q.b	Input variable into the caller of M.Q corresponding to the actual OUT parameter of Q
M.Q.b	Class of the information in M.Q.b
implicit	Class of the implicit flow sent out by a procedure to its callee
IMP	Class of the implicit flow received by a procedure from its caller
dsv⇑M.Q	Class of a dynamically-bound state variable prior to the call or probe to M.Q
dsv⇓M.Q	Class of a dynamically-bound state variable upon returning from the call or probe to M.Q when it is not an actual OUT parameter of Q

Figure 4.7: Summary of notations

implicit flows, they are s-assignments that increment the classes of dynamic state variables. They can only change classes of variables to a higher class. We will say that a probe *s-increments* a dynamic state variable.

4.2.3.6 Summary of notations In Figure 4.7, we show a summary of the notations we will use.

4.2.3.7 Example In Figure 4.8, we give an example of a simple template to illustrate the notations and notions presented in this section. The given template is for a procedure R.P with two IN and one OUT parameters. In the s-definition

part, under the OUTS category, the s-definition of the OUT parameter c defines its security class upon completion of the procedure. The class of c is sent in the return message along with the value of c. The s-expression defining c shows that SECRET information flows into c. In addition, two dynamic variables flow into c. M1.Q.e representing the class of the actual OUT parameter from the call to M1.Q. Also, the dynamic state variable dsv1 flows into c but the flow occurs after the call to M1.Q; the use of dsv1⇓M1.Q implies that M1.Q ambiguously defines dsv1 which does not appear as an actual OUT parameter in the call. Finally, the implicit flow, IMP, received by P from its caller, is shown since it flows into all variables that are written by P.

Under the INS s-definitions, one procedure call to M1.Q is listed. The class of the actual IN parameter for the call is given by an s-definition. Also, s-definitions for implicit and dsv1⇑M1.Q are included. Implicit and a are sent with the RPC. Notice how a is computed using the class of P.a which is an actual IN parameter to P. P.a is determined from the RPC message calling P. The s-definition dsv1⇑M1.Q is used to update dsv1 before the RPC M1.Q is sent.

Under the STATES s-definitions, the final classes of dsv1 and dsv2, after the execution of P, are given by s-definitions.

The s-checks part shows that two static variables, b and ssv, receive flow in P. These flows must be checked at run time when the values of the s-variables can be determined.

Note that the compile-time mechanism can also use the s-checks to reject a procedure if they show an illegal flow. For example, the following s-check can be used to immediately reject a procedure at compile time

R.P(a,b | c) TEMPLATE

S-definitions Part

OUTS

c ≡ ⊕(SECRET, M1.Q.e, dsv1⇓M1.Q, IMP)

INS

M1.Q(a | e, dsv2)

implicit ≡ ⊕(CONFIDENTIAL, IMP)

dsv1⇑M1.Q ≡ ⊕(SECRET, IMP)

a ≡ ⊕(P.a, IMP)

STATES

dsv1 ≡ ⊕(CONFIDENTIAL, M1.Q.dsv2, IMP)

dsv2 ≡ ⊕(M1.Q.dsv2, IMP)

S-checks Part

b(CONFIDENTIAL) ← ⊕(P.b, IMP)

ssv(SECRET) ← ⊕(SECRET, M1.Q.e, IMP)

Figure 4.8: An example information template

```
int ssv(SECRET);      !ssv is a SECRET static state var.
int dsv1, dsv2;       !dsv1 and dsv2 are dynamic state vars.

P(a,b | c)
int a, b(CONFIDENTIAL);
real c;
{   bool e;

    if (b < 0) then {
       dsv1 = ssv;
       M1.Q(a | e, dsv2);
    } else
       send-probe(M1.Q);  !inserted by compile-time mechanism
    if (e) then
       c = dsv1 * ssv;
    else
       ssv = ssv + e;
    dsv1 = b + dsv2;
}
```

Figure 4.9: Code for the example template

$$\underline{b}(\text{CONFIDENTIAL}) \leftarrow \oplus(\text{SECRET, P.b, IMP}).$$

It shows an illegal flow regardless of what run-time flows occur.

Notice how we can deduce the information flow caused by the procedure by looking only at the template. The template can match many different procedure codes. However, we are not interested in the codes but only with flow they cause. Most of the times we will only show templates and not the code from which they are generated. For illustration purposes, in Figure 4.9, we show a possible procedure code for the template in Figure 4.8.

4.2.4 The run-time mechanism

The information flow control run-time mechanism certifies the security of RPCs at message passing time. When a procedure is invoked, a *flow instance* of its flow template is created in the called module. When messages are received or sent, the mechanism replaces all the s-variables in the flow instance whose values are known at that instant by their class values. Then, it proceeds by evaluating the s-checks to detect any possible security violations.

For example, assume a call R.P(x, y, z) is received for the procedure of Figure 4.8, where x is SECRET, y is LOW. Also, assume that the call is conditioned on a CONFIDENTIAL class (i.e., IMP is CONFIDENTIAL). Then, the template instance of the call will be as shown in Figure 4.10. Note that the s-checks show no flow violations at this point and the call can proceed. Also note that, in this case, the INS s-definitions for M1.Q can be completely evaluated to show that the call will be conditioned on CONFIDENTIAL, that the actual IN parameter is SECRET, and that dsv1 must be updated to secret before the call (or probe) leaves the module.

The details of the steps that are taken by the run-time algorithm follow. The actions that are taken depend on the type of the message (i.e., RPC, probe, or return) and on whether a message is being sent or received.

4.2.4.1 Handling RPCs When an RPC to M.P is received, a new instance of the template of M.P is created, and then, the following steps are taken.

1. In the flow instance, replace all s-variables corresponding to dynamic state variables with their current classes. All these s-variables are of the form dsv; the values of the other forms such as the ones corresponding to ambiguous

R.P(a,b | c) TEMPLATE INSTANCE

S-definitions Part

OUTS

c ≡ ⊕(SECRET, M1.Q.e, dsv1⇓M1.Q, CONFIDENTIAL)

INS

M1.Q(a | e, dsv2)

implicit ≡ ⊕(CONFIDENTIAL, CONFIDENTIAL)

dsv1⇑M1.Q ≡ ⊕(SECRET, CONFIDENTIAL)

a ≡ ⊕(SECRET, CONFIDENTIAL)

STATES

dsv1 ≡ ⊕(CONFIDENTIAL, M1.Q.dsv2, CONFIDENTIAL)

dsv2 ≡ ⊕(M1.Q.dsv2, CONFIDENTIAL)

S-checks Part

b(CONFIDENTIAL) ← ⊕(LOW, CONFIDENTIAL)

ssv(SECRET) ← ⊕(SECRET, M1.Q.e, CONFIDENTIAL)

Figure 4.10: An example flow instance

s-assignments (dsv⇓M.P) are not known at this point.

2. In the flow instance, replace all occurrences of IMP by the security class of the implicit flow caused and carried by the RPC.

3. In the flow instance, replace the s-variables for all formal IN parameters (of the form P.a) by their value carried in by the RPC.

4. Evaluate the s-checks; in other words, check for possible flow violations as determined by the s-checks. Report any violations to the user and abort.

5. Check that all implicit's in the INS part are bounded by the security clearance of the subject on whose behalf the call is running. If it is not the case, report low clearance to the user and abort.

The execution of the call is allowed to proceed only if steps 4 and 5 do not detect any potential flow violations. Step 5 is taken so as not to leak any information to the subject through flow error messages. As discussed earlier, performing this check at this particular point of receiving the RPC does make the mechanism less precise than if the check is embedded in the code of the procedure.

When a procedure P makes an RPC (e.g., M.Q), the following steps are taken.

1. P is suspended.

2. All s-definitions in the INS part for that call are evaluated. Accordingly, the current classes of the dynamic state variables that have been written into are updated using the s-deflnitions of s-variables of the form dsv⇑M.Q. Also, the classes of the actual IN parameters and of implicit are sent along with the RPC

message. Those classes are used, as explained above, in the flow instance of the procedure receiving the message.

4.2.4.2 Handling probes In case a probe is being sent, the same actions as in sending an RPC are taken except that the part handling the actual parameters is skipped. Namely,

1. the procedure is suspended,
2. the classes of the dynamic state variables are updated, and
3. implicit is sent along with the probe message.

When a module receives a probe for a certain procedure, the way it is handled is very similar to the way RPCs are handled. One obvious difference is that there are no formal IN parameters to handle.

Another difference is that probes are *propagated* to check for the security of the implicit flows in further skipped procedure calls and to update the classes of dynamic state variables in the objects that they visit.

There is always an *initial probe* that triggers this propagation process. We call the procedure that sends that initial probe the *probe originator.* It is the module receiving the initial probe that starts propagating it to other modules which in turn, must propagate it too. All the modules visited by the probe are simply called *probe recipients.*

Probe recipients will be identified by the name of the procedure mentioned in the probe message they receive. For example, assume a procedure M1.P executes a send-probe(M2.Q) statement. Also, assume that in M2.Q three different procedure

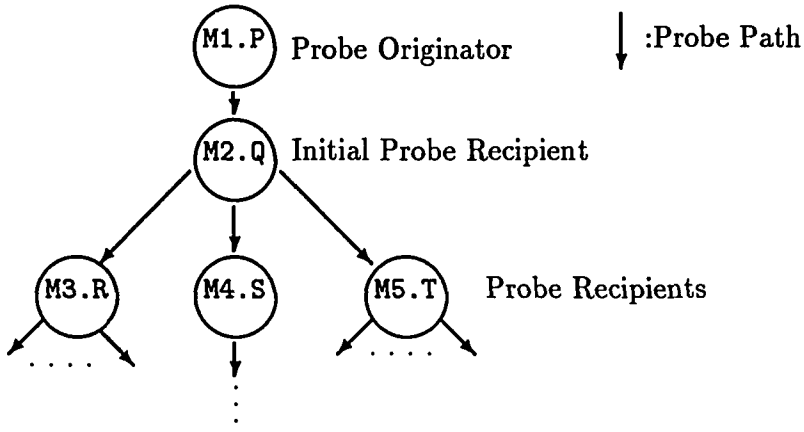

Figure 4.11: Example of a probe tree

calls appear: M3.R, M4.S, and M5.T. Then, M1.P (the probe originator) initially sends the probe to M2.Q (the initial recipient) which in turn propagates the same probe carrying the same implicit flow class to M3.R, M4.S, and M5.T (the probe recipients). Of course, those probe recipients may still need to propagate the probe to other recipients. The path of the propagated probe of this example is shown in Figure 4.11. Notice how the path and the visited procedures starting with the initial recipient form a tree. We refer to such trees as *probe trees.*

Next we present the way incoming probes are handled at run time. First, the recipient procedure of the probe checks if it is being revisited by the same probe. If it is the first time it receives the probe, a flow instance for the procedure is created. Then, the following steps are performed.

1. The special s-variable IMP is replaced in the s-checks of the template instance by the class of the implicit flow carried in by the probe.

2. Check that IMP $\leq \underline{sv}$, where sv is a static variable that has an s-check in the template, and thus, appears as receiving flow in the procedure code. This step is equivalent to evaluating the s-checks since only IMP among the s-variables has a known value when a probe and not an actual call is received

3. If no violations are detected, the probe is propagated to all procedures that appear under INS (i.e., that could be called by this procedure). If the INS part is empty (i.e., the probe recipient is a leaf in the probe tree), no propagation occurs and the following steps are taken.

   (a) S-increment the dynamic state variables that are written into in the procedure code by the class of the implicit flow carried in by the probe.

   (b) Send a *probe-certified* message to the sender of the probe.

   (c) Delete the flow instance.

   Upon receiving a probe-certified message from all its children in the probe tree, a module, in turn, gets to perform the above three steps. Eventually, if no violations are detected, the probe originator receives a special probe-certified message, a *probe-return*, from the initial probe recipient. Only then will the originator be unsuspended.

4. If a flow violations is detected at any point during the propagation of the probes, then a message to that effect is propagated to all probe senders upwards trough the tree. The originator of the probe is not allowed to proceed and a flow violation is reported.

If the same probe revisits a procedure, no matter which procedure is its sender, there is no need to redo the checking and propagating. The revisited procedure simply responds to the revisiting probe the same way it responded to the probe at the time of its first visit.

4.2.4.3 Handling returns In addition to probe-returns, there is another type of return messages: RPC-returns. An *RPC-return* message is sent when a procedure terminates.

When sending an RPC-return, the s-definitions in OUTS are evaluated and the results are sent along with the return. Also, the classes of the dynamic state variables are updated according to the s-definitions in STATES. Finally, the flow instance is destroyed.

When a module receives an RPC-return message, the following steps are taken.

1. The flow instance is updated by replacing all s-variables of the form M.P.dsv corresponding to actual OUT parameters of the call.

2. The s-variables holding the class of dynamic state variables after the call are updated. These s-variables are of the form dsv⇓M.P in the template and they take the value of the current classes of the state variables they represent. An s-variable dsv⇓M.P indicates an ambiguous assignment.

3. Finally, the s-checks are evaluated and the class of the subject is tested against the s-definitions of implicit's in INS. This is done because some of the s-variables might not have had a known value when flow violation were checked first.

Only the above latter two steps are taken when a module receives a probe-return

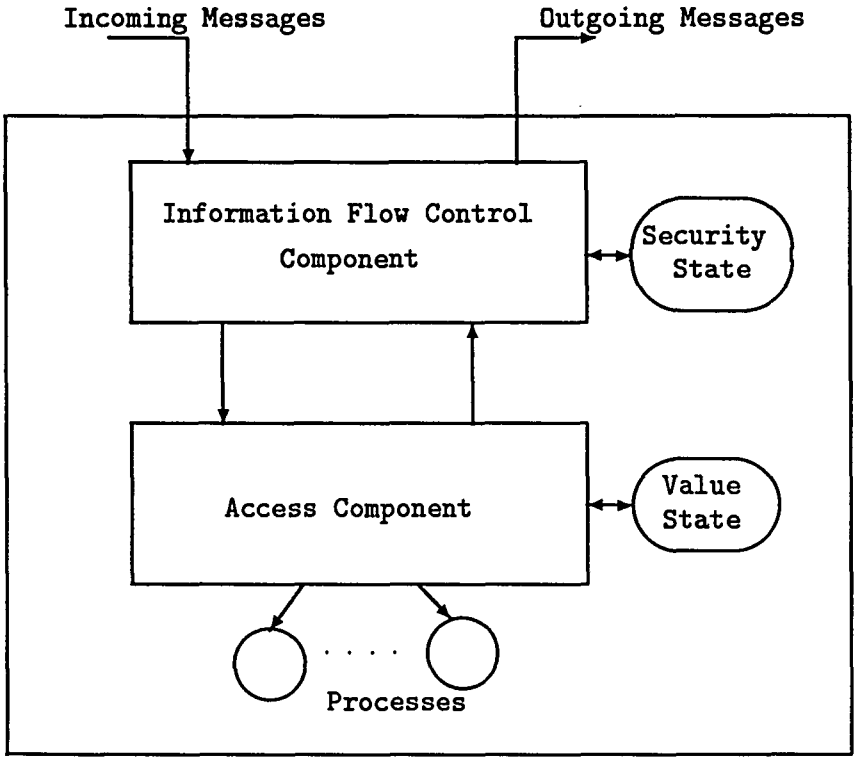

Figure 4.12: Components of a resource module object

message for a probe it originated. Obviously, there is no need to handle actual OUT parameters when probes return.

4.3 The Protected Resource Module Model

The *Protected Resource Module* (PRM) model is the resource module model augmented by the information flow control mechanism. An RM can now be diagramatically represented as shown in Figure 4.12.

The figure shows that each RM can be logically thought of as having two components. The information flow control component receives and sends messages, runs the

flow control mechanism, and manages the security state of the RM. The access component manages the user processes that are created by received RPCs. It manages the value state of the RM.

4.4 Shortcomings of the Flow Model

4.4.1 Restrictions on concurrency

In the information flow control mechanism we presented in this chapter, we have shown how module-recursive processes can be handled. However, another important restriction is that the mechanism does not provide a concurrency control mechanism to allow for preserving the consistency of the state of modules in the presence of internal module concurrency. Most importantly, we are interested in preserving the consistency of the security state of modules with dynamically-bound state variables. The mechanism must control concurrent accesses to the classification information of the shared variables. Otherwise, if several processes that are running concurrently within a module try to access the same state variable, they may get unexpected results. Worse yet, they may cause flow violations to go undetected.

A simple solution is of course not to allow dynamically-bound state variables. However, we argue that allowing dynamic static variable adds certain advantages to our system. For example, assume an object encapsulates a state variable sv. If sv is statically-bound to the class SECRET, then no user with clearance less than SECRET may read it. However, sv may actually contain CONFIDENTIAL information, in which case preventing a user with a CONFIDENTIAL clearance from reading the variable is unnecessary. Obviously, by using a dynamic state variable, the class of the variable will reflect the class of the information stored in it and no unnecessary

denials of reading requests will occur.

Also, in the static case, no TOPSECRET information is allowed in sv. This necessitates the creation of different instance objects (and may be different classes) to encapsulate state variables of different classifications.

So, we would like to have dynamically-bound state variables, but this will require the addition of a concurrency control scheme. In Figure 4.13, we show how inconsistencies can result in the absence of such scheme. Assume that M1 and M2 get instantiated and that their RPCs, M3.Read and M3.Write from their init procedures, are received at the same time at M3 and two concurrent process are created to service those calls. Also, assume that initially dsv1 is LOW and that the statement `dsv1 = a` gets executed first. At this point dsv1 contains SECRET information since the actual IN parameter to the call is x and it is classified at SECRET. Then if M3.Write is interrupted before it executes the return, the run-time mechanism is not given the chance yet to update the current class of dsv1.

Now it is obvious that, while M3.Write is suspended, M3.Read can execute and return with a return message that carries a LOW class for the formal OUT parameter b. Whereas, at the same time, b carries SECRET information from dsv1. Therefore, when the return message arrives at M1, the s-check

$$\underline{y}(\text{LOW}) \leftarrow \oplus(\text{M3.Read.y, IMP})$$

in the template instance of init is evaluated to

$$\underline{y}(\text{LOW}) \leftarrow \oplus(\text{LOW, IMP})$$

and the flow violation is not detected.

```
M1 ={ ...

   init()
   {   int y(LOW);

       ...
       M3.Read( | y);
   }
}

M2 ={ ...

   init()
   {   int x(SECRET);

       ...
       M3.Write(x);
   }
```

```
M3 ={ int dsv1;

   Read( |b)
   int b;
   {
       b = dsv1;
   }

   Write(a)
   int a;
   {
       dsv1 = a;
   }
}
```

Figure 4.13: Undetected illegal flows in the absence of concurrency control

The flow violation of the example is present because the security state of the RM was left inconsistent as a result of uncontrolled concurrency. Concurrency can also lead to an inconsistent value state of an RM. The lost update and inconsistent retrieval anomalies of Chapter 3 may occur in the absence of concurrency control. Internal module concurrency is an important requirement in our model, but such insecure behavior cannot be tolerated.

In the next chapter, we suggest a concurrency control mechanism to remove the restrictions on concurrency to allow dynamically-bound state variables in RMs.

4.4.2 Overclassification of dynamically-bound state variables

When s-definitions are generated for dynamically-bound variables, they show all possible flows into the variables. They do not take into consideration the actual execution path taken by a process. As a result, when those s-definitions are evaluated, they usually give an upper bound on the class of the information stored in the variables rather than the actual class.

For example, for

```
if (e1) then dsv = e2 else dsv = e3;
```

the following s-definition is generated

$$\underline{dsv} = \oplus(e1, e2, e3, IMP).$$

That obviously is an overclassification of dsv since in no case will the statement cause both e1 and e2 to flow into dsv.

In chapter 6, we develop a new form of s-definitions which allows a more precise calculation of the classes of dynamic variables by including information about the execution path of the processes.

The increased precision in the computing of the classes of dynamic variables will also result in a more precise flow control mechanism (in the sense that less secure programs will be rejected). The reason is that the s-variables of dynamic variables in the s-checks will tend to be less overclassified. For example, if the s-check

$$\underline{ssv}(SECRET) \leftarrow \oplus(dsv, IMP)$$

is evaluated, it signals a flow violation if dsv is incorrectly overclassified to TOPSECRET. However, if the actual information in dsv is less than TOPSECRET and

dsv is computed more precisely to reflect the class of that information, then a flow violation does not result and the program is not rejected.

5. A CONCURRENCY CONTROL MECHANISM FOR THE PROTECTED RESOURCE MODULE SYSTEM

In this chapter, we suggest a concurrency control mechanism based on Moss' nested transaction model. The mechanism is run by a concurrency control component in the protected RMs. Figure 5.1 shows an RM augmented by the new component. Later, we explain how the information flow control and access components use the concurrency control component to preserve the consistency of their states.

5.1 Introduction

As explained in the previous chapter, an interesting property of our system is that we view the state of each RM as composed of two (sub)states: a *value* state and a *security* state. In other words, each state variable is thought of as encompassing a pair: its value and its security class.

It is obvious that only the consistency of the security states corresponding to dynamically-bound state variables is affected by concurrency; the security state corresponding to statically-bound variables is constant. So, our discussion will mostly concentrate on dynamic variables, noting that the locking mechanisms we develop will be able to be applied to the value state of static variables, their only variable part.

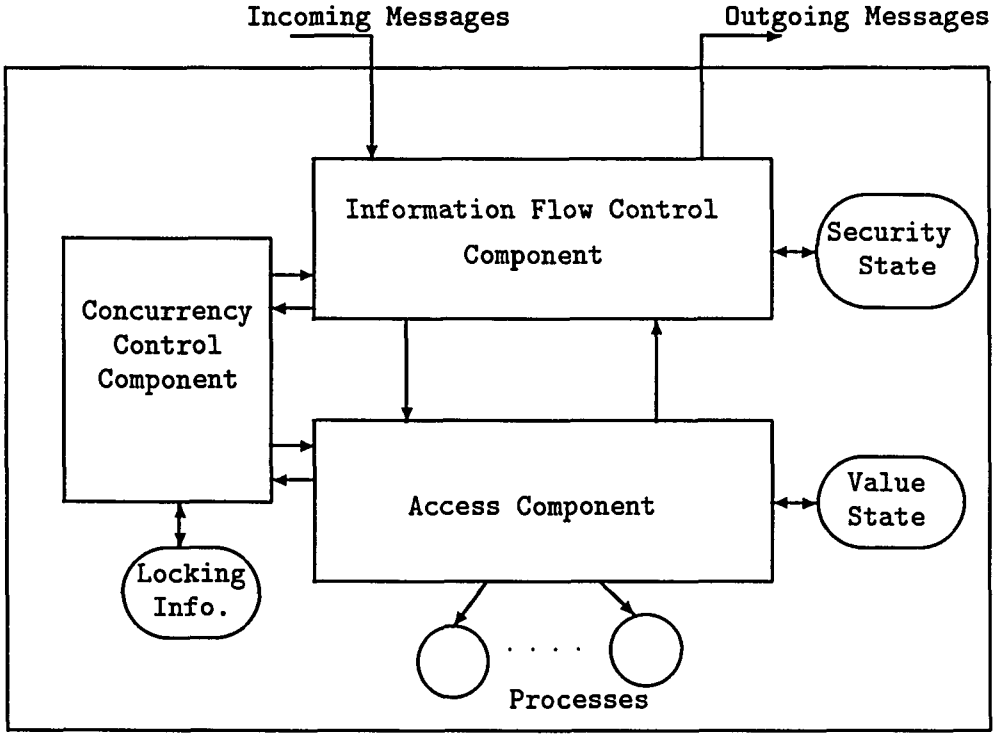

Figure 5.1: An RM object with a concurrency control component

To preserve the consistency of RM states, Moss' nested transaction scheme [31] was found to be a good starting point for our investigation because of its use of locks and its elegant handling of RPCs. However, we found that in its original form, using two locks, the scheme may not actually be adequate. The reason is that we are concerned with the performance of the system if send-probe operations become frequent. Since probes propagate to RM objects in a tree-like fashion, a probe initiated at a certain object may actually end up being sent to an exponential number of other objects. Moreover, since probes must cause the class of some dynamic state variables to be updated, they must lock those variables, preventing other operations (and other probes) from accessing the variables. Therefore, a decrease in concurrency may occur.

A couple of major observations about probes were made. First, probes only access the class part of a variable in an object; they do not access the value of a variable. This may possibly mean that a probe need not prevent other operations from accessing the value of a variable.

For the second and most important observation, we assume that a class increment operation using the least-upper-bound operator, $\oplus$, can be performed atomically on a dynamic state variable. In other words, for a dynamic state variable dsv and a security class C, the following class assignment operation

$$\underline{\text{dsv}} = \underline{\text{dsv}} \oplus \text{C}$$

is atomic[1]. Then, we note that probes access the class of a variable in a very specific way: they increment it by using $\oplus$; they do not just modify it completely. Because

[1] As explained in Chapter 3, standard mutual exclusion techniques can be used to implement atomicity of such a basic operation.

of the assumed atomicity of class increment operations and of the associative and commutative properties of $\oplus$, different probes may freely attempt to increment the class of the same variable concurrently.

To illustrate, assume two probes, carrying implicit flows with classes C1 and C2 respectively, are sent to the object containing dsv. It does not matter which of the probes is allowed to go first in updating dsv since $\oplus$ is commutative and associative; in other words,

$$(\underline{\text{dsv}} \oplus \text{C1}) \oplus \text{C2}$$

has the same value as

$$(\underline{\text{dsv}} \oplus \text{C2}) \oplus \text{C1}.$$

As a consequence, a nested transaction scheme with a locking mechanism which allows probes to execute concurrently and which separates locking the value of a variable from locking its class is desirable for better concurrency. The sought locking mechanism should relax the constraints on the concurrent execution of probes while it should still impose the necessary constraints on the read and write operations of classes and values.

5.2 Overview

Moss' nested transaction model is employed to run programs as nested transactions and to implement RPCs and probes as subtransactions as described in Chapter 3. The root of a transaction tree is the initialization procedure of an RM that is instantiated by the user. The other (sub)transactions at all other levels of the tree are caused by either RPCs or probes.

As in Chapter 3, when an RPC is requested, two subtransactions are created to service it: a *local call transaction* at the site of the calling RM and a *remote call transaction* at the site of the called RM. Similarly, we will talk about two transactions that service a probe a *local probe transaction* and a *remote probe transaction.*

For example, assume that the following code is being executed as part of a transaction T on site A.

```
...
M1.P();
if (a = 0) then
   M2.Q();
ELSE
   send-probe(M2.Q);
...
```

Also, assume that M1 resides on remote site B and M2 on remote site B, and finally, that the value of a is not 0. Then, the transaction subtree with root T is as shown in Figure 5.2.

A local transaction takes care of sending the message while a remote transaction handles receiving the message. When the run-time flow control mechanism is called upon at message passing time, its actions are included within the local or the remote transaction, depending on whether it is invoked as a result of an incoming or an outgoing message.

In the next sections, we first present a locking mechanism that employs five types of locks in such a nested transaction model. Explaining the mechanism will allow us to present the issues that are involved in adding a locking mechanism to our

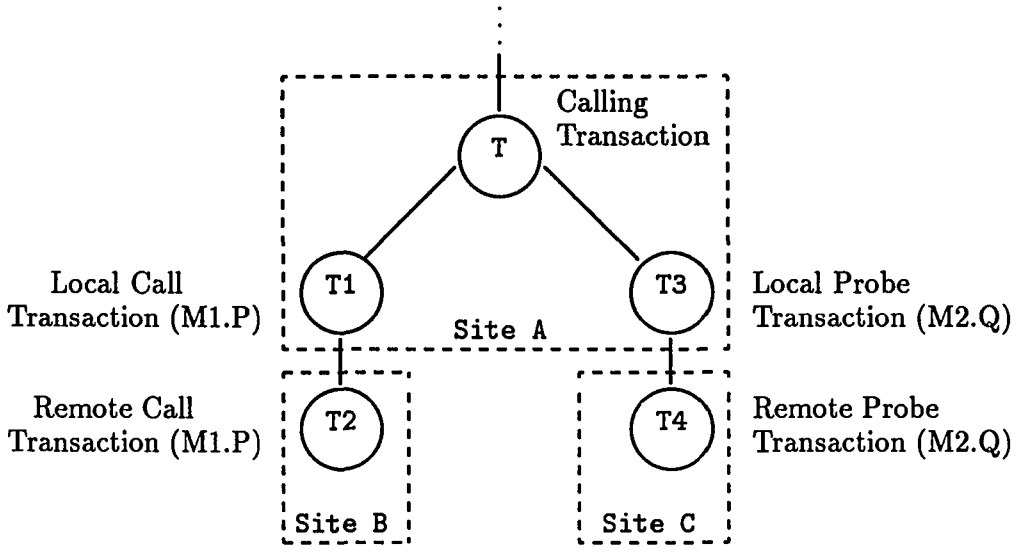

Figure 5.2: Tree showing subtransactions handling probes and RPCs

protected RM model.

After introducing the locking mechanism, we will be able to suggest further modifications to the run-time flow mechanism. The modified flow mechanism will be more precise since it will be able to take into consideration part of the actual execution path of procedures when evaluating s-expressions.

Then, we argue for using a locking mechanism with a reduced number of lock types. Reducing the number of lock types cuts down on the overhead of the first mechanism, but results in a (not very significant) decrease in the degree concurrency among transactions.

5.3 An Initial Locking Mechanism

The idea behind our initial locking mechanism is to allow for the complete decoupling of the value and class of the variables so as to permit as much concurrency as possible (while preserving consistency, of course). The decoupling is made by categorizing the operations that can be performed on the dynamic state variables into *value operations*, which deal with the value part of a variable, and *class operations* dealing with the class part.

For a variable s, there is a total of five basic (assumed atomic) operations that can be performed on it. There are two obvious operations on its value: a *value-read* on s (vr[s]) and a *value-write* on s (vw[s]). Those value operations are performed by the processes executing procedures inside (the access component of) the RM objects.

In addition, there are three possible atomic class operations: a *class-read* on s (cr[s]), a *class-write* on s (cw[s]) and a *class-increment* on s (ci[s]). These operation are used by the information flow control run-time mechanism. For example, a probe performs ci[dsv] operations for all dynamic state variables dsv, such that dsv receives a flow in the probed procedure.

In this section, we consider a mechanism in which each of the above operations has a different corresponding lock type. For a variable s, the five lock types are denoted as follows: *value-read-lock* on s (vrl[s]), *value-write-lock* on s (vwl[s]), *class-read-lock* on s (crl[s]), *class-write-lock* on s (cwl[s]), and *class-increment-lock* on s (cil[s]).

The two value locks are requested by a user process servicing an RPC request. For a state variable s, vrl[s] is requested before a read operation is attempted by a process on the variable, while vwl[s] is needed before attempting a write operation.

Those value locks are requested for both static and dynamic state variables. For example, consider the following procedure that accesses some state variables.

```
int ssv(LOW), dsv1, dsv2;
...
P()
{   int  a;

    if (dsv1) then
    {   a = dsv2;
        dsv2 = ssv;
        ssv = a;
    }
}
```

If a process is running P in the access component of the RM, then the lock requests are as shown in the following code.

```
P()
{   int  a;

    LOCK-REQUEST(vrl[dsv1]);
    if (dsv1) then
    {
        LOCK-REQUEST(vrl[dsv2]);
        a = dsv2;
```

```
            LOCK-REQUEST(vrl[ssv]);
            LOCK-REQUEST(vwl[dsv2]);
            dsv2 = ssv;
            LOCK-REQUEST(vwl[ssv]);
            ssv = a;
        }
    }
```

The lock requests are passed to the access component that is running the process. The requests are finally handled by the concurrency control component.

The three class locks are used by the information flow run-time mechanism (the flow control component) at message passing time. They are exclusively requested for dynamic state variables since static state variables have a fixed class. The type of lock needed depends on the type of the passed message and whether it is being sent or received.

By looking at the different steps that the run-time flow control mechanism performs for each message type, a very simple way for using the three class lock types can be devised. Then, for a dynamic state variable dsv, some simple lock usage rules can be listed as follows.

1. Whenever the mechanism calls for replacing a dsv s-variable of any form (dsv, M.Q.dsv or dsv⇓M.Q) in the template instance by the current class of dsv, it must request crl[dsv] before attempting the cr[dsv] operation.

2. Whenever the mechanism calls for updating the current class of dsv, it must request a cwl[dsv] before attempting the cw[dsv] operation.

3. Whenever the mechanism needs to increment the class of dsv as a result of a certified probe, it must request a cil[dsv] before attempting the ci[dsv] operation.

These simple rules may sometimes result in requesting locks well in advance of attempting an operation. That, in turn, results in a decrease in the degree of concurrency. Allowing transactions to hold locks too early may force other transactions to wait longer for the locks that conflict with the held ones.

For example, the code

```
if (a == 0) then
   M1.P();
else
   send-probe(M1.P);
if (b == 0) then
{
   c = dsv;
   M2.Q(c);
}
else
   send-probe(M2.Q);
```

will result in having the s-definition

$$\underline{c} \equiv \oplus(\text{dsv}\Downarrow\text{M1.P, b, IMP})$$

for the actual IN parameter to the call to M2.Q. Assume the transaction running the code is allowed an early hold on the lock crl[dsv] to replace the s-variable $\underline{\text{dsv}}\Downarrow$M1.P

by it current value when the RPC is received. Then, in general, other transactions will be blocked from performing other class operations on dsv. The blocking may be for a long period especially if the call or probe to M1.P takes a long time.

Moreover, to make matters worse, b may be nonzero. In that case, the class of dsv may not need to be read at all. Then, holding the lock becomes wasteful and blocking the other transactions completely unnecessary.

As a result, we would like to improve on the simple lock usage rules. We next explain how the above five locks can be better used at run time. The main idea is to request locks only when necessary.

5.3.1 Lock usage at run time

Here, we discuss lock usage rules which show at exactly what points each lock type is requested. In the rules, the dynamic state variable on which a lock is requested depends on the position and form of its s-variable appearing in the template instance of the procedure receiving or sending the message. In Figure 5.3, we present a general template in which we show all possible positions of an s-variable representing a dynamic state variable in a template.

In the figure, the s-variables are also shown in all their possible forms. DSV represents the initial current class of a dynamic state variables when the RPC is received and before any code is executed. M.P.DSV represents the class of a dynamic state variable as an actual OUT parameter of a called procedure. Finally, DSV⇓M.P represents the class of a dynamic state variable upon a return from a called procedure in which the variable does not appear as an actual OUT parameter.

We have used DSV to represent a dynamic state variable in general and its differ-

TEMPLATE

S-definitions Part

OUTS

$\underline{b} \equiv \oplus(\cdots, \text{DSV } \boxed{1}, \cdots, \text{M.P.DSV } \boxed{2}, \cdots, \text{DSV}\Downarrow\text{M.Q } \boxed{3}, \cdots)$

INS

`M.P(x, ... | ..., DSV)`

$\underline{\text{implicit}} \equiv \oplus(\cdots, \text{DSV } \boxed{4}, \cdots, \text{M.Q.DSV } \boxed{5}, \cdots, \text{DSV}\Downarrow\text{M.R } \boxed{6}, \cdots)$

$\underline{x} \equiv \oplus(\cdots, \text{DSV } \boxed{7}, \cdots, \text{M.Q.DSV } \boxed{8}, \cdots, \text{DSV}\Downarrow\text{M.R } \boxed{9}, \cdots)$

$\underline{\text{DSV}}\Uparrow\text{M.P } \boxed{10} \equiv \oplus(\cdots, \text{DSV } \boxed{11}, \cdots, \text{M.Q.DSV } \boxed{12}, \cdots, \text{DSV}\Downarrow\text{M.R } \boxed{13}, \cdots)$

STATES

$\underline{\text{DSV}} \boxed{14} \equiv \oplus(\cdots, \text{DSV } \boxed{15}, \cdots, \text{M.P.DSV } \boxed{16}, \cdots, \text{DSV}\Downarrow\text{M.Q } \boxed{17}, \cdots)$

S-checks Part

$\underline{\text{sv}}(\text{SECRET}) \leftarrow \oplus(\cdots, \text{DSV } \boxed{18}, \cdots, \text{M.P.DSV } \boxed{19}, \cdots, \text{DSV}\Downarrow\text{M.Q } \boxed{20}, \cdots)$

Figure 5.3: A general template

ent occurrences throughout the template may or may not refer to the same variable. For easy reference, the different positions of DSV are numbered. The numbers will be used to explain which locks, in general, are requested for each message. In the general case, we may simply assume that differently numbered positions of the s-variable for DSV correspond to different dynamic state variables. We will use dsv_i to denote the dynamic state variable whose s-variable appears in the position marked by $\boxed{i}$.

5.3.1.1 Lock requests by incoming messages There are three types of messages that an RM can receive. When RPCs and probes are received, they are serviced respectively by remote call and remote probe transactions. A return message is handled as part of the transaction receiving the return.

The run-time flow mechanism is called upon whenever a message is received. As

a result, all steps taken by it become part of the actions of the transaction handling the incoming message. The locks that are requested by those transactions depend on the type of the message.

Case 1: Incoming RPC Messages:

Assume that a remote call transaction, say T1, is handling the RPC. The possible locks that T1 may need to request, when the run-time flow mechanism starts running, are

crl[DSV] for positions [4] [18].

Those are the locks needed to only evaluate all implicit's and s-checks for detecting any potential flow violations. Also, only those s-variables that appear in the form DSV are locked at this point; they need to be replaced by the current class of the variable. Since the other forms (M.P.DSV and DSV⇓M.P) do not have a value yet, a lock is not requested for them.

The other s-definitions for the output variables are not evaluated because they are not needed at this point. So, the positions [1], [7], [11], and [15] are not locked when an RPC is received. A question arises on the effects of not requesting locks for the s-variables in those positions, since their values may be needed later, for example, when a return is sent.

Not locking the positions means that another transaction, say T2, is able to access the class of their variables. If the access is only for a class read operation, then T1 is not affected because it will still be able to obtain its class read lock later.

On the other hand, T2 may s-assign or s-increment the variables of the unlocked positions. For example, assume T2 requests and is granted the lock cil[dsv_1] for the s-variable in the s-definition of the formal OUT parameter. Then, the class of dsv_1

may be incremented. So, when T1 gets to evaluate the class of its OUT parameter, it will use the new, possibly higher, class and not the class the variable had when T1 started. That might lead to a flow violation which might not have occurred were T1 allowed to lock dsv_1 at the start. In this case, T1 needs to be aborted.

It is also possible that T1 finds the new class of dsv_1 to be lower. T1 then might avoid a flow that could have occurred if T2 does not change the class of the variable.

This type of situation can exist in any locking scheme. Not allowing early locking for transactions may produce different results than if it were allowed. However, no inconsistencies are introduced and the gain in the degree of concurrency can be substantial[2].

Case 2: Incoming Probe Messages:

When a probe is received, its remote probe transaction requests the lock

cil[DSV] for position $\boxed{14}$.

Cil is the only type of lock that is needed since position $\boxed{14}$ represents the dynamic state variables that receive flow in the code of the probed procedure. Those dynamic state variables needs to be s-incremented by the incoming implicit flow in the probe message.

Case 3: Incoming Return Messages:

When a return is received, it is handled by either a local probe transaction or local call transaction depending on whether it is a probe-return or an RPC-return.

[2]One case in which early locking can be useful is if it is known that a transaction is 'long' and relatively time consuming and that aborting it is more expensive than blocking other transactions. This problem also exists in any transaction system and is beyond the scope of our research. Our main concern is increasing the degree of allowed concurrency; the subject of determining 'long' transactions is not treated.

For an RPC-return, the following lock types are possibly requested:

crl[DSV] for positions [5] [6] [19] [20].

As when an RPC is received, an incoming RPC-return causes the flow mechanism to check for flow violations. For that, the locks that are requested allow the evaluation of the s-checks and of the s-definitions of all the implicit's in INS of the template instance.

The variables that are class-locked correspond to s-variables that are of the form M.P.DSV or DSV⇓M.P, assuming that the return is from M.P. Then, after the call to M.P, the values of those s-variables can be be determined and plugged in in the s-expressions in which they appear.

Note that positions [4] and [18] have already been locked when the caller of M.P started. Also, note that the s-variables of the form M.P.DSV or DSV⇓M.P that appear in other positions are not locked at this point because they are not needed.

Finally, in case it is a probe-return, no locks are requested since no evaluation in the template instance is performed. As in the case of an RPC-return, s-variables of the form DSV⇓M.P are not locked at this point.

5.3.1.2 Lock requests by outgoing messages There are three types of messages that an RM can send. When RPCs and returns are being sent, they are serviced by local call transactions. Whereas, an outgoing probe message is handled as part of a local probe transaction.

The run-time flow mechanism is called upon whenever a message is being sent. As a result, all steps taken by it become part of the actions of the transaction handling the outgoing message. The locks that are requested by those transactions depend on

the type of the message.

Case 1: Outgoing RPC Messages:

In the case where an RPC is being sent by a transaction T1, the possible types of locks that the local call transaction, say T2, requests are connected to the s-definitions of output variables whose classes need to be sent out with the message and to s-definitions of dynamic state variables that need to be updated. In general, the requested locks are:

crl[DSV] for [7] [11]

and

crl[DSV] for [5] [6] [8] [9] [12] [13]

and

cwl[DSV] or cil[DSV] for [10].

Note that position [4] of the form DSV was already locked by T1 when it started.

Positions [7] and [11] also of the form DSV may need to be locked at this point since their locking was delayed until they are needed to evaluate the s-definitions in which they appear.

The rest of the read locks deal with s-variables of the form M.Q.DSV and DSV⇓M.R. Those can be replaced by their values now that they are needed to evaluate the s-definitions. Note that when the procedures or probes that gave them their values returned, the scheme postponed their locking.

Not all those class-read-locks will actually be needed. That is because the actual execution path of the code may not have read the values of the the variables whose classes are represented by the listed s-variables.

To illustrate, we present an example. Consider the following piece of code, where dsv1 and dsv2 are dynamic state variables.

```
...
a = dsv1;
M.Q();
if (b == 0)  then
   a = dsv2;
M.P(a);
...
```

Then, the s-definition for the actual IN parameter a is

$$\underline{a} \equiv \oplus(\text{dsv1, b, dsv2}\Downarrow\text{M.Q, IMP}).$$

If b is not 0, then dsv2 actually does not flow into a. In that case, $\underline{\text{dsv2}}$ need not figure in in $\underline{a}$ to avoid overclassifying the actual IN parameter.

We can avoid the overclassification in certain cases by taking advantage of the locking system. The way it is done is for the run-time mechanism to inquire if the transaction already owns a vrl[dsv] for each dsv for which it is required to request crl[dsv]. If the response is negative then the run-time mechanism may remove the class of that variable from the s-expression it is trying to evaluate.

In the example, if it is determined that vrl[dsv2] is not already owned then the run-time mechanism does not request crl[dsv2] and the s-definition is evaluated as

$$\underline{a} \equiv \oplus(\text{dsv1, b, IMP}).$$

Note that crl[dsv1] is requested in all cases.

Not requesting crl[dsv2] when the procedure started executing leads to a higher degree of concurrency. Another advantage is that we avoid overclassification in some cases in which the class of a dynamic state variable appears in some s-expression.

The reason why not all overclassification cases can be resolved in the described way is illustrated by the following example.

```
...
c = dsv;
M.Q(c);
if (b == 0)  then
   a = dsv;
M.P(a);
...
```

In this example, when $\underline{a}$ is being computed, the transaction is always found to already own vrl[dsv]. That is because the lock was acquired for a previous operation. As a result, even if b is not 0, the run-time mechanism will go ahead and include $\underline{dsv}\Downarrow$M.Q in $\underline{a}$.

Another way for the transaction to be found to already own crl[dsv] is if it inherited it from a descendant transaction.

So, in general, the test whether a crl[dsv] is owned does not always accurately indicate if the value of dsv has actually been read by the transaction owning the lock. Only if the transaction does not own the lock can we deduce with certainty that it did not read the variable. So, the scheme only improves on the precision in evaluating s-expressions and does not completely eliminate overclassification.

To further improve on the situation, the locking system can keep with each lock

that is owned by a transaction information on how it was obtained. This way, when responding to inquiries about locks, it can inform the flow control mechanism if the lock is inherited. If the lock is indeed inherited then the evaluation of the s-expression can proceed without including the class in question.

The last type of requested locks to be discussed is cwl for position [10]. This is needed to update the dynamic state variables whose class may have changed before the message is sent out.

Again here, the lock may not be needed if the variable has not been actually unambiguously s-assigned. For example, if b is not 0 in the following code

```
...
if (b == 0)  then
   dsv = a;
M.P();
...
```

then dsv need not be updated before the RPC M.P is sent out. However, it still need to be incremented by b.

We can use a similar test as above to decrease the overclassification of dsv. Namely, an inquiry on whether the transaction already owns vwl[dsv] should be sent to the concurrency control component, first. If the response is positive than the only conclusion that can be drawn is that a write operation has already been performed at some point. So, cwl[dsv] must be requested.

Otherwise, if the transaction does not already own cwl[dsv], then a cil[dsv] is requested rather than a cwl[dsv]. This has the result not only to improve the precision in computing the classes of dynamic state variables, but also of improving on the

degree of concurrency since a cil[dsv] still allows other transactions to s-increment dsv unlike a cwl[dsv].

The scheme is not always accurate for the same reasons as the previous scheme involving crl's. Namely, if a vwl[dsv] is indeed already owned, one cannot make definite conclusions on when and how it was acquired.

Case 2: Outgoing Probe Messages:

As in the case of Outgoing RPCs, classes of output variables need to be sent out and some dynamic state variables need to be updated. The appropriate locks need to be requested and they, in general, are as follows.

crl[DSV] for [11]

and

crl[DSV] for [5] [6] [12] [13]

and

cwl[DSV] or cil[DSV] for [10].

It could be seen that the only difference from the previous case handling outgoing RPCs is that the locks for the actual input parameters are not needed. Otherwise, all the lock types and the locked positions in the template are the same.

The discussion in the outgoing RPC case also applies to outgoing probes and we will not repeat it here.

Case 3: Outgoing Return Messages:

In this case, the classes of output variables corresponding to the formal OUT parameters need to be computed. Also, the final classes, as modified by the returning procedure, of the dynamic state variables must be computed and the current classes of the corresponding variables must be updated.

crl[DSV] for [1] [15]

and

crl[DSV] for [2] [3] [16] [17]

and

cwl[DSV] or cil[DSV] for [14].

Again, the exact same argument as in the RPC case can be conducted, however, on different positions in the template.

Positions [1] and [15] are read locked now since they were not at the beginning of the procedure.

The classes of the variables in positions [2], [3], [16], and [17] are locked only if their values have already been locked.

Finally, a cwl type is requested for variables in position [14] only if a vwl has already been acquired for those variables. Otherwise, a cil type is instead requested.

5.3.1.3 Summary of lock usage To summarize, for each of the five lock types, we list the cases in which it is possibly requested.

Value Read Lock (vrl): Requested by the access component.

1. A user process attempting to read a state variable while executing a procedure in response to an RPC.

Value Write Lock (vwl): Requested by the access component.

1. A user process attempting to write a state variable while executing a procedure in response to an RPC.

Class Read Lock (crl): Requested by the flow control component.

1. An incoming RPC.
2. An incoming (RPC or probe) return.
3. An outgoing RPC if vrl was already acquired.
4. An outgoing Probe if vrl was already acquired.
5. An outgoing RPC-return if vrl was already acquired.

Class Write Lock (cwl): Requested by the flow control component.

1. An outgoing RPC only if vwl was already acquired.
2. An outgoing probe only if vwl was already acquired.
3. An outgoing RPC-return only if vwl was already acquired.

Class Increment Lock (cil): Requested by the flow control component.

1. An incoming probe.
2. An outgoing RPC in case cwl is not needed.
3. An outgoing probe in case cwl is not needed.
4. An outgoing RPC-return in case cwl is not needed.

5.3.2 The flow control mechanism revisited

The flow control mechanism of the previous chapter needs to be modified to take advantage of the the presence of the locking system.

5.3.2.1 Changes to the compile-time mechanism When evaluating s-expressions the run-time mechanism needs to be able to differentiate between s-variables that represent implicit flows and the ones that represent explicit flows. Then, when it requests a cil for a dynamic state variable instead of a cwl (in case a vwl is not owned already), it will be able to determine the class by which the variable must be s-incremented.

For example, consider the following code.

```
int ssv(SECRET);
int a, b, x, y(CONFIDENTIAL);
...
if (x + y  = 0) then
   dsv = a + b + ssv;
```

The s-definition (in its unsimplified form) for dsv is

$$\underline{dsv} \equiv \oplus(a, b, \text{SECRET}, x, \text{CONFIDENTIAL}, \text{IMP}),$$

where the first three components ($\underline{a}$, $\underline{b}$ and SECRET) of its s-expression correspond to explicit flows and the remaining components ($\underline{x}$, CONFIDENTIAL, and IMP) correspond to implicit flows.

Assuming that the value of (x+y) is not zero, when class of dsv is about to be updated, the run-time mechanism requests cil[dsv] rather than cwl[dsv] since vwl[dsv] is not owned by the transaction. Then, it needs to be able to determine that the security class by which the class of dsv needs to incremented is

$$\oplus(x, \text{CONFIDENTIAL}, \text{IMP}).$$

For that to be possible, the compile-time mechanism needs to modify the way the s-expression are represented to allow the separation of explicit flows from implicit flows. Whenever we need to differentiate between explicit flows and implicit flows in s-expressions, we will use the following new notation

$$\oplus(a, b, \text{SECRET } \# \text{ x}, \text{CONFIDENTIAL}, \text{IMP}).$$

The symbol '#' is used to separate the two types of flows.

Another change in the representation of s-expression is to have the constant security classes that correspond to static state variables tagged by the name of the variables. This allows the run-time mechanism to first check if the transaction owns a vrl on the static state variable appearing in the tag of a constant class before it includes it in the evaluation of the s-expression.

For example, in the following code

```
int ssv(SECRET);
...
if (a == 0) then
    dsv = ssv;
```

the class of ssv may be excluded from the computation of dsv if vrl[ssv] is not owned by the transaction. If the s-definition is represented as

$$\underline{\text{dsv}} \equiv \oplus(\text{SECRET } \# \text{ a}, \text{IMP}),$$

there is no way for the mechanism to recognize that the constant security class SECRET appears as a result of the flow from ssv to dsv.

However, using the following representation of s-expressions in the s-definition of dsv

$$\underline{\text{dsv}} \equiv \oplus(\text{SECRET}^{ssv} \# \text{ a, IMP}),$$

enables the run-time mechanism to recognize that the class SECRET corresponds to the state variable ssv. Then, it can inquire whether vrl[ssv] is owned by the transaction.

Subsequently, we will not use the new more complicated representations of s-expressions unless they are needed.

5.3.2.2 Changes to the run-time mechanism As explained, the run-time mechanism can take advantage of the presence of a locking mechanism to improve on the accuracy of evaluating s-expressions. This leads to improving the precision of the computed classes of various variables and implicit flows. In turn, this may lead to an improvement in the precision of the mechanism in rejecting insecure programs if the s-expressions in the s-checks become more accurate.

For incoming messages, the only change to the mechanism is listed below.

- When an RPC or an RPC-return are received, the only s-variables for dynamic state variables that are replaced by their current values are the ones that appear in s-expressions in the s-checks and in the implicit's under INS.

The changes to the mechanism that affect the steps taken to handle outgoing messages are as follows.

1. When an s-expression is being evaluated, for each s-variable, first inquire if a vrl on the corresponding variable has already been acquired. If not, then the s-variable is not included the computation.

2. When an s-expression is being evaluated, for each tagged constant security class, first inquire if a vrl on the variable in the tag has already been acquired. If not, then the tagged constant security class is not included the computation.

3. If an s-variable is to be included in an s-expression, then replace it with its current value (if one exists).

4. Whenever the class of a dynamic state variable needs to be updated, first inquire that a vwl on the variable has already been acquired. If not, then only the implicit flow part of the s-expression need to be evaluated and the result is used to s-increment the state variable.

5.3.3 Lock compatibility

We need to produce a lock compatibility table for the five lock types. First, we analyze all cases in which one lock type on a variable is held by one transaction while another transaction requests a lock on the same variable. The analysis explains whether the request should be granted or blocked. Then, we will be able to produce the lock compatibility table.

To study the different cases, we assume that a transaction T1 holds a certain lock on a variable dsv and that another different transaction T2 is requesting a lock on the same variable.

5.3.3.1 Case 1: T1 holds vrl[dsv]:

T2 requests vrl[dsv]: GRANTED.

This case is simple; it follows the usual locking rules. Both transactions are reading the value of the variable; they do not affect each other.

T2 requests vwl[dsv]: WAIT.

Again, this case must follow the usual rules: the requested lock conflicts with an already acquired lock.

T2 requests crl[dsv]: GRANTED.

The two operations, vr[dsv] and cr[dsv], commute: the effect is independent of the order in which they execute.

T2 requests cwl[dsv]: N/A.

This case is not possible in our scheme. If T1 owns vrl[dsv], then T2 definitely does not own vwl[dsv] because the two locks are not compatible. In this case, T2 requests cil[dsv] instead since it was determined that T2 did not cause an explicit flow to dsv.

T2 requests cil[dsv]: GRANTED.

In this case, T1 has read the value of dsv and now T2 is trying to s-increment it. We can assume that T1 does not own crl[dsv] (this case is handled later). Then, none of the template instance computations for T1 involved the class of dsv. As a consequence, T2 should be allowed to s-increment dsv. This of course, will cause T1 to block later when it eventually requests crl[dsv].

5.3.3.2 Case 2: T1 holds vwl[dsv]:

T2 requests vwl[dsv]: WAIT.

Obviously, the request cannot be granted and T2 must be blocked.

T2 requests crl[dsv]: WAIT.

Here, T1 had caused the value of dsv to change but the run-time mechanism

did not have the chance to update the class of dsv accordingly. We can assume that T2 does not hold vrl[dsv] because it would conflict with T1's lock. Then, if T2 is handling an outgoing message, our scheme states that T2 does not really need to read the class of dsv.

If T2 is an incoming message, then allowing T2 to read the class of dsv would cause the flow checks to be based on a class of dsv that does not accurately reflect its value. That might lead to an undetected flow violation; the request must not be granted and T2 is blocked.

T2 requests cwl[dsv]: N/A.

T2 does not own vwl[dsv] so instead it requests cil[dsv].

T2 requests cil[dsv]: GRANTED.

By granting this request, we force T1 to wait when it requests cwl[dsv] until T2 commits and releases the granted lock. When T1 resumes, it eventually acquires cwl[dsv] and updates the class of dsv.

5.3.3.3 Case 3: T1 holds crl[dsv]:

T2 requests crl[dsv]: GRANTED.

Two read locks never conflict.

T2 requests cwl[dsv]: N/A.

T2 would not make this request unless it already owns vwl[dsv]. However, vwl[dsv] and crl[dsv] are conflicting locks and cannot be owned by both T1 and T2. Therefore, this case is not possible.

T2 requests cil[dsv]: WAIT.

This a classical case of a conflict between the operations of reading a value and modifying a value. The request may not be granted.

5.3.3.4 Case 4: T1 holds cwl[dsv]:

T2 requests cwl[dsv]: N/A.

T1 must already own vwl[dsv] so T2 must not have that lock and this type of request is not possible.

T2 requests cil[dsv]: WAIT.

This a simple case of conflict between two modification attempts. T2 must be blocked.

5.3.3.5 Case 5: T1 holds cil[dsv]:

T2 requests cil[dsv]: GRANTED.

As we have seen, two increment operations commute and the end result does not depend on which of the two transactions goes first. Two class increment transactions should be allowed to execute concurrently.

The omitted cases can be analyzed by arguments deduced by symmetry of the above cases. So, at this point, we are ready to produce the compatibility table. It is shown in Figure 5.4.

The compatibility table shows that in most cases while one transaction may be accessing the value of a dynamic state variable, another transaction may access its class. Most importantly, probes are allowed to access the same variable concurrently since cil's can be held by different transactions.

	vrl	vwl	crl	cwl	cil
vrl	yes	no	yes	n/a	yes
vwl	no	no	no	n/a	yes
crl	yes	no	yes	n/a	no
cwl	n/a	n/a	n/a	n/a	no
cil	yes	yes	no	no	yes

Figure 5.4: Compatibility matrix for class and value locks

5.4 Reducing the Number of Lock Types

With the 5-type lock mechanism, more different types of operations may run concurrently. However, the next concern is of the overhead created from managing five lock types. Does the gain in concurrency justify the inefficiency introduced by this overhead? In this section, we argue that the answer to that question is no. Then, we show which lock types can be combined.

The class-increment-lock type is shown in our initial locking mechanism to be very useful in two ways. It allows different probes to run concurrently. Also, it is a substitute to the class-write-lock type when a transaction is determined not to have modified the value of a variable (that is because in that case only the implicit flows occur). So, the cil lock type is necessary for our system and must not be eliminated.

Similarly, the two value locks are obviously needed when user processes in the access component read and write state variables. Therefore, the only locks whose usefulness is in question and needs to be analyzed are the remaining class lock types, the class-write-locks and the class-read-locks. We argue that they can be combined with value-write-locks and value-read-locks, respectively.

5.4.1 Combining the write-lock types

A cwl is requested only by outgoing messages to update the class of a state variable. However, the value of the variable must have been modified also. The same transaction requesting a cwl must have already acquired a vwl. So, an acquisition of a vwl type is always followed by a request for a cwl type on the same variable. That is why all but one of the entries of the row of the cwl type in the compatibility table are 'n/a'. As a result, it seems possible to combine the cwl and vwl types into one write-lock type.

However, the only remaining entry in the cwl row which is not 'n/a' is 'no'. Whereas, the corresponding entry in the row of the vwl type is 'yes'. That says that having two separate write locks allows more concurrency since the cil type is compatible with one of those locks, namely the vwl type.

For example, assume a transaction T1 acquires vwl[dsv]. The only increase in concurrency that can be gained from having a separate cwl type is by allowing other transactions to acquire cil[dsv] before T1 eventually requests cwl[dsv] and while it is performing other operations. The question is how long can the gap be between the time when T1 acquires vwl[dsv] and the time it requests cwl[dsv]? The request for cwl[dsv] is made when any message is sent out. So, the gap is equal to the time it takes for the next message sending request to be made.

If we assume that message sending (RPCs, returns or probes) is frequent in most procedures and that operations not involving message passing are considerably less time-consuming[3], then we can conclude that the gap must actually be short and that

[3]In certain systems, it may be found that these assumptions are not valid. In such systems, most procedures contain major time-consuming local computations

the gain in concurrency is small. As a result, we can eliminate class write locks and actually allow the write-locking of the class and the value of a variable using one lock type to be requested when the variable is being written into in the access components of the RMs.

5.4.2 Combining the read-lock types

We also argue that crl's and vrl's can be combined into one lock type without incurring a significant loss in concurrency. A similar argument as above can be made. When a crl on a variable is requested by an outgoing message, it must have already acquired the vrl on the same variable. The elapsed time between the request for the class lock and the acquisition of the value lock may be assumed not to be very long.

Again, here, the only additional concurrency gained from separating crl and vrl types is by allowing a cil to be acquired by one transaction while a vrl is held by another transaction. If we assume that this gain is not very significant, then in the case when vrl's are requested before crl's (when messages are outgoing), we can say that the crl type is not needed.

However, crl's are also requested by incoming messages (RPCs and returns). So, a crl request is not always preceded by a vrl acquisition. In certain cases, a crl may be requested before a vrl. However, from the entries in the compatibility table, it can be seen that crl's are a "stronger" lock type than vrl's are. In other words, by holding a crl, a transaction can also be considered as holding the corresponding vrl. That is because holding a crl prevents other transactions from accessing the variable in the same cases as holding a vrl. This can be seen since there is a 'no' entry in the

not involving external procedure calls. Then, the system designers may decide to keep the 5-type locking scheme to increase concurrency.

	rl	wl	il
rl	yes	no	no
wl	no	no	no
il	no	no	yes

Figure 5.5: Compatibility matrix for class and value locks

row of the crl type at each column where a 'no' exists in the row of the vrl type. We say vrl can be *converted* into a crl [5].

In summary, we showed that in the case where the crl is acquired first it is as if the vrl is also acquired. In the case the vrl is requested first, we can argue as for vwl to eliminate crl since the gain in concurrency is not great. We conclude that in both cases, the crl type and the vrl type can be combined. Whenever either the class or the value of a state variable needs to be read, both are locked. Again, no significant loss of concurrency occurs.

5.4.3 The 3-type lock scheme

We can settle on using three lock types for our locking mechanism: *a read lock* (rl), *a write lock* (wl), and an *increment lock* (il). This is the result of the realization that decoupling the class and value gained minimal concurrency in the case of reading and writing. However, the increment lock type is absolutely needed to permit greater concurrency with the potentially expensive probes.

The 3-type lock mechanism has the compatibility table that is shown in Figure 5.5.

5.4.4 Usage of the three lock types

The new lock usage rules can be deduced from the lock usage rules for the five lock types. We explain the new rules here.

Read Lock The read lock type is requested by both access and flow control components. It is requested if a user process attempts to read the value of a variable and if the information flow control mechanism attempts to replace an s-variable in a template.

To still take advantage of the locking mechanism to improve on the precision of computing classes of dynamic state variables, the concurrency control mechanism must keep a new flag with each held read lock. The flag should indicate whether the read lock is held as a result of a request from the flow control mechanism rather than from the access component.

To use the flag, when a message is being sent, the flow control mechanism can inquire if a read lock is already held. As usual, if the answer is negative then the s-variable for which the lock is being requested can be omitted from the computation of the s-expression.

On the other hand, if the answer is positive, then the new flag is checked. If the flag indicates that the access component is not responsible for the acquisition of the lock, then it can again be deduced that the variable has not been read in and that its s-variable can be omitted. Otherwise, the s-variable must be included in the computations.

Increment Lock This lock is exclusively requested by the flow control mechanism when it needs to s-increment a dynamic state variable. S-increments are per-

formed as a result of an incoming probe. Alternatively, they are performed when an outgoing message is being sent and it is determined that the transaction does not already hold the write lock.

Write Lock The write lock type is only requested by the access component on behalf of user processes that need to write a variable. The flow control component does not use this lock since if it does not already own it, it requests the increment lock.

5.4.5 Locking and recovery rules for nested transactions

The following locking rules extend or modify Moss' locking rules for nested transactions to include the new type of lock.

1. A transaction may obtain a read lock on an item only if all holders of write or increment locks (if any) on the item are its ancestors.

2. A transaction may obtain a write lock on an item only if all holders of any (read, write or increment) locks (if any) on the item are its ancestors.

3. A transaction may obtain an increment lock on an item only if all holders of any read or write locks are its ancestors.

The recovery rules for the three lock scheme are as follows. They are simple extensions of the rules stated in Chapter 3.

1. When a transaction obtains a write lock on an item, a version of the item is made. All updates to the item are made to this shadow version.

2. When a transaction obtains an increment lock on an item, a version of the item is made. All class increments to the item are made to this shadow version.

3. When a transaction commits, the shadow version is inherited by its parent. If the parent already has its own version of the same item, the child's version takes precedence. When the top-level transaction commits, the shadow versions it holds are installed on stable storage.

4. When a transaction fails, its shadow versions are discarded.

6. PRECISE CALCULATIONS OF CLASSES OF DYNAMIC VARIABLES

In the previous chapter, we took advantage of the presence of a locking mechanism to increase the precision of the class computing mechanism. We also argued that that increase in precision leads to an increase in the precision of the information flow control mechanism. Since locks are only requested for state variables, computing the classes of dynamic local variables is not affected and still involves significant overclassifications. That in turn leads to overclassifying dynamic state variables.

For example, consider the following code in which dsv is a dynamic state variable and dv is a dynamic local variable.

```
dv = exp1;
if (a == 0) then
   dv = exp2;
dsv = dv;
```

If dv is overclassified as

$$\oplus(\text{exp1}, \text{exp2}, \text{a}, \text{IMP})$$

then dsv will be overclassified also. The locking mechanism cannot help in computing a more precise class for dv.

In this chapter, we present a new class computing mechanism for updating classes of dynamic state variables. The new mechanism can be used to precisely compute classes of any dynamic variables. Our presentation will deal with dynamic state variables. However, it should be remembered that the scheme applies to local variables also.

6.1 Introduction

The way the class of a dynamic state variable is updated is by evaluating an s-definition and the result is used to update the current class of the variable. In the original class updating mechanism of Chapter 4, an s-definition represents the upper bound on the value of the class that a variable can reach by executing the code of a procedure. An s-definition does not take into consideration conditional executions in which the actual update to the variable may not occur. As a result, s-definitions may not reflect the actual class of the information contained in a variable. The class updating mechanism is imprecise and overclassifies dynamic state variables.

In theory, through the continuous updating and incrementing of the classes of dynamic variables, a system may reach a stage in which the encapsulated information inside objects is all classified at the highest class (e.g., TOPSECRET). At this point, we know of no way that automatically declassifies information. The process of bringing down the classes of information can only be done by a trusted information manager. The larger the system is the more costly and slow this process becomes.

The imprecision of the class updating mechanism which overclassifies state variables worsens the above problem by unnecessarily accelerating the upward move of the classes of state variables. We have developed a new form for s-definitions which

takes into account the actual execution path taken within processes. By doing that, the new s-definitions can be used to compute a precise class to be assigned to dynamic variables. The precise class reflects the actual classification of the information contained in the variables[1].

Before we present the details of the new scheme, we illustrate the problem and our solution with a simple example. Assume a procedure contains the following code:

```
dsv = exp1;
if (E1)
        dsv = exp2;
```

where E_1 is a boolean expression, exp_1 and exp_2 are expressions of the appropriate type and dsv represents a dynamic state variable to which the procedure has access.

Upon completion of the procedure, the class of dsv must be updated. In the original mechanism, the new class is determined by evaluating the following s-definition:

$$\underline{dsv} \equiv \oplus(exp_1, exp_2, E_1, IMP).$$

The possible overclassification is obvious since in no way can both expressions, exp_1 and exp_2, affect the class of dsv at the same time.

Unlike s-definitions in their new form, the above s-definition does not allow the class updating mechanism to consider whether at execution time E_1 evaluates to true or false. For the above example code, the new s-definition is generated from a flow graph that is built at compile time. The nodes of the flow graph are basic blocks of code which have one entry point and one exit point.

[1]It should be remembered that our scheme updates classes of dynamic variables only at message passing time. This makes the updating mechanism more efficient than a pure run-time approach which performs the updates after each s-assignment.

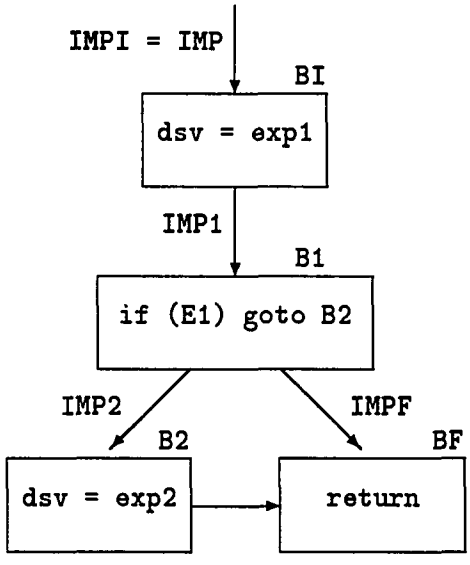

Figure 6.1: A flow graph

The flow graph is shown in Fig 6.1. It shows four basic blocks: B_I (the initial basic block), B_1, B_2, B_F (the final basic block). The arcs show the possible flows of control. Each arc is labeled by the class of the implicit flow into the basic block. For example, the class of the implicit flow into basic block B_2 is represented by IMP_2 and is equal to $\oplus$ (E_1, IMP).

The new s-definition for dsv generated from the example flow graph is:

$$\underline{dsv} \equiv B_2[\oplus(exp_2, E_1, IMP)] \vee \neg B_2[\oplus(exp_0, E_1, IMP)].$$

This s-definition can be evaluated as follows. If basic block B_2 is entered at run time then the value of the s-definition is the value of the s-expression enclosed between the square brackets that are marked by B_2, $\oplus(exp_2, E_1, IMP)$; otherwise, the computed class is the s-expression marked by $\neg B_2$, $\oplus(exp_0, E_1, IMP)$. The overclassification is avoided since only one of the s-variables $\underline{exp_0}$ or $\underline{exp_1}$ is included in computing the

new dsv.

In the next sections, we present the details of the new scheme. We start by defining basic blocks and flow graphs and presenting their relevant properties. The basic definitions and properties are all adapted from Chapters 9 and 10 in [1].

6.2 Basic Blocks and Flow Graphs in the PRM System

6.2.1 Procedure syntax

We assume the following statement syntax for the procedures in our PRM model.

$$
\begin{aligned}
S \rightarrow\ & \textbf{id} = E \mid RPC \mid probe \mid S\ ;\ S \mid \\
& \textbf{if (E) then } S \mid \\
& \textbf{if (E) then S else } S \mid \\
& \textbf{while (E) do } S
\end{aligned}
$$

$$
E \rightarrow \textbf{id} + \textbf{id} \mid \textbf{id}
$$

RPC and probe stand for a remote procedure call request and a send-probe instruction, respectively. We have chosen a simplified form of expressions where the operator '+' is representative of all other operators.

6.2.2 Basic blocks

The code of a procedure is divided into basic blocks. A *basic block*, denoted B_i, is a sequence of statements with one entry point (the beginning of the block) and one

exit point (the end of the block). Within a basic block, flow of control does not halt or branch except at the end of the block [2].

For a procedure, We denote its first basic block by B_I and its last basic block by B_F. We assume that B_F of any procedure consists of a return statement.

RPC and probe statements are assumed to form their own basic blocks that contain no other statements.

For example, the following code is divided into two basic blocks, B_1 and B_2.

```
B1 : dsv1 = exp1;
     b = ssv;
     ssv = dsv1;
B2 : M.P(a | dsv2);
```

For each basic block B_i, we associate a security class, IMP_i, which represents the class of the implicit flow into B_i. In the above example, IMP_1 flows into B_1, IMP_2 flows into B_2 and IMP_1 and IMP_2 represent the same security class.

In our discussions, basic blocks are represented in either of two ways, as shown in Figure 6.2 which shows block B_1 of the example. The difference between the two representations is that the one on the right does not show the code of the basic block. The arcs going into the basic blocks are labeled by the implicit flow.

The code in a basic block may s-assign dynamic variables that appear as receiving flow in it. For a basic block B_i and a dynamic state variable dsv, $\underline{dsv}_{\uparrow i}$ and $\underline{dsv}_{\downarrow i}$ are s-variables used to denote the classes of dsv before control enters B_i and after it exits B_i, respectively. For example, $\underline{dsv}_{\uparrow I}$ is the class of dsv prior to the execution of

[2] Loops are not treated until flow graphs are introduced later. Until then, we will assume that the basic blocks are not included within loops.

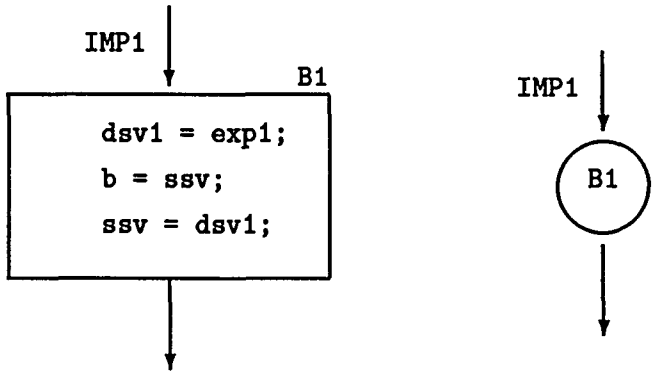

Figure 6.2: Representations of basic blocks

the first basic block of a procedure, i.e., before a procedure is executed. Also, $\underline{\text{dsv}}_{\downarrow F}$ represents the class of dsv upon the completion of a procedure.

Going back to the basic blocks B_1 and B_2 of the above example, assume that dsv1 is LOW and that dsv2 is SECRET immediately prior to the beginning of B_1. Then, we have the following s-variables with their values.

$$\underline{\text{dsv1}}_{\uparrow 1} \equiv \oplus(\text{LOW})$$
$$\underline{\text{dsv1}}_{\downarrow 1} \equiv \oplus(\text{exp}_1, \text{IMP}_1)$$
$$\underline{\text{dsv2}}_{\uparrow 1} \equiv \oplus(\text{SECRET})$$
$$\underline{\text{dsv2}}_{\downarrow 1} \equiv \oplus(\text{dsv2}_{\uparrow 1})$$
$$\underline{\text{dsv1}}_{\uparrow 2} \equiv \oplus(\text{dsv1}_{\downarrow 1})$$
$$\underline{\text{dsv1}}_{\downarrow 2} \equiv \oplus(\text{dsv1}\Downarrow\text{M.P})$$
$$\underline{\text{dsv2}}_{\uparrow 2} \equiv \oplus(\text{dsv2}_{\downarrow 1})$$
$$\underline{\text{dsv2}}_{\downarrow 2} \equiv \oplus(\text{M.P.dsv2})$$

We define *dsv basic blocks* as basic blocks whose codes s-assign dsv. In general, for each dynamic state variable dsv, we can divide dsv basic blocks into two cate-

gories: ones that contain ambiguous dsv s-assignments and ones that only contain unambiguous dsv s-assignments. The basic blocks in the first category will be called *ambiguous dsv basic blocks.* Similarly, the second category basic blocks will be called *unambiguous dsv basic blocks.*

For a dynamic state variable dsv, an ambiguous dsv basic block B_i can be in one of two types. If the B_i code consists of a probe statement, then we will call it a *probe ambiguous dsv basic block.* Otherwise, if it consists of an RPC statement in which dsv does not appear as actual OUT parameter, then it will be called a *call ambiguous dsv basic block.*

There are also two types of unambiguous dsv basic blocks. A *call unambiguous dsv basic block* is one that consists of an RPC in which dsv does appear as an actual OUT parameter. An *assignment unambiguous dsv basic block* is one whose code shows dsv on the L.H.S. of at least one assignment statement.

If B_i is an assignment unambiguous dsv basic block, we will denote by $\underline{\text{dsv-exp}_i}$ the the class of the expression that appears on the R.H.S. of the last assignment statement in B_i that s-assigns dsv. For example, if the following code forms one basic block B_1

```
B1: dsv = x;
    ...
    dsv = y;
    ...
    dsv = z;
    ...
```

then $\underline{\text{dsv-exp}_1}$ will be used to denote $\underline{z}$ in B_1.

In Figure 6.3, we show examples of the different types of basic blocks that s-assign a dynamic state variable.

For a dynamic state variable dsv, the s-variable $\underline{\text{dsv}}_{\downarrow i}$ has a different value depending on the type of the basic block B_i. If B_i is not a dsv basic block, then $\underline{\text{dsv}}_{\downarrow i}$ is defined as

$$\underline{\text{dsv}}_{\downarrow i} \equiv B_i[\oplus(\underline{\text{dsv}}_{\uparrow i})].$$

If B_i is a call unambiguous dsv basic block consisting of an RPC to M.P, then we have

$$\underline{\text{dsv}}_{\downarrow i} \equiv B_i[\oplus(\text{M.P.dsv}, \text{IMP}_i)].$$

If B_i is an assignment unambiguous dsv basic block, then we have

$$\underline{\text{dsv}}_{\downarrow i} \equiv B_i[\oplus(\text{dsv-exp}_i, \text{IMP}_i)].$$

If B_i is any (call or probe) ambiguous dsv basic block consisting of a probe or an RPC to M.P, then we have

$$\underline{\text{dsv}}_{\downarrow i} \equiv B_i[\oplus(\underline{\text{dsv}}\Downarrow\text{M.P})].$$

The use of the notation $B_i[\cdots]$ in the above indicates that the included s-expression is to be evaluated only if B_i is entered at run time. Since it is the information flow control component of an RM that evaluates s-definitions, it needs for the access component to communicate to it at message passing time whether each basic block is executed or not. This can be accomplished if we assume the existence of a flag for each basic block in a procedure. This flag should be set by the access component if control enters its basic block. For a basic block B_i, the flag will be also

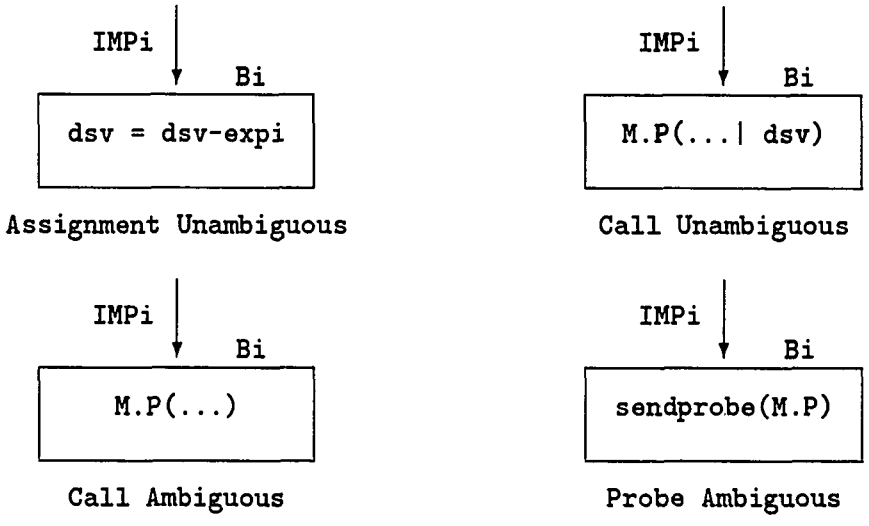

Figure 6.3: The four types of dsv basic blocks

denoted by B_i[3]; context will allow for the differentiation between the two notations. For any basic block B_i, we will assume that its first statement is as follows.

```
Bi: Set Flag Bi
    < Bi code >
    ...
```

6.2.3 Flow graphs

6.2.3.1 Definitions A *(control) flow graph* of a procedure is a directed graph whose nodes are the basic blocks of the procedure and whose arcs represent the execution paths traced by all possible flows of control. The initial node of a flow graph is always B_1, which is the entry point to the procedure.

[3]A one-bit flag is enough only if the basic block is not contained in a loop. Otherwise, more bits are needed as we will see later.

An edge in the graph from B_i to B_j is denoted by $B_i \rightarrow B_j$, and B_i is called its *tail* and B_j is its *head.*

If there is a path in the flow graph from B_i to B_j, we say that B_i is a *predecessor* of B_j and that B_j is a *successor* of B_i.

A node B_d *dominates* a node B_i in the flow graph if every path from B_I to B_i goes through B_d.

A *back edge* is an edge in the flow graph whose head dominates its tail.

6.2.3.2 Regions of flow graphs A *region S* is a portion of a flow graph that corresponds to a (compound) statement (as defined in the given syntax), S, in the code of the procedure. The nodes of a region S1 include a *header* basic block, denoted $B_{I_{S1}}$, which dominates all the other nodes in the region. When control enters a region, it flows to its header. When control leaves a region, it can flow to just one block.

For regions, we will use similar notations as we used for basic blocks. For example, IMP_{Si} denotes the the implicit flow into the region Si. IMP_{Si} is defined as $IMP_{I_{Si}}$, the implicit flow into the header of Si. As another example, the s-variable $\underline{dsv}_{\downarrow Si}$ denotes the class of dsv after exiting the region Si.

6.2.3.3 Building flow graphs For our assumed procedure syntax, the flow graph for each of the statements can be simply generated. In Figure 6.4, we show the flow graphs associated with the statements[4]. S1 and S2 represent regions in the

[4]The flow graphs in Figure 6.4 are shown using more primitive control structures than our assumed procedure syntax. Namely, the flow of control is shown to be directed through conditional branches (goto statements) which form basic blocks. This was done to simplify the graphical representation of basic blocks. However,

flow graphs which may be composed of one or more basic blocks.

As shown in the flow graphs, we associate one more expression with a basic block B_i that contains a conditional branch. We denote by E_i the boolean expression whose value determines the successor of B_i in the flow graph. The classes of the E_i's will figure in the computation of the implicit flows into basic blocks. $\underline{E_i}$ is assumed to be LOW if E_i does not exist, i.e., the basic block does not contain any conditional jumps.

6.2.3.4 Loops in flow graphs The assumed procedure syntax uses structured control statements. As a result, the generated flow graphs have the property that loops that occur in them are easily identifiable.

To find all loops, we start by identifying the back edges in a flow graph. Each back edge $B_i \rightarrow B_d$ gives rise to a set of nodes forming a loop defined as B_d plus all nodes that can reach B_i without going through B_d. B_d is called the *header* of the loop and it dominates all nodes in the loop.

6.2.3.5 An example flow graph In Figure 6.5, we show a complete flow graph for a procedure. We use this example flow graph to illustrate the definitions and notions we presented above. Note that we have labeled each E_i by the type of control structure to which it corresponds.

As an example of a loop, consider the back edge $B_{12} \rightarrow B_4$ in which its head, B_4, dominates its tail, B_{12}. This back edge determines the loop with header B_4 consisting of the nodes B_4, B_5 through B_9, and B_{10} through B_{12}.

these control structures in the flow graphs are still consistent with and equivalent to the procedure syntax we presented.

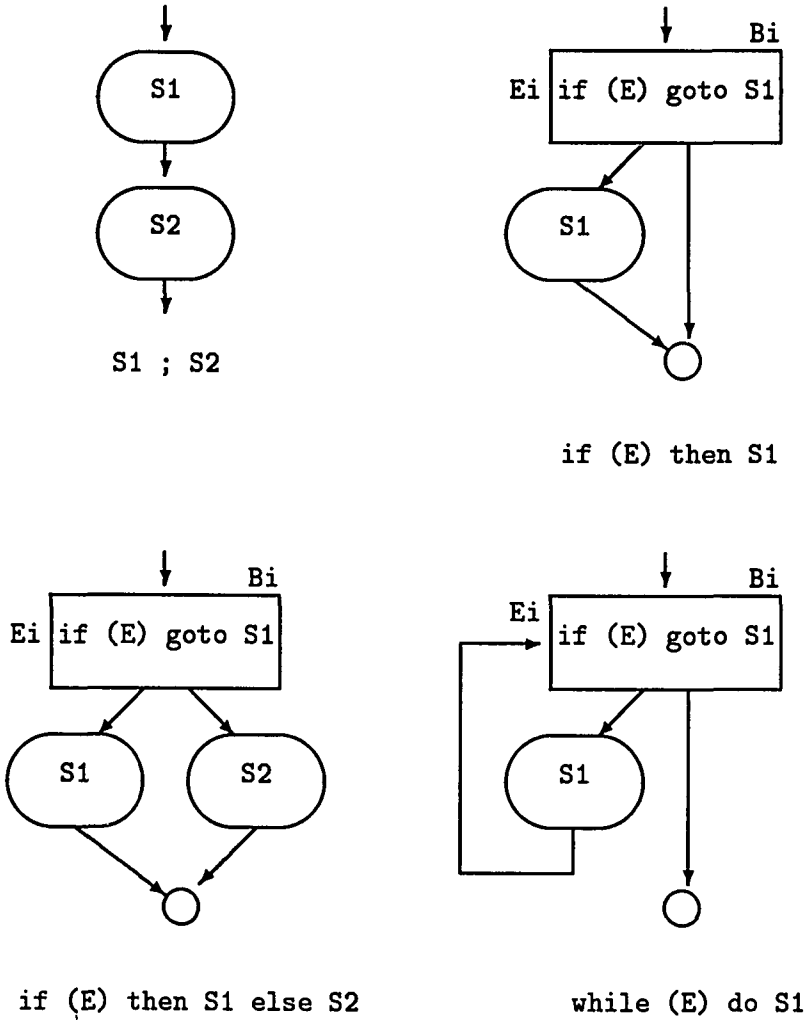

Figure 6.4: Flow graphs of the various statements

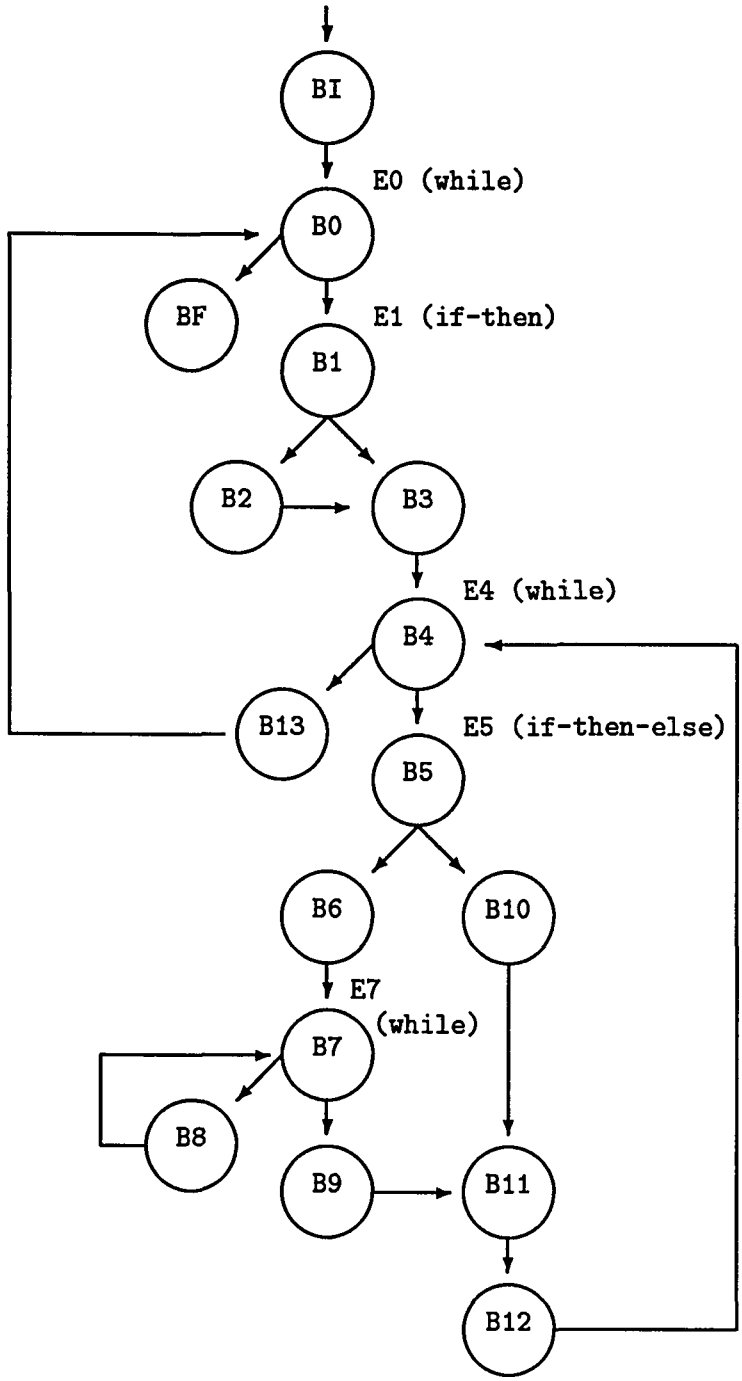

Figure 6.5: An example flow graph

The other loops in the graph have headers B_I and B_7 and correspond to the back edges $B_8 \rightarrow B_7$ and $B_{13} \rightarrow B_I$. Note that the boolean expressions E_I, B_4, and B_7 are associated with the nodes corresponding to headers of loops.

Below, we show the possible control structures that correspond to the example flow graph in Figure 6.5. We divide the code into the basic blocks that are suggested by the flow graph.

```
BI: ...
B0: while (E0) do
     B1: if (E1) then
          B2: ...
     B3: ...
     B4: while (E4) do
          B5: if (E5) then
               B6: ...
               B7: while (E7) do
                    B8: ...
               B9: ...
           else
               B10: ...
          B11: ...
          B12: ...
     B13: ...
BF: return
```

The basic blocks that are left unspecified contain no conditional branches. For in-

stance, B_2, B_{12}, and B_{13} could consist of the following pieces of code.

```
B2:  M.P(a, b | dsv);

B12: a = b;
     dsv = c - a;
     goto B4;

B13: goto B0;
```

As an example of a region, consider the region with header B_1. It includes basic blocks B_1, B_2, and B_3 and corresponds to an if-then-statement as can be seen from the edges between those basic blocks. Exiting this region can be only through B_4.

As another example, B_4 through B_{12} form a region corresponding to a while-statement. The only way into this region is B_4 and thus, its header is B_4. Control must leave the region through B_{13}. Note also that B_4 dominates all the blocks in the region.

6.2.4 Summary of notations

In Figure 6.6, we show a summary of the notations we have introduced.

6.3 Determining Implicit Flows into Basic Blocks

First, we need to explain how to compute the classes of the implicit flows into individual basic blocks of a given flow graph of a procedure. For any basic block B_i in the flow graph, IMP_i is defined as the least upper bound of the security classes that

B_i	Basic block number i or Flag indicating if control reached basic block B_i
Si	Statement number i or Region in the flow graph corresponding to statement Si
B_I	Initial basic block in a flow graph
$B_{I_{Si}}$	Initial basic block in region Si
B_F	Final basic block (return) in a flow graph
IMP_i	Implicit flow into basic block B_i
IMP_{Si}	Implicit flow into region Si; same as $IMP_{I_{Si}}$
E_i	Condition associated with basic block B_i that determines the flow of control
$\underline{dsv}_{\uparrow i}$	Class of variable dsv upon entering basic block B_i
$\underline{dsv}_{\uparrow Si}$	Class of variable dsv upon entering region Si
$\underline{dsv}_{\downarrow i}$	Class of variable dsv upon exiting basic block B_i
$\underline{dsv}_{\downarrow Si}$	Class of variable dsv upon exiting region Si
dsv-exp_i	Expression assigned to dsv by assignment unambiguous dsv basic block B_i
$B_i[\oplus(\langle$s-expr$\rangle)]$	Enclosed s-expression is relevant only if flag B_i is true
$\neg B_i[\oplus(\langle$s-expr$\rangle)]$	Enclosed s-expression is relevant only if flag B_i is false

Figure 6.6: Summary of flow graph notations

are computed as explained in the following rules. The rules are meant to respectively handle the propagation of local and global implicit flows within a procedure (as explained in Chapter 2), the propagation of implicit flows among successive basic blocks and the propagation of interprocedural implicit flows.

1. Include in IMP_i all $\underline{E_k}$'s, such that E_k corresponds to the conditional expression of an if-then-statement and B_i is the immediate leftmost child (corresponding to the body of the statement) of B_k.

2. Include in IMP_i all $\underline{E_l}$'s, such that E_l corresponds to the conditional expression of an if-then-else-statement and B_i is an immediate child of B_l.

3. Include in IMP_i all $\underline{E_m}$'s, such that E_m corresponds to the conditional expression of a while-loop and B_i is a successor of B_m.

4. Include in IMP_i all IMP_j's, such that B_j is an immediate parent and a dominator of B_i.

5. Include in IMP_i the class of the incoming implicit flow sent by the caller of the procedure, IMP.

The first rule takes care of the case of an if-then-statement. In this case, the class of the condition in the statement forms a local implicit flow into the first basic block of the body of the if-then-statement, i.e., the leftmost child of the basic block that contains the conditional jump. (The fourth rule handles propagating this flow beyond the first block of the statement body.) For instance, in the example flow graph of Figure 6.5, IMP_2 includes $\underline{E_1}$.

The second rule is similar to the first rule except that it deals with an if-then-else-statement. In this case, the class of the condition must flow into the first basic blocks of both bodies of code: the else-part and the then-part. In the example, $\underline{E_5}$ must figure in the computation of both IMP_6 and IMP_7.

The third rule handles including global implicit flows from the conditions of while-statements. Since the classes of those conditions implicitly flow into each statement that logically follow the while-loop (see Chapter 2), they must be included into any IMP_i of any B_i that can be reached from the basic block that tests the conditions. For example, $\underline{E_0}$ must figure in the computation of all IMP_i's in the flow graph of Figure 6.5 except IMP_I, which cannot be reached from B_0. Similarly, $\underline{E_4}$ and $\underline{E_7}$ flow into all basic blocks except B_I.

The fourth rule in the computation of IMP_i includes the class of an implicit flow going into B_i as a result of it first going into a predecessor basic block, B_j, and of not having any intermediate blocks between B_j and B_i that could increment or change that implicit flow. In this case, control unconditionally flows from B_j to B_i and the implicit flow must be propagated from B_j to B_i. However, B_j must dominate B_i for IMP_j to be included in IMP_i. Otherwise, the implicit flows into basic blocks that follow conditional statements would be wrongly overclassified.

For instance, in the example flow graph, IMP_1 must be included in IMP_2 and IMP_3. However, IMP_2 should not be included in IMP_3 since to reach B_3, control can flow through B_1 while skipping B_2. Similarly, IMP_9 and IMP_{10} should figure in IMP_{11} since B_9 and B_{10} do not dominate B_{12}.

Finally, the last rule for the computation of a basic block implicit flow includes the implicit flow that is received from external procedures.

Below, we apply the above rules to list the values of the basic block implicit flows in the example flow graph.

$$\begin{aligned}
IMP_I &\equiv \oplus(IMP) \\
IMP_0 &\equiv \oplus(E_0, E_4, E_7, IMP_I, IMP) \\
IMP_1 &\equiv \oplus(E_0, E_4, E_7, IMP_0, IMP) \\
IMP_2 &\equiv \oplus(E_0, E_4, E_7, E_1, IMP_1, IMP) \\
IMP_3 &\equiv \oplus(E_0, E_4, E_7, IMP_1, IMP) \\
IMP_4 &\equiv \oplus(E_0, E_4, E_7, IMP_3, IMP) \\
IMP_5 &\equiv \oplus(E_0, E_4, E_7, IMP_4, IMP) \\
IMP_6 &\equiv \oplus(E_0, E_4, E_7, E_5, IMP_5, IMP) \\
IMP_7 &\equiv \oplus(E_0, E_4, E_7, IMP_6, IMP) \\
IMP_8 &\equiv \oplus(E_0, E_4, E_7, IMP_7, IMP) \\
IMP_9 &\equiv \oplus(E_0, E_4, E_7, IMP_7, IMP) \\
IMP_{10} &\equiv \oplus(E_0, E_4, E_7, E_5, IMP_5, IMP) \\
IMP_{11} &\equiv \oplus(E_0, E_4, E_7, IMP)
\end{aligned}$$

Note that there is some redundancy in the computation of the implicit flows; some security classes get to be included more than once. For example, in IMP_0, IMP is included twice since IMP_I is equal to IMP. However, this is does not introduce any overclassification since $\oplus$ is idempotent.

Also, note how $\underline{E_5}$ is propagated to IMP_7 by including IMP_6 in IMP_7 as stated in rule 4. However, $\underline{E_5}$ is not propagated to IMP_{11} since neither IMP_9 nor IMP_{10} are included in IMP_{11}.

6.4 Generating Flow Graph S-Definitions

In this section, we present the new s-definitions with their new forms which allow taking execution paths of procedures into consideration. The new s-definitions will be called *flow graph s-definitions.*

We have already shown (in Section 6.2) the flow graph s-definitions for dsv basic blocks. The algorithm for generating flow graph s-definitions for a complete procedure is presented by explaining how they are generated in case of each of the control structures in our procedure syntax.

The rules for generating the new s-definitions are inductive. The basis of the induction is when a statement is a single basic block. The rules for the basis case, as presented in Section 6.2, state that the formation of the s-definition depends on the type of the dsv basic block (see Figure 6.4).

Next, we explain the rules for the formation of the flow graph s-definitions for an entire region, S, in a flow graph. The rules are stated in terms of the component regions of S, such as S1 and S2.

6.4.1 Compound statements

Assume that the flow graph of a compound statement, S, includes two regions S1 and S2. (See Figure 6.4.) Then for any dynamic state variable dsv, the flow graph s-definition for $\underline{\text{dsv}}_{\downarrow S}$ is as follows.

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{dsv}_{\downarrow S2})$$

In other words, the class of dsv upon exiting from S is defined as the class of dsv upon exiting from S2.

For example, consider the following code.

```
S: S1: B1: dsv = x;
           dsv = dsv + 1;
   S2: S3: B2: M1.P();
       S4: S5: B3: M2.Q( | dsv);
           S6: B4: x = 0;
```

Region S1 consists of B_1 and region S2 consists of B_2, B_3, and B_4. We have

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{dsv}_{\downarrow S2}).$$

We can apply the same rule again until $\underline{\text{dsv}}_{\downarrow S2}$ can be defined as an induction base case. Assume that S2 is a compound of S3 (B_2) and S4 (B_3 and B_4) and also that S4 is a compound of S5 (B_5) and S6 (B_6). Then, we have

$$\underline{\text{dsv}}_{\downarrow S2} \equiv \oplus(\text{dsv}_{\downarrow S4}),$$
$$\underline{\text{dsv}}_{\downarrow S4} \equiv \oplus(\text{dsv}_{\downarrow S6}), \text{ and}$$
$$\underline{\text{dsv}}_{\downarrow S6} \equiv \oplus(\text{dsv}_{\downarrow 4}).$$

Since S6 consists of one block, B_4, then $\underline{\text{dsv}}_{\downarrow S6}$ is $\underline{\text{dsv}}_{\downarrow 4}$ which can be defined as

$$\underline{\text{dsv}}_{\downarrow 4} \equiv \oplus(\text{dsv}_{\uparrow 4})$$

because B_4 is not a dsv basic block. In turn, we have

$$\underline{\text{dsv}}_{\uparrow 4} \equiv \oplus(\text{dsv}_{\downarrow 3}) \text{ and}$$
$$\underline{\text{dsv}}_{\downarrow 3} \equiv \oplus(\text{M2.Q.dsv}, \text{IMP}_3)$$

since B_3 is a call unambiguous dsv basic block (see Section 6.2). So, we finally obtain the following s-definition:

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{M2.Q.dsv}, \text{IMP}_3).$$

Of course, IMP_3 can be defined further as explain previously. For example, if no statements precede S in the procedure, we have

$$\text{IMP}_3 \equiv \oplus(\text{IMP}).$$

6.4.2 If-then statements

Assume that the flow graph of an if-then statement, S, includes a conditional branch block, B_i, and a region S1 for the code of the body. (See Figure 6.4.) Then for any dynamic state variable dsv, we generate an s-definition as follows.

Case 1: If S1 includes at least one dsv unambiguous block, then $\underline{\text{dsv}}_{\downarrow S}$, the class of dsv upon exit from region S, is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \neg\text{B}_{I_{S1}}[\oplus(\text{dsv}_{\uparrow i}, \text{IMP}_i, \text{E}_i)].$$

The flow graph s-definition can be read as follows. If the first block ($\text{B}_{I_{S1}}$) of region S1 is entered during the execution of the procedure then the the class of dsv after the execution of the if-statement (S) is

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{dsv}_{\downarrow S1})$$

i.e., the class of dsv upon exiting from region S1[5]. On the other hand, if $\text{B}_{I_{S1}}$ is not entered, then the class of dsv is

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{dsv}_{\uparrow i}, \text{IMP}_i, \text{E}_i).$$

[5]The s-variable $\text{dsv}_{\downarrow S1}$ is defined by induction. It already includes the implicit flows coming into region S1.

i.e., the class of dsv upon entering the conditional branch basic blocks B_i incremented by the implicit flows IMP_i and $\underline{E_i}$. The reason the implicit flows need to be included is that we have assumed that there exists an unambiguous s-assignment of dsv in S1. As explained, in Chapters 2 and 4, implicit flows occur even if explicit assignment are skipped.

As an example, consider the following code.

```
B4: ...
S: B5: if (E5) then
          S1: B6: dsv = x;
              B7:  M.P();
B8: M.Q();
```

Region S consists of B_5 and region S1 consists of B_6 (its header) and B_7. The flow graph s-deflnition for $\underline{dsv}_{\downarrow S}$ is

$$\underline{dsv}_{\downarrow S} \equiv B_6[\oplus(dsv_{\downarrow S1})] \vee \neg B_6[\oplus(dsv_{\uparrow 5}, IMP_5, E_5)].$$

The implicit flows IMP_5 and $\underline{E_5}$ are included in the case of skipping S1. However, they need not be included in the other case since $\underline{dsv}_{\downarrow S1}$ already includes them.

We can further define $\underline{dsv}_{\downarrow S1}$ as $\underline{dsv}_{\downarrow 7}$, the class of dsv upon exiting B_7, which, in turn, is defined (by the rules of the basis of the induction in Section 6.2) as

$$\underline{dsv}_{\downarrow 7} \equiv \oplus(dsv \Downarrow M.P).$$

Note that the s-definition for $\underline{dsv}_{\downarrow S}$ also defines $\underline{dsv}_{\uparrow 8}$ and $\underline{dsv} \Uparrow M.Q$. The latter s-variable is the one that appears in the template of the procedure under the INS part for the M.Q RPC. We finally have

$$\underline{\text{dsv}}\Uparrow\text{M.Q} \equiv \text{B}_6[\oplus(\text{dsv}\Downarrow\text{M.P})] \vee \neg\text{B}_6[\oplus(\text{dsv}_{\uparrow 5}, \text{IMP}_5, \text{E}_5)].$$

Of course, $\underline{\text{dsv}}_{\uparrow 5}$ and IMP_5 need to be further defined depending on the code preceding B_5.

Case 2: Implicit flows need not be included if there are no unambiguous s-assignments of dsv and if S1 is skipped. So, if S1 does not include any dsv unambiguous block, then $\underline{\text{dsv}}_{\downarrow S}$ is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \neg\text{B}_{I_{S1}}[\oplus(\text{dsv}_{\uparrow i})].$$

As example, consider the following code.

```
B4: ...
S: B5: if (E5) then
          S1: B6:  a = b;
              B7:  M.P();
B8: M.Q();
```

In this case, B_6 is not a dsv basic block anymore and there are no unambiguous dsv s-assignment in S1. Therefore, the implicit flows need not be included since only ambiguous flows appear. Then, the flow graph s-definition for $\underline{\text{dsv}}_{\downarrow S}$ is

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_6[\oplus(\text{dsv}_{\downarrow S1})] \vee \neg\text{B}_6[\oplus(\text{dsv}_{\uparrow 5})].$$

6.4.3 If-then-else statements

Assume that the flow graph of an if-then-else statement, S, includes a conditional branch block, B_i, and a region S1 for the code under the then-part and a region S2 for

the code under the else-part. (See Figure 6.4.) Then for any dynamic state variable dsv, we generate an s-definition as follows.

Case 1: If neither S1 nor S2 include any dsv unambiguous block, then $\underline{\text{dsv}}_{\downarrow S}$ is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \text{B}_{I_{S2}}[\oplus(\text{dsv}_{\downarrow S2})].$$

In this case, S1 and S2 either contain ambiguous dsv s-assignments or do not contain dsv s-assignments at all. Implicit flows need not be explicitly included since no unambiguous flows occur in either S1 or S2.

Case 2: If only S1 includes at least one dsv unambiguous block while S2 does not, then $\underline{\text{dsv}}_{\downarrow S}$ is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \text{B}_{I_{S2}}[\oplus(\text{dsv}_{\downarrow S2}, \text{IMP}_i, \text{E}_i)].$$

If S2 is visited, it must mean that control has skipped S1. In that case, since S1 is assumed to contain an unambiguous dsv s-assignment, the implicit flows occur and must be included. So, the implicit flows are included in the $\text{B}_{I_{S2}}$ bracket because of the unambiguous s-assignment in S1, regardless of what S2 contains.

As an example, consider the following piece of code.

```
B4: ...
S: B5: if (E5) then
          S1: B6:  dsv = x;
       else
          S2: B7:  M.P();
```

Region S consists of B_5 and S1 (B_6) and S2 (B_7). We have

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_6[\oplus(\text{dsv}_{\downarrow S1})] \vee \text{B}_7[\oplus(\text{dsv}_{\downarrow S2}, \text{IMP}_5, \text{E}_5)],$$
$$\underline{\text{dsv}}_{\downarrow S1} \equiv \oplus(\text{dsv}_{\downarrow 6}),$$
$$\underline{\text{dsv}}_{\downarrow 6} \equiv \oplus(\text{x}, \text{IMP}_6),$$
$$\text{IMP}_6 \equiv \oplus(\text{E}_5, \text{IMP}_5),$$
$$\underline{\text{dsv}}_{\downarrow S2} \equiv \oplus(\text{dsv}_{\downarrow 7}), \text{ and}$$
$$\underline{\text{dsv}}_{\downarrow 7} \equiv \oplus(\text{dsv}\Downarrow\text{M.P}).$$

So, finally, we can write

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_6[\oplus(\text{x}, \text{E}_5, \text{IMP}_5)] \vee \text{B}_7[\oplus(\text{dsv}\Downarrow\text{M.P}, \text{IMP}_5, \text{E}_5)].$$

Note how the implicit flows E_5 and IMP_5 into the B_6 bracket ended up being included because they were included in $\underline{\text{dsv}}_{\downarrow S1}$. Also, the implicit flows must be included in the B_7 bracket. Otherwise, in case B_6 is skipped and M.P does not s-assign dsv, $\underline{\text{dsv}}$ would be underclassified to a class that does not reflect the implicit flows resulting from the conditional explicit assignment in B_6.

Case 3: The reverse of the above case is treated similarly. If only S2 includes at least one dsv unambiguous block while S1 does not, then $\underline{\text{dsv}}_{\downarrow S}$ is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1}, \text{IMP}_i, \text{E}_i)] \vee \text{B}_{I_{S2}}[\oplus(\text{dsv}_{\downarrow S2})].$$

Case 4: Next, we treat the last possible case. Namely, if both S1 and S2 each include at least one dsv unambiguous block, then $\underline{\text{dsv}}_{\downarrow S}$ is defined as

$$\underline{\text{dsv}}_{\downarrow S} \equiv \text{B}_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1}, \text{IMP}_i, \text{E}_i)] \vee \text{B}_{I_{S2}}[\oplus(\text{dsv}_{\downarrow S2}, \text{IMP}_i, \text{E}_i)].$$

The implicit flows IMP_i and $\underline{\text{E}_i}$ are included in all cases since both S1 and S2 unambiguously s-assign dsv. The inclusion of IMP_i and $\underline{\text{E}_i}$ in the $\text{B}_{I_{S1}}$ bracket is

because of the unambiguous flow in S2. Similarly, the inclusion of IMP_i and $\underline{E_i}$ in the $B_{I_{S2}}$ bracket is because of the unambiguous flow in S1. However, since $\underline{dsv}_{\downarrow S1}$ and $\underline{dsv}_{\downarrow S2}$ already include those implicit flows, to avoid including redundant classes, the s-definition can be simplified to

$$\underline{dsv}_{\downarrow S} \equiv B_{I_{S1}}[\oplus(dsv_{\downarrow S1})] \vee B_{I_{S2}}[\oplus(dsv_{\downarrow S2})].$$

Note that this s-definition is the same as the one generated in case neither S1 nor S2 contain unambiguous flows.

As an example, consider the following piece of code.

```
B4: ...
S: B5: if (E5) then
          S1: B6:  dsv = x;
       else
          S2: B7:  M.P( | dsv);
```

Region S consists of B_5 and S1 (B_6) and S2 (B_7). Both B_6 and B_7 are dsv unambiguous basic blocks. We have

$$\underline{dsv}_{\downarrow S} \equiv B_6[\oplus(dsv_{\downarrow S1})] \vee B_7[\oplus(dsv_{\downarrow S2})],$$
$$\underline{dsv}_{\downarrow S1} \equiv \oplus(dsv_{\downarrow 6}),$$
$$\underline{dsv}_{\downarrow 6} \equiv \oplus(x, IMP_6),$$
$$IMP_6 \equiv \oplus(E_5, IMP_5),$$
$$\underline{dsv}_{\downarrow S2} \equiv \oplus(dsv_{\downarrow 7}),$$
$$\underline{dsv}_{\downarrow 7} \equiv \oplus(M.P.dsv, IMP_7), \text{ and}$$
$$IMP_7 \equiv \oplus(E_5, IMP_5).$$

So, finally, we can write

$$\underline{\mathrm{dsv}}_{\downarrow S} \equiv \mathrm{B}_6[\oplus(\mathrm{x},\ \mathrm{IMP}_5,\ \mathrm{E}_5)] \vee \mathrm{B}_7[\oplus(\mathrm{M.P.dsv},\ \mathrm{IMP}_5,\ \mathrm{E}_5)].$$

Note how the implicit flows E_5 and IMP_5 ended up being included because they were already included in $\underline{\mathrm{dsv}}_{\downarrow S1}$ and $\underline{\mathrm{dsv}}_{\downarrow S2}$ which contain unambiguous dsv blocks. There was no need to include them in the original s-definition.

6.4.4 While statements

Among the types of statements, loops are the most complicated to handle. The basic idea that we will follow is to assume that a while statement consists of a succession of if-then statements. This way we simulate the iterations of the while loop. Then, we generate a dsv s-definition for each of those if-statements as explained above. The questions are how many s-definition is it necessary to generate and are there an infinite number of them?

Mizuno in [27] proves that for any loop, there is always only a finite number of s-definitions to be generated. The idea of the proof is simple: there is always a finite number of constant security classes and of s-variables that could possibly be included in an s-definition. Therefore, after a certain number, say n, of generated s-definitions, no new s-variables or constant classes can be added to the $(n+1)^{\mathrm{st}}$ s-definition. Thus, the process of generating s-definitions can terminate when it is found that no new classes are being added to a previously generated s-definition.

Nesting loops also introduces complications. As a result, our treatment of loops is divided into two cases. First, we treat the simplest case, that is, we consider loops that are not nested in any other loops. Those loops will be called *level-1* loops. The handling of level-1 loops is a generalization of of the handling of if-then statements.

Second, we treat loops that may be nested at any level n, *level-n* loops. The handling of level-n loops is a generalization of the handling of level-1 loops.

6.4.4.1 Level-1 loops We will need to introduce some new notations. First however, as for the previous control structures, we assume that the flow graph of a level-1 while statement, S, includes a conditional branch block, B_i, and a region S1 for the code of the body of the loop. (See Figure 6.4.) At the end of S1 there must be an unconditional jump to B_i. However, to simulate while statements as a succession of if-then statements, we will assume that the unconditional jump statement is omitted from the code bodies of the if-then statements. We will explain how to generate a (finite) series of s-definitions for any dynamic state variable dsv.

In Figure 6.7, we show a flow graph for a level-1 while statement as a succession of if-then statements. The k^{th} if-then in the sequence of a while statement, S, will be denoted by $S^{(k)}$. We introduce the notations for $S^{(k)}$ in Figure 6.8. They are similar to our previously established notations except that a superscript is included to identify the number of the iteration.

We also denote, by $\underline{\text{dsv}}^{(0)}_{\downarrow S}$, the class of dsv if the level-1 loop is skipped (i.e., iterated 0 times). Then, we have the following flow graph s-definitions for dsv. If S1 contains at least one unambiguous dsv basic block then we have

$$\underline{\text{dsv}}^{(0)}_{\downarrow S} \equiv \oplus(\text{dsv}^{(1)}_{\uparrow i}, \text{IMP}^{(1)}_i, \text{E}^{(1)}_i).$$

This means that if the loop is completely skipped (i.e., $S1^{(1)}$ is not entered), then the class of dsv is equal to its class before the loop, incremented by the implicit flows since S1 contains an unambiguous dsv s-assignment.

If S1 does not contain any unambiguous dsv basic blocks, the we have

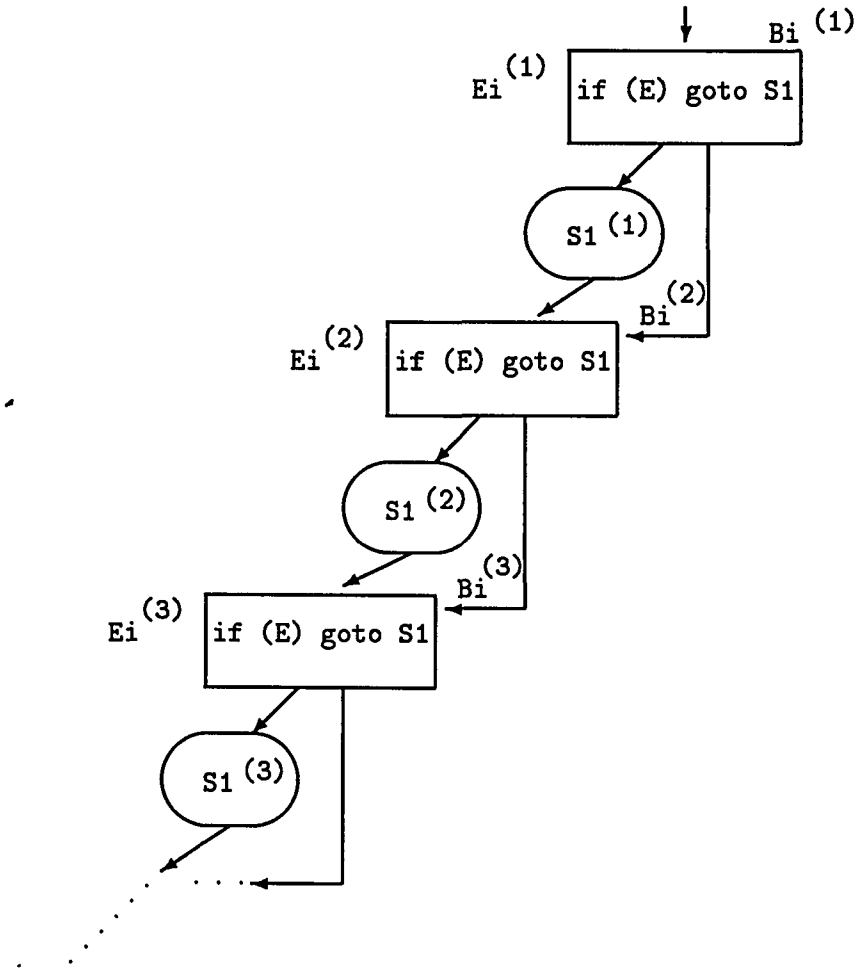

Figure 6.7: A while statement as a succession of if-then statements

$B_i^{(k)}$	Basic block B_i of $S^{(k)}$ or the flag associated with B_i in the loop.
$E_i^{(k)}$	The condition appearing in basic block $B_i^{(k)}$.
$S1^{(k)}$	The region of the body of $S^{(k)}$.
$IMP_i^{(k)}$	The implicit flow into basic block $B_i^{(k)}$.
$IMP_{S1}^{(k)}$	The implicit flow into region $S1^{(k)}$.
$B_{I_{S1}}^{(k)}$	The header of region $S1^{(k)}$.
$\underline{dsv}_{\uparrow i}^{(k)}$	The class of dsv upon entering basic block $B_i^{(k)}$.
$\underline{dsv}_{\downarrow i}^{(k)}$	The class of dsv upon exiting basic block $B_i^{(k)}$.
$\underline{dsv}_{\uparrow S1}^{(k)}$	The class of dsv upon entering region $S1^{(k)}$.
$\underline{dsv}_{\downarrow S1}^{(k)}$	The class of dsv upon exiting region $S1^{(k)}$.

Figure 6.8: Notations for while loops

$$\underline{\text{dsv}}^{(0)}_{\downarrow S} \equiv \oplus(\text{dsv}^{(1)}_{\uparrow i}),$$

which omits the implicit flows.

Assume now that the body of the loop is entered at least once. Then, after iteration k, the flow graph s-definition for dsv is

$$\underline{\text{dsv}}^{(k)}_{\downarrow S} \equiv \oplus(\text{dsv}^{(k)}_{\downarrow S1}).$$

In other words, the class of dsv after iteration k of the loop is defined to be the class of dsv upon exiting the region $S1^{(k)}$, the k^{th} if-then in the sequence. The implicit flows are not included since they are included in the computation of $\text{dsv}^{(k)}_{\downarrow S1}$.

As an example, consider the following code.

```
B4: ...
S: B5: while (E5: dsv1 > 0) do
          S1: B6: dsv1 = dsv2;
                  dsv2 = dsv3;
```

Region S consists of B_5 and B_6 and region S1 consists of B_6. The condition E_5 is (dsv1 > 0). The series of flow graph s-definitions for $\underline{\text{dsv1}}_{\downarrow S}$ is as follows. Note that S1 contains an unambiguous dsv1 basic block.

Case 1: No iterations.

$$\underline{\text{dsv1}}^{(0)}_{\downarrow S} \equiv \oplus(\text{dsv1}^{(1)}_{\uparrow 5}, \text{IMP}^{(1)}_{5}, \text{E}^{(1)}_{5})$$
$$\underline{\text{dsv1}}^{(1)}_{\uparrow 5} \equiv \oplus(\text{dsv1}_{\uparrow S})$$
$$\text{IMP}^{(1)}_{5} \equiv \oplus(\text{IMP}_{S})$$
$$\underline{\text{E}^{(1)}_{5}} \equiv \oplus(\text{dsv1}_{\uparrow S})$$

Finally, we get

$$\underline{\mathrm{dsv1}}^{(0)}_{\downarrow S} \equiv \oplus(\mathrm{dsv1}_{\uparrow S}, \mathrm{IMP}_S).$$

Case 2: One iteration.

$$\begin{aligned}
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow S} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow S1}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow S1} &\equiv \oplus(\mathrm{dsv2}^{(1)}_{\uparrow 6}, \mathrm{IMP}^{(1)}_6) \\
\underline{\mathrm{dsv2}}^{(1)}_{\uparrow 6} &\equiv \oplus(\mathrm{dsv2}^{(1)}_{\downarrow 5}) \\
\underline{\mathrm{dsv2}}^{(1)}_{\downarrow 5} &\equiv \oplus(\mathrm{dsv2}^{(1)}_{\uparrow 5}) \\
\underline{\mathrm{dsv2}}^{(1)}_{\uparrow 5} &\equiv \oplus(\mathrm{dsv2}_{\uparrow S}) \\
\mathrm{IMP}^{(1)}_6 &\equiv \oplus(\mathrm{IMP}^{(1)}_5, \mathrm{E}^{(1)}_5) \\
\mathrm{IMP}^{(1)}_5 &\equiv \oplus(\mathrm{IMP}_S) \\
\underline{\mathrm{E}^{(1)}_5} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\uparrow 5}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\uparrow 5} &\equiv \oplus(\mathrm{dsv1}_{\uparrow S})
\end{aligned}$$

Finally, we get

$$\underline{\mathrm{dsv1}}^{(1)}_{\downarrow S} \equiv \oplus(\mathrm{dsv2}_{\uparrow S}, \mathrm{IMP}_S, \mathrm{dsv1}_{\uparrow S}).$$

Case 3: Two iterations.

$$\begin{aligned}
\underline{\mathrm{dsv1}}^{(2)}_{\downarrow S} &\equiv \oplus(\mathrm{dsv1}^{(2)}_{\downarrow S1}) \\
\underline{\mathrm{dsv1}}^{(2)}_{\downarrow S1} &\equiv \oplus(\mathrm{dsv2}^{(2)}_{\uparrow 6}, \mathrm{IMP}^{(2)}_6) \\
\underline{\mathrm{dsv2}}^{(2)}_{\uparrow 6} &\equiv \oplus(\mathrm{dsv2}^{(2)}_{\downarrow 5}) \\
\underline{\mathrm{dsv2}}^{(2)}_{\downarrow 5} &\equiv \oplus(\mathrm{dsv2}^{(2)}_{\uparrow 5}) \\
\underline{\mathrm{dsv2}}^{(2)}_{\uparrow 5} &\equiv \oplus(\mathrm{dsv2}^{(1)}_{\downarrow 6}) \\
\underline{\mathrm{dsv2}}^{(1)}_{\downarrow 6} &\equiv \oplus(\mathrm{dsv3}^{(1)}_{\uparrow 6}) \\
\underline{\mathrm{dsv3}}^{(1)}_{\uparrow 6} &\equiv \oplus(\mathrm{dsv3}_{\uparrow S})
\end{aligned}$$

$$\begin{aligned}
\mathrm{IMP}_6^{(2)} &\equiv \oplus(\mathrm{IMP}_5^{(2)}, \mathrm{E}_5^{(2)}) \\
\mathrm{IMP}_5^{(2)} &\equiv \oplus(\mathrm{IMP}_5^{(1)}, \mathrm{E}_5^{(1)}) \\
\mathrm{IMP}_5^{(1)} &\equiv \oplus(\mathrm{IMP}_S) \\
\underline{\mathrm{E}_5^{(1)}} &\equiv \oplus(\mathrm{dsv1}_{\uparrow S}) \\
\underline{\mathrm{E}_5^{(2)}} &\equiv \oplus(\mathrm{dsv1}_{\uparrow 5}^{(2)}) \\
\underline{\mathrm{dsv1}}_{\uparrow 5}^{(2)} &\equiv \oplus(\mathrm{dsv1}_{\downarrow S1}^{(1)})
\end{aligned}$$

Finally, we get

$$\underline{\mathrm{dsv1}}_{\downarrow S}^{(2)} \equiv \oplus(\mathrm{dsv3}_{\uparrow S}, \mathrm{IMP}_S, \mathrm{dsv1}_{\uparrow S}, \mathrm{dsv2}_{\uparrow S}).$$

Case 4: Three iterations.

$$\begin{aligned}
\underline{\mathrm{dsv1}}_{\downarrow S}^{(3)} &\equiv \oplus(\mathrm{dsv1}_{\downarrow S1}^{(3)}) \\
\underline{\mathrm{dsv1}}_{\downarrow S1}^{(3)} &\equiv \oplus(\mathrm{dsv2}_{\uparrow 6}^{(3)}, \mathrm{IMP}_6^{(3)}) \\
\underline{\mathrm{dsv2}}_{\uparrow 6}^{(3)} &\equiv \oplus(\mathrm{dsv2}_{\downarrow 5}^{(3)}) \\
\underline{\mathrm{dsv2}}_{\downarrow 5}^{(3)} &\equiv \oplus(\mathrm{dsv2}_{\uparrow 5}^{(3)}) \\
\underline{\mathrm{dsv2}}_{\uparrow 5}^{(3)} &\equiv \oplus(\mathrm{dsv2}_{\downarrow 6}^{(2)}) \\
\underline{\mathrm{dsv2}}_{\downarrow 6}^{(2)} &\equiv \oplus(\mathrm{dsv3}_{\uparrow 6}^{(2)}) \\
\underline{\mathrm{dsv3}}_{\uparrow 6}^{(2)} &\equiv \oplus(\mathrm{dsv3}_{\downarrow 5}^{(2)}) \\
\underline{\mathrm{dsv3}}_{\downarrow 5}^{(2)} &\equiv \oplus(\mathrm{dsv3}_{\uparrow 5}^{(2)}) \\
\underline{\mathrm{dsv3}}_{\uparrow 5}^{(2)} &\equiv \oplus(\mathrm{dsv3}_{\downarrow 6}^{(1)}) \\
\underline{\mathrm{dsv3}}_{\uparrow 6}^{(1)} &\equiv \oplus(\mathrm{dsv3}_{\uparrow S}) \\
\mathrm{IMP}_6^{(3)} &\equiv \oplus(\mathrm{IMP}_5^{(3)}, \mathrm{E}_5^{(3)}) \\
\mathrm{IMP}_5^{(3)} &\equiv \oplus(\mathrm{IMP}_5^{(2)}, \mathrm{E}_5^{(2)}) \\
\mathrm{IMP}_5^{(2)} &\equiv \oplus(\mathrm{IMP}_5^{(1)}, \mathrm{E}_5^{(1)})
\end{aligned}$$

$$\begin{aligned}
&\mathrm{IMP}_5^{(1)} \equiv \oplus(\mathrm{IMP}_S)\\
&\underline{\mathrm{E}_5^{(1)}} \equiv \oplus(\mathrm{dsv1}_{\uparrow S})\\
&\underline{\mathrm{E}_5^{(2)}} \equiv \oplus(\mathrm{dsv1}_{\downarrow S1}^{(1)})\\
&\underline{\mathrm{E}_5^{(3)}} \equiv \oplus(\mathrm{dsv1}_{\uparrow 5}^{(3)})\\
&\underline{\mathrm{dsv1}}_{\uparrow 5}^{(3)} \equiv \oplus(\mathrm{dsv1}_{\downarrow S1}^{(2)})\\
&\underline{\mathrm{dsv1}}_{\uparrow 5}^{(2)} \equiv \oplus(\mathrm{dsv1}_{\downarrow S1}^{(1)})
\end{aligned}$$

Finally, we get

$$\underline{\mathrm{dsv1}}_{\downarrow S}^{(3)} \equiv \oplus(\mathrm{dsv3}_{\uparrow S}, \mathrm{IMP}_S, \mathrm{dsv1}_{\uparrow S}, \mathrm{dsv2}_{\uparrow S}).$$

So, after simulating three iterations, we find that $\underline{\mathrm{dsv1}}_{\downarrow S}^{(3)}$ and $\underline{\mathrm{dsv1}}_{\downarrow S}^{(4)}$ have equivalent s-definitions and the process of generating s-definitions can halt. If at run-time the loop is found to have run 0, 1, or 2 times, then to compute the class of dsv1, the run-time mechanism uses $\underline{\mathrm{dsv1}}_{\downarrow S}^{(0)}$, $\underline{\mathrm{dsv1}}_{\downarrow S}^{(1)}$, and $\underline{\mathrm{dsv1}}_{\downarrow S}^{(2)}$, respectively. If the loop executes 3 or more times, $\underline{\mathrm{dsv1}}_{\downarrow S}^{(2)}$ is also used.

In general, the $\underline{\mathrm{dsv}}_{\downarrow S}^{(k)}$ s-definitions are generated for

$$\mathrm{k} = 0, 1, \ldots, n-1, n,$$

where the s-definitions for $\underline{\mathrm{dsv}}_{\downarrow S}^{(n)}$ and $\underline{\mathrm{dsv}}_{\downarrow S}^{(n+1)}$ are equivalent. If at run-time it is found that the loop iterated k times and if $\mathrm{k} \leq \mathrm{n}$, then the run-time mechanism uses the $\underline{\mathrm{dsv}}_{\downarrow S}^{(k)}$ s-definition to compute the $\underline{\mathrm{dsv}}$. On the other hand, if $\mathrm{k} > \mathrm{n}$, then the $\underline{\mathrm{dsv}}_{\downarrow S}^{(n)}$ s-definition is used.

At run-time, for each loop, S, with body code, S1, a counter is maintained to indicate the number of times the loop is executed. In other words, it counts the number of times $\mathrm{B}_{I_{S1}}$ is entered. We will denote the counter, for a while statement S, by S also.

6.4.4.2 Level-n loops To generate s-definitions for a level-1 loop, we generalized the if-then statement case and we introduced a (one-dimensional) superscript, say k, in our notations to indicate the number of the iteration. To generate s-definitions for a level-n loop, S, we further generalize the level-1 case and allow the superscript in our notations to stand for an n-dimensional vector of n counters (n-tuple). The vector represents the number of iterations of the various loops that contain the loop S.

For a vector $\underline{k} = (k_1, k_2, \ldots, k_n)$, k_1 is a counter that represents the number of iterations of the most outer loop (a level-1 loop) containing the level-n loop S, k_2 represents the number of iterations for the level-2 loop containing S, and so on.... The counter k_n represents the number of iterations of S itself.

For example, for a level-3 loop, S, $S^{(2,4,1)}$ represents the if-then statement simulating S on one S iteration of the fourth directly containing loop iteration on second iteration of the most outer loop. Associated with $S^{(2,4,1)}$, are a conditional branch block, $B_i^{(2,4,1)}$, a condition, $E_i^{(2,4,1)}$, and so on....

It is assumed that for a loop vector, if one of its components is 0 then all the succeeding components are 0. That is obvious since if a level-i loop does not execute then a contained level-(i+1) loop does not execute either.

We denote the 0 vector by $\underline{0}$ and we use $(\underline{k_{i-1}}, \underline{m})$ to denote the vector whose components are different than m, except for the i^{th} and beyond positions.

We define the following s-definitions for a level-n loop, S, with body code, S1, and conditional branch block B_i. For 0 iteration of S, $\underline{dsv}_{\downarrow S}^{(k_1, \ldots, k_{n-1}, 0)}$, the class of dsv after 0 iterations of S and $k_1, k_2, \ldots$ and k_{n-1} iterations of its respective containing loops, is

$$\underline{\mathrm{dsv}}_{\downarrow S}^{(k_{n-1},0)} \equiv \oplus(\ \mathrm{dsv}_{\uparrow i}^{(k_{n-1},1)}, \mathrm{IMP}_i^{(k_{n-1},1)}, \mathrm{E}_i^{(k_{n-1},1)}\ ),$$

if the loop contains unambiguous dsv s-assignments, or

$$\underline{\mathrm{dsv}}_{\downarrow S}^{(k_{n-1},0)} \equiv \oplus(\ \mathrm{dsv}_{\uparrow i}^{(k_{n-1},1)}\ ),$$

if there are no dsv unambiguous s-assignments.

For k_n iterations of S, we have

$$\underline{\mathrm{dsv}}_{\downarrow S}^{(k)} \equiv \oplus(\mathrm{dsv}_{\downarrow S1}^{(k)}).$$

As an example, consider the following code.

```
S: B3: while (E3: dsv1 > 0) do
   S1: B4: dsv1 = dsv4;
       S2: B5: while (E5: dsv1 > 0) do
           S3: B6: dsv1 = dsv2;
                   dsv2 = dsv3;
       B7: dsv4 = dsv1;
```

Region S is a a level-1 loop that consists of B_3, B_4, B_5, B_6, and B_7. Region S2 is a level-2 loop consisting of B_5 and B_6. Note that S2 is a loop identical to the loop in the example we gave for level-1 loops.

We use the level-1 s-definitions to derive the following s-definitions for $\underline{\mathrm{dsv1}}_{\downarrow S}^{(0)}$.

$$\underline{\mathrm{dsv1}}_{\downarrow S}^{(0)} \equiv \oplus(\mathrm{dsv1}_{\uparrow 3}^{(1)}, \mathrm{IMP}_3^{(1)}, \mathrm{E}_3^{(1)})$$
$$\underline{\mathrm{dsv1}}_{\uparrow 3}^{(1)} \equiv \oplus(\mathrm{dsv1}_{\uparrow S})$$
$$\mathrm{IMP}_3^{(1)} \equiv \oplus(\mathrm{IMP}_S)$$
$$\underline{\mathrm{E}_3^{(1)}} \equiv \oplus(\mathrm{dsv1}_{\uparrow S})$$

Finally, we get

$$\underline{\mathrm{dsv1}}^{(0)}_{\downarrow S} \equiv \oplus(\mathrm{dsv1}_{\uparrow S}, \mathrm{IMP}_S).$$

Again, we use the level-1 s-definitions to derive the following s-definitions for one iteration of S.

$$\begin{aligned}
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow S} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow S1}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow S1} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow 7}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow 7} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\uparrow 7}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\uparrow 7} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow S2})
\end{aligned}$$

Since S2 is a level-2 loop, we need to use the level-2 s-definitions to produce a series of s-definitions for $\underline{\mathrm{dsv1}}^{(1,0)}_{\downarrow S2}$ (1 iteration of S and 0 iterations of S2), $\underline{\mathrm{dsv1}}^{(1,1)}_{\downarrow S2}$ (1 iteration of S and 1 iteration of S2), $\underline{\mathrm{dsv1}}^{(1,2)}_{\downarrow S2}$ (1 iteration of S and 2 iterations of S2), and so on.... The s-definition for $\underline{\mathrm{dsv1}}^{(1,0)}_{\downarrow S2}$ is derived as follows.

$$\begin{aligned}
\underline{\mathrm{dsv1}}^{(1,0)}_{\downarrow S2} &\equiv \oplus(\mathrm{dsv1}^{(1,0)}_{\uparrow 5}, \mathrm{IMP}^{(1,0)}_5, \mathrm{E}^{(1,0)}_5) \\
\underline{\mathrm{dsv1}}^{(1,0)}_{\uparrow 5} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\uparrow S2}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\uparrow S2} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow 4}) \\
\underline{\mathrm{dsv1}}^{(1)}_{\downarrow 4} &\equiv \oplus(\mathrm{dsv4}^{(1)}_{\uparrow 4}, \mathrm{IMP}^{(1)}_4) \\
\underline{\mathrm{dsv4}}^{(1)}_{\uparrow 4} &\equiv \oplus(\mathrm{dsv4}^{(1)}_{\uparrow 3}) \\
\underline{\mathrm{dsv4}}^{(1)}_{\uparrow 3} &\equiv \oplus(\mathrm{dsv4}_{\uparrow S}) \\
\mathrm{IMP}^{(1)}_4 &\equiv \oplus(\mathrm{dsv1}_{\uparrow S}, \mathrm{IMP}_S) \\
\mathrm{IMP}^{(1,0)}_5 &\equiv \oplus(\mathrm{dsv1}_{\uparrow S}, \mathrm{IMP}_S) \\
\underline{\mathrm{E}^{(1,0)}_5} &\equiv \oplus(\mathrm{dsv1}^{(1)}_{\downarrow 4})
\end{aligned}$$

Finally, we get

$$\underline{\text{dsv1}}^{(1,0)}_{\downarrow S2} \equiv \oplus(\text{dsv4}_{\uparrow S}, \text{dsv1}_{\uparrow S}, \text{IMP}_S).$$

We also have the following s-definitions.

$$\begin{aligned}
\underline{\text{dsv1}}^{(1,1)}_{\downarrow S2} &\equiv \oplus(\text{dsv1}^{(1,1)}_{\downarrow S3}) \\
\underline{\text{dsv1}}^{(1,1)}_{\downarrow S3} &\equiv \oplus(\text{dsv2}^{(1,1)}_{\uparrow 6}, \text{IMP}^{(1,1)}_{6}) \\
\underline{\text{dsv2}}^{(1,1)}_{\uparrow 6} &\equiv \oplus(\text{dsv2}^{(1,1)}_{\uparrow 5}) \\
\underline{\text{dsv2}}^{(1,1)}_{\uparrow 5} &\equiv \oplus(\text{dsv2}^{(1)}_{\uparrow 4}) \\
\underline{\text{dsv2}}^{(1)}_{\uparrow 4} &\equiv \oplus(\text{dsv2}^{(1)}_{\uparrow 3}) \\
\underline{\text{dsv2}}^{(1)}_{\uparrow 3} &\equiv \oplus(\text{dsv2}_{\uparrow S}) \\
\text{IMP}^{(1,1)}_{6} &\equiv \oplus(\text{IMP}^{(1,1)}_{5}, \text{E}^{(1,1)}_{5}) \\
\text{IMP}^{(1,1)}_{5} &\equiv \oplus(\text{dsv1}^{(1)}_{\downarrow 4}) \\
\underline{\text{dsv1}}^{(1)}_{\downarrow 4} &\equiv \oplus(\text{dsv4}_{\uparrow S}, \text{dsv1}_{\uparrow S}, \text{IMP}_S) \\
\underline{\text{E}^{(1,1)}_{5}} &\equiv \oplus(\text{dsv1}^{(1,1)}_{\uparrow 5}) \\
\underline{\text{dsv1}}^{(1,1)}_{\uparrow 5} &\equiv \oplus(\text{dsv1}^{(1)}_{\uparrow S2}) \\
\underline{\text{dsv1}}^{(1)}_{\uparrow S2} &\equiv \oplus(\text{dsv1}^{(1)}_{\downarrow 4})
\end{aligned}$$

Finally, we get

$$\underline{\text{dsv1}}^{(1,1)}_{\downarrow S2} \equiv \oplus(\text{dsv2}_{\uparrow S}, \text{dsv4}_{\uparrow S}, \text{dsv1}_{\uparrow S}, \text{IMP}_S).$$

We similarly get

$$\underline{\text{dsv1}}^{(1,2)}_{\downarrow S2} \equiv \oplus(\text{dsv3}_{\uparrow S}, \text{dsv2}_{\uparrow S}, \text{dsv4}_{\uparrow S}, \text{dsv1}_{\uparrow S}, \text{IMP}_S).$$

Now, we derive the following s-definitions for two iterations of S.

$$\underline{\text{dsv1}}^{(2)}_{\downarrow S} \equiv \oplus(\text{dsv1}^{(2)}_{\downarrow S1})$$

$$\underline{\text{dsv1}}^{(2)}_{\downarrow S1} \equiv \oplus(\text{dsv1}^{(2)}_{\downarrow 7})$$
$$\underline{\text{dsv1}}^{(2)}_{\downarrow 7} \equiv \oplus(\text{dsv1}^{(2)}_{\uparrow 7})$$
$$\underline{\text{dsv1}}^{(2)}_{\uparrow 7} \equiv \oplus(\text{dsv1}^{(2)}_{\downarrow S2})$$

Once again, we need to derive s-definitions for the level-2 loop, S: $\underline{\text{dsv1}}^{(2,0)}_{\downarrow S2}$, $\underline{\text{dsv1}}^{(2,1)}_{\downarrow S2}$, $\underline{\text{dsv1}}^{(2,2)}_{\downarrow S2}$, and so on.... We obtain the following s-definitions after a lengthy derivation.

$$\underline{\text{dsv1}}^{(2,0)}_{\downarrow S2} \equiv \oplus(\text{dsv4}^{(1)}_{\downarrow S}, \text{dsv1}^{(1)}_{\downarrow S}, \text{IMP}^{(1)}_{S})$$
$$\underline{\text{dsv1}}^{(2,1)}_{\downarrow S2} \equiv \oplus(\text{dsv2}^{(1)}_{\downarrow S}, \text{dsv4}^{(1)}_{\downarrow S}, \text{dsv1}^{(1)}_{\downarrow S}, \text{IMP}^{(1)}_{S})$$
$$\underline{\text{dsv1}}^{(2,2)}_{\downarrow S2} \equiv \oplus(\text{dsv3}^{(1)}_{\downarrow S}, \text{dsv2}^{(1)}_{\downarrow S}, \text{dsv4}^{(1)}_{\downarrow S}, \text{dsv1}^{(1)}_{\uparrow S}, \text{IMP}^{(1)}_{S})$$

Each of the above s-definitions gives rise to three s-definitions since $\underline{\text{dsv1}}^{(1)}_{\downarrow S}$ is defined in terms of three s-definitions depending on how many times S2 iterated during the first iteration of S. So, the total number of s-definitions for $\underline{\text{dsv1}}^{(2)}_{\downarrow S}$ (for two iterations of S) is nine.

6.4.5 Run-time support for evaluating s-definitions

At compile-time, while templates are being built for the procedures of an RM, the new flow graph s-definitions need to be generated from the flow graphs of the procedures. To evaluate these s-definition, some run-time support in the access component of an RM is needed.

Let a basic block, B_i, in a procedure, P, be a header of a region that is conditionally entered and that is not included in a loop. We require the access component to maintain a one-bit flag, also denoted by B_i, to indicate whether the region headed by B_i is entered during the execution of P. Then, when any s-definition in which

an s-expression is enclosed in brackets marked by B_i or $\neg B_i$, the run-time flow control mechanism checks the flag B_i and decides whether to evaluate the enclosed s-expression or not.

For a level-1 loop, S, a one-dimensional counter is needed to inform the run-time flow control mechanism about the execution path taken at run-time. The one-dimensional counter, also denoted S, is maintained to indicate the number of times that the body of S is entered. Then, it is checked at run-time to decide which s-definition, among the series of s-definitions associated with the loop, to evaluate. For a basic block, B_j in P, that is included in the loop S, a series of one-bit flags, $B_j^{(k)}$, must be maintained to indicate if B_j is entered at iteration k of S.

When loops are nested, a series of vectors needs to be maintained to indicate the number of iterations at each level of nesting. For a level-n loop, Sn, a series of n-dimensional vectors whose components are counters are needed. Each vector is of the form

$$\underline{k} = (k_1, k_2, \ldots, k_n)$$

where k_i indicates the number of iterations of the level-i loop containing Sn. For a basic block, B_l in P, that is included in a level-n loop, S, a series of one-bit flags, $B_l^{(\underline{k})}$, must be maintained to indicate if B_l is entered at iteration k_n of S, during the iteration k_{n-1} of the enclosing level-(n-1) loop, etc....

The reason a series of vectors is needed rather than just one is that in general, for a given value for a k_i, there are more than one associated value for k_{i+1}. For example, for a 2-dimensional loop counter vector whose first component is m, for m = 1, 2, 3,... , we can associate the vectors

$$(m, 0), (m, 1), (m, 2), \text{etc...}$$

depending on the number of times the level-2 loop iterates. As a result, a question about the number of times that the level-2 loop iterated during a certain iteration of the containing level-1 loop cannot be answered unless all the vectors are maintained.

The maintained series of vectors serves as a *history of execution* for a loop. Through that history, the run-time flow control mechanism can choose the right s-definitions to evaluate.

There is a certain overhead cost associated with maintaining a history of execution for a level-n loop. To avoid such a cost, we can instead require that an updating of the s-assigned dynamic variables to take place at the time control exits from a loop. By doing that, all loops can be treated as level-1 loops and thus, only one-dimensional counters would be needed. That is because by updating the classes at the end of each loop, the results of the loop iterations are saved and a history for future reference is not needed.

Updating classes at the end of loops not only eliminates the overhead of maintaining n-dimensional counters, but it also cuts down on the number of s-definitions that needs to be stored in each template. The reason is again that all loops can be treated as level-1 loops. As a consequence, the only s-definitions that are generated are the ones that deal directly with the loop in question regardless of the nesting of other loops.

For example, assume that in our previous example, we include some update instructions at the end of loop S2. That is, assume we have the following code.

```
S: B3: while (E3: dsv1 > 0) do
   S1: B4: dsv1 = dsv4;
       S2: B5: while (E5: dsv1 > 0) do
```

```
S3: B6: dsv1 = dsv2;
        dsv2 = dsv3;
    UPDATE(dsv1, dsv2);
B7: dsv4 = dsv1;
```

Then, the following s-definitions

$$\underline{\text{dsv1}}^{(1)}_{\downarrow S} \equiv \oplus(\text{dsv1}^{(1)}_{\downarrow S2})$$
$$\underline{\text{dsv1}}^{(2)}_{\downarrow S} \equiv \oplus(\text{dsv1}^{(2)}_{\downarrow S2})$$

need not be expanded further (as we did earlier) to take care of all possible number of iterations for S2. By performing the updating after S2, the current class of dsv1 reflects its correct class regardless of how many iteration of S2 are executed. Then, the current class of dsv1 can be substituted for $\underline{\text{dsv1}}^{(1)}_{\downarrow S2}$ and $\underline{\text{dsv1}}^{(2)}_{\downarrow S2}$, at the time of the evaluation of the s-definitions.

By including updates at the end of each loop, we avoid the cost of maintaining the vectors of counters. However, we incur the cost of the extra updates. Also, we violate the property of our system that states that class updates are part of the information flow control mechanism which is called upon only at message passing time.

As a result, the decision of which scheme to adopt depends, in a given implementation, on how the efficiency of one scheme compares with the efficiency of the other. Thus, we leave the option of the choice of schemes to the implementor.

6.5 Examples

In this section, we present some examples in which the new flow graph s-definitions are used in generated templates.

6.5.1 Example 1

The first example is shown in Figure 6.9. It does not contain any loops. We have labeled the basic blocks in the code, and they are shown in the flow graphs of M1.P and M2.Q in Figure 6.10. The templates of the procedures are shown in Figure 6.11.

Next, we generate s-definitions for some of the s-variables that are shown in the templates. We assume that in M1.P, the nested if-statement is region S2 (B_2, B_3, and B_4) and the outer if-statement is S1 (B_1, S2, and B_5).

In the template of M1.P, we need to expand the s-definitions for $\underline{\text{dsv1}}_{\uparrow F}$, and $\underline{\text{dsv2}}_{\uparrow F}$.

$$\underline{\text{dsv2}}_{\uparrow F} \equiv \oplus(\text{dsv2}_{\downarrow 6})$$
$$\underline{\text{dsv2}}_{\downarrow 6} \equiv \oplus(\text{dsv2}_{\uparrow 6})$$
$$\underline{\text{dsv2}}_{\uparrow 6} \equiv \oplus(\text{dsv2}_{\downarrow S1})$$
$$\underline{\text{dsv2}}_{\downarrow S1} \equiv \text{B}_2[\oplus(\text{dsv2}_{\downarrow S2})] \vee \text{B}_5[\oplus(\text{dsv2}_{\downarrow 5})]$$
$$\underline{\text{dsv2}}_{\downarrow S2} \equiv \text{B}_3[\oplus(\text{dsv2}_{\downarrow 3})] \vee \text{B}_4[\oplus(\text{dsv2}_{\downarrow 4})]$$
$$\underline{\text{dsv2}}_{\downarrow 3} \equiv \oplus(\text{dsv2}_{\downarrow 2}, \text{IMP}_3)$$
$$\underline{\text{dsv2}}_{\downarrow 2} \equiv \oplus(\text{dsv2}_{\downarrow 1})$$
$$\underline{\text{dsv2}}_{\downarrow 1} \equiv \oplus(\text{dsv2}_{\uparrow I})$$
$$\text{IMP}_3 \equiv \oplus(\text{dsv1}_{\downarrow 1}, \text{P.a}, \text{IMP})$$
$$\underline{\text{dsv1}}_{\downarrow 1} \equiv \oplus(\text{dsv1}_{\downarrow I})$$

```
M1 ={ int dsv1;
      int dsv2;
      int ssv1(CONF);
 ...

 P(a | b)
 int a, b(SECRET);
 {
  BI: dsv1 = ssv1;
  B1: if (a == 0) then
  B2:    if (dsv1 == 0) then
  B3:      dsv1 = dsv2;
         else
  B4:      dsv2 = dsv1;
      else
  B5:    dsv2 = a;
  B6: b = dsv2;
 }
}
```

```
M2 = { int dsv3;
       int dsv4;
       int ssv2(SECRET);
  ...

  Q()
  {
   BI: if (ssv2 > 0) then
       {
   B1:    M1.P( dsv3 | dsv4);
   B2:    dsv3 = ssv2;
       }
       else
   B3:    sendprobe(M.P);
  }
}
```

Figure 6.9: Example 1

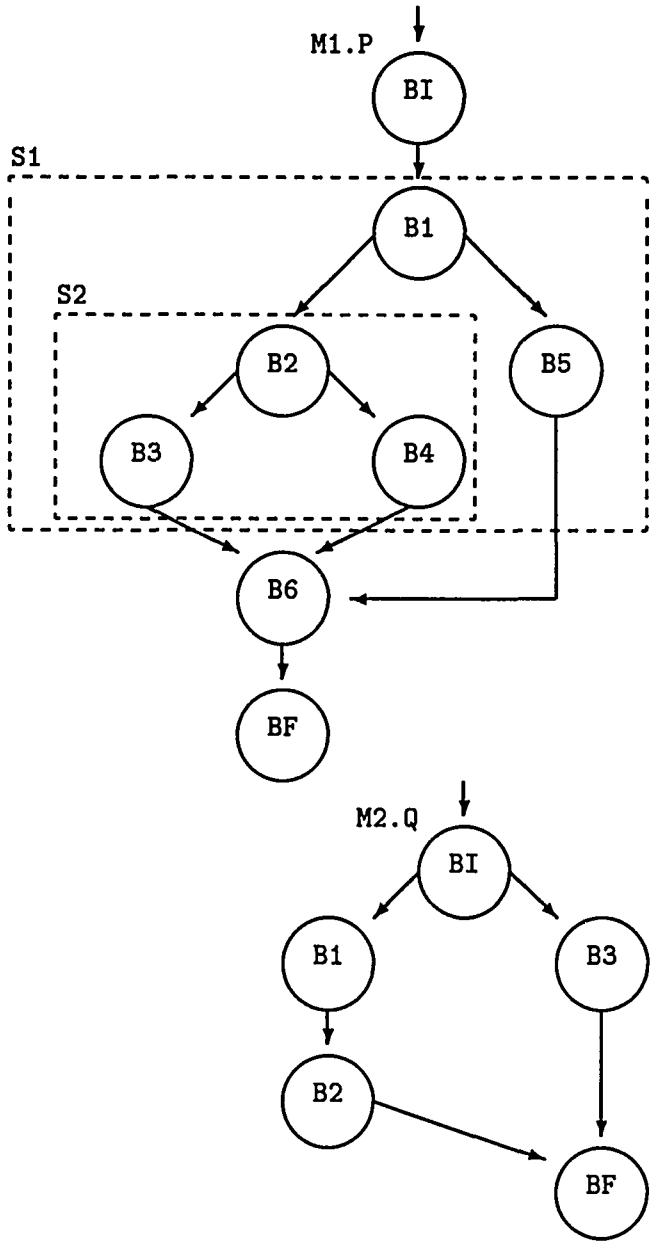

Figure 6.10: Flow graphs for example 1

M1.P(a | b) TEMPLATE

S-definitions Part

OUTS

$\underline{b} \equiv \oplus(\text{SECRET})$

INS

<empty>

STATES

$\underline{dsv1} \equiv \oplus(dsv1_{\uparrow F})$

$\underline{dsv2} \equiv \oplus(dsv2_{\uparrow F})$

S-checks Part

$\underline{b}(\text{SECRET}) \leftarrow \oplus(\text{P.a, SECRET, IMP})$

M2.Q() TEMPLATE

S-definitions Part

OUTS

<empty>

INS

M1.P(dsv3 | dsv4)

$\underline{\text{implicit}} \equiv \oplus(\text{SECRET, IMP})$

$\underline{dsv3}\Uparrow\text{M1.P} \equiv \oplus(dsv3_{\uparrow 1})$

$\underline{dsv4}\Uparrow\text{M1.P} \equiv \oplus(dsv4_{\uparrow 1})$

STATES

$\underline{dsv3} \equiv \oplus(dsv3_{\uparrow F})$

$\underline{dsv4} \equiv \oplus(dsv4_{\uparrow F})$

S-checks Part

<empty>

Figure 6.11: Templates for example 1

$$\underline{\text{dsv1}}_{\downarrow I} \equiv \oplus(\text{SECRET, IMP})$$

$$\underline{\text{dsv2}}_{\downarrow 4} \equiv \oplus(\text{dsv1}_{\uparrow 4}, \text{IMP}_4)$$

$$\underline{\text{dsv1}}_{\uparrow 4} \equiv \oplus(\text{dsv1}_{\downarrow 2})$$

$$\underline{\text{dsv1}}_{\downarrow 2} \equiv \oplus(\text{dsv1}_{\downarrow 1})$$

$$\text{IMP}_4 \equiv \oplus(\text{dsv1}_{\downarrow 1}, \text{P.a, IMP})$$

$$\underline{\text{dsv2}}_{\downarrow 5} \equiv \oplus(\text{P.a, IMP}_5)$$

$$\text{IMP}_5 \equiv \oplus(\text{P.a, IMP})$$

We finally get

$$\underline{\text{dsv2}}_{\uparrow F} \equiv \text{B}_2[\oplus(\text{dsv2}_{\downarrow S2})] \vee \text{B}_5[\oplus(\text{ P.a, IMP})]$$

where

$$\underline{\text{dsv2}}_{\downarrow S2} \equiv \text{B}_3[\oplus(\text{dsv2}_{\uparrow I}, \text{P.a, SECRET, IMP})] \vee \text{B}_4[\oplus(\text{P.a, SECRET, IMP})].$$

Next, we derive $\underline{\text{dsv1}}_{\uparrow F}$.

$$\underline{\text{dsv1}}_{\uparrow F} \equiv \oplus(\text{dsv1}_{\downarrow S1})$$

$$\underline{\text{dsv1}}_{\downarrow S1} \equiv \text{B}_2[\oplus(\text{dsv1}_{\downarrow S2})] \vee \text{B}_5[\oplus(\text{dsv1}_{\downarrow 5}, \text{IMP}_5)]$$

$$\underline{\text{dsv1}}_{\downarrow S2} \equiv \text{B}_3[\oplus(\text{dsv1}_{\downarrow 3})] \vee \text{B}_4[\oplus(\text{dsv1}_{\downarrow 4})]$$

$$\underline{\text{dsv1}}_{\downarrow 3} \equiv \oplus(\text{dsv2}_{\uparrow 3}, \text{IMP}_3)$$

$$\underline{\text{dsv2}}_{\downarrow 3} \equiv \oplus(\text{dsv2}_{\uparrow I})$$

$$\text{IMP}_3 \equiv \oplus(\text{SECRET, P.a, IMP})$$

$$\underline{\text{dsv1}}_{\downarrow 4} \equiv \oplus(\text{dsv1}_{\uparrow 4}, \text{IMP}_4)$$

$$\underline{\text{dsv1}}_{\downarrow 4} \equiv \oplus(\text{SECRET, IMP, P.a})$$

$$\underline{\text{dsv1}}_{\downarrow 5} \equiv \oplus(\text{dsv1}_{\uparrow 5})$$

$$\underline{\text{dsv1}}_{\uparrow 5} \equiv \oplus(\text{dsv1}_{\downarrow 1})$$

$$\underline{\text{dsv1}}_{\downarrow 1} \equiv \oplus(\text{SECRET, IMP})$$

We finally get

$$\underline{\text{dsv1}}_{\uparrow F} \equiv \text{B}_2[\oplus(\text{dsv1}_{\downarrow S2})] \vee \text{B}_5[\oplus(\text{P.a, SECRET, IMP})]$$

where

$$\underline{\text{dsv1}}_{\downarrow S2} \equiv \text{B}_3[\oplus(\text{dsv2}_{\uparrow I}, \text{P.a, SECRET, IMP})] \vee \text{B}_4[\oplus(\text{P.a, SECRET, IMP})].$$

In the template of M2.Q, we need to expand the s-definitions for $\underline{\text{dsv3}}_{\uparrow 1}$, $\underline{\text{dsv4}}_{\uparrow 1}$, $\underline{\text{dsv3}}_{\uparrow F}$, and $\underline{\text{dsv4}}_{\uparrow F}$. The first two s-definitions are simple.

$$\underline{\text{dsv3}}_{\uparrow 1} \equiv \oplus(\text{dsv3}_{\uparrow I})$$
$$\underline{\text{dsv4}}_{\uparrow 1} \equiv \oplus(\text{dsv4}_{\uparrow I})$$

The s-definitions for the final class of dsv3 is more involved.

$$\underline{\text{dsv3}}_{\downarrow F} \equiv \text{B}_1[\oplus(\text{dsv3}_{\downarrow 2})] \vee \text{B}_3[\oplus(\text{dsv3}_{\downarrow 3})]$$
$$\underline{\text{dsv3}}_{\downarrow 2} \equiv \oplus(\text{SECRET, IMP}_2)$$
$$\text{IMP}_2 \equiv \oplus(\text{SECRET, IMP})$$
$$\underline{\text{dsv3}}_{\downarrow 3} \equiv \oplus(\text{dsv3}\Downarrow\text{M.P})$$

We finally get

$$\underline{\text{dsv3}}_{\downarrow F} \equiv \text{B}_1[\oplus(\text{SECRET, IMP})] \vee \text{B}_3[\oplus(\text{dsv3}\Downarrow\text{M.P})].$$

The s-definitions for the final class of dsv4 is as follows.

$$\underline{\text{dsv4}}_{\downarrow F} \equiv \text{B}_1[\oplus(\text{dsv4}_{\downarrow 2})] \vee \text{B}_3[\oplus(\text{dsv4}_{\downarrow 3})]$$
$$\underline{\text{dsv4}}_{\downarrow 2} \equiv \oplus(\text{dsv4}_{\uparrow 2})$$

$$\underline{\text{dsv4}}_{\uparrow 2} \equiv \oplus(\text{dsv4}_{\downarrow 1})$$

$$\underline{\text{dsv4}}_{\downarrow 1} \equiv \ominus(\text{M.P.dsv4, IMP}_1)$$

$$\text{IMP}_1 \equiv \oplus(\text{SECRET, IMP})$$

$$\underline{\text{dsv4}}_{\downarrow 3} \equiv \oplus(\text{dsv3}\Downarrow\text{M.P})$$

We finally get

$$\underline{\text{dsv4}}_{\downarrow F} \equiv \text{B}_1[\oplus(\text{M.P.dsv4, SECRET, IMP})] \vee \text{B}_3[\oplus(\text{dsv4}\Downarrow\text{M.P})].$$

Assume that M2.Q is executing under an implicit flow of LOW. Assume also that ssv2 is greater than 0, and that dsv3 and dsv4 have classes CONFIDENTIAL and TOPSECRET, respectively.

When M1.P is called, the outgoing implicit flow is evaluated to SECRET. The class of the actual IN parameter is evaluated to CONFIDENTIAL. Finally, since neither dsv3 nor dsv4 were modified before the call, then there is no need to update them. (Actually, the run-time information flow control mechanism checks that no write-locks are acquired for dsv3 or dsv4.)

When the RPC M1.P is received at M1, an instance of the template is generated, P.a is replaced by CONFIDENTIAL and the s-check is evaluated. No flow violations are detected and the execution of M2.Q begins. Note that the s-check still checks for the worst case of flows into b. The s-expressions in the s-checks have not changed. Any program that has a potential of being insecure is rejected, regardless of the actual execution path.

The classes of dsv1 and dsv2 are updated at the end of the procedure. The new classes depend on the execution path that is taken in M.P. For example, assume that a and dsv1 are found to be 0. Then,

$$\underline{\text{dsv1}}_{\downarrow S2} \equiv B_3[\oplus(\text{dsv2}_{\uparrow I}, \text{P.a, SECRET, IMP})] \vee B_4[\oplus(\text{P.a, SECRET, IMP})].$$

evaluates to

$$\underline{\text{dsv1}}_{\downarrow S2} \equiv \oplus(\text{dsv2}_{\uparrow I}, \text{P.a, SECRET, IMP})$$

which is

$$\underline{\text{dsv1}}_{\downarrow S2} \equiv \oplus(\text{CONFIDENTIAL, SECRET, CONFIDENTIAL})$$

assuming that $\underline{\text{dsv2}}_{\uparrow I}$ is LOW. This results in

$$\underline{\text{dsv1}}_{\uparrow F} \equiv B_2[\oplus(\text{dsv1}_{\downarrow S2})] \vee B_5[\oplus(\text{P.a, SECRET, IMP})]$$

evaluating to

$$\underline{\text{dsv1}}_{\uparrow F} \equiv \oplus(\underline{\text{dsv1}}_{\downarrow S2}), \text{ and}$$
$$\underline{\text{dsv1}}_{\downarrow S2} \equiv \oplus(\text{SECRET}).$$

Thus, the final class of dsv1 is SECRET.

6.5.2 Example 2

The second example is shown in Figure 6.12. The procedure M1.P contains a loop. The templates of all the procedures are in Figure 6.13. The flow graph of procedure M1.P is shown in Figure 6.14. The other flow graphs are trivial.

Next, we expand the s-definitions that appear in the templates. We assume that in M1.P, the while-statement and the if-statement form regions S1 and S2, respectively. In the template of M1.P, we need to expand the s-deflnitions for $\underline{\text{dsv1}}^{(k)}_{\uparrow 3}$, $\underline{\text{dsv2}}^{(k)}_{\uparrow 2}$, $\underline{\text{dsv2}}^{(k)}_{\uparrow 3}$, $\underline{\text{dsv1}}_{\uparrow F}$, and $\underline{\text{dsv2}}_{\uparrow F}$. We start with $\underline{\text{dsv1}}^{(k)}_{\uparrow 3}$. The case of no

iterations of S1 is not handled, since then neither the call nor the probe to M2.Q are sent anyway.

Case 1: One iteration of S1.

$$\underline{\text{dsv1}}^{(1)}_{\uparrow 3} \equiv \oplus(\text{dsv1}^{(1)}_{\uparrow 1})$$
$$\underline{\text{dsv1}}^{(1)}_{\uparrow 1} \equiv \oplus(\text{dsv1}_{\uparrow I})$$

Case 2: Two iterations of S1.

$$\underline{\text{dsv1}}^{(2)}_{\uparrow 3} \equiv \oplus(\text{dsv1}^{(1)}_{\downarrow S1})$$
$$\underline{\text{dsv1}}^{(1)}_{\downarrow S1} \equiv \oplus(\text{dsv2}^{(1)}_{\uparrow 5})$$
$$\underline{\text{dsv2}}^{(1)}_{\uparrow 5} \equiv \oplus(\text{dsv1}\Downarrow\text{M.Q})$$

Case 3: Three iterations of S1.

$$\underline{\text{dsv1}}^{(3)}_{\uparrow 3} \equiv \oplus(\text{dsv1}^{(2)}_{\downarrow S1})$$
$$\underline{\text{dsv1}}^{(2)}_{\downarrow S1} \equiv \oplus(\text{dsv1}\Downarrow\text{M.Q})$$

We expand only $\underline{\text{dsv2}}^{(k)}_{\uparrow 2}$, noting that $\underline{\text{dsv2}}^{(k)}_{\uparrow 3}$ has the same s-definition.

Case 1: One iteration of S1.

$$\underline{\text{dsv2}}^{(1)}_{\uparrow 2} \equiv \oplus(\text{dsv2}^{(1)}_{\uparrow 1})$$
$$\underline{\text{dsv2}}^{(1)}_{\uparrow 1} \equiv \oplus(\text{dsv2}_{\downarrow 1})$$
$$\underline{\text{dsv2}}_{\downarrow 1} \equiv \oplus(\text{LOW, IMP})$$

Case 2: Two iterations in S1.

$$\underline{\text{dsv2}}^{(2)}_{\uparrow 2} \equiv \oplus(\text{dsv2}^{(1)}_{\downarrow S1})$$
$$\underline{\text{dsv2}}^{(1)}_{\downarrow S1} \equiv \oplus(\text{dsv2}^{(1)}_{\downarrow 5})$$

```
M1 ={ int dsv1;
      int dsv2;
      int ssv1(CONF);
 ...

 P()
 {  int i(LOW);

 BI: i = 0;
     dsv2 = 0;
 B1: while (i <= 1) do {
 B2:    if (dsv2 == 0) then
 B3:       M2.Q();
        else
 B4:       sendprobe(M2.Q);
 B5:    dsv2 = ssv1;
        ssv1 = dsv1;
        i++; }
 } /* P */
```

```
      R(a) /* M1.R */
      int a;
      {
      BI: if (a <> 0) then
      B1:    dsv1 = a;
      }
     } /* M1 */

M2 = { int dsv3;
       int ssv2(SECRET);
   ...

   Q()
   {
    BI:    M1.R(dsv3);
    B1:    dsv3 = ssv2;
   }
  } /* M2 */
```

Figure 6.12: Example 2

M1.P() TEMPLATE

S-definitions Part

INS

M2.Q()

$\underline{\text{implicit}} \equiv \oplus(\text{dsv2}^{(k)}_{\uparrow 2}, \text{LOW}, \text{IMP})$

$\underline{\text{dsv1}}\Uparrow\text{M2.Q} \equiv \oplus(\text{dsv1}^{(k)}_{\uparrow 3})$

$\underline{\text{dsv2}}\Uparrow\text{M2.Q} \equiv \oplus(\text{dsv2}^{(k)}_{\uparrow 3})$

STATES

$\underline{\text{dsv1}} \equiv \oplus(\text{dsv1}_{\uparrow F})$

$\underline{\text{dsv2}} \equiv \oplus(\text{dsv2}_{\uparrow F})$

S-checks Part

$\underline{\text{i}}(\text{LOW}) \leftarrow \oplus(\text{LOW}, \text{IMP})$

$\underline{\text{ssv1}}(\text{CONFIDENTIAL}) \leftarrow \oplus(\text{dsv1}\Downarrow\text{M2.Q}(*), \text{LOW}, \text{IMP})$

M1.R() TEMPLATE

S-definitions Part

STATES

$\underline{\text{dsv1}} \equiv \oplus(\text{R.a}, \text{IMP})$

M2.Q() TEMPLATE

S-definitions Part

INS

M1.R(dsv3)

$\underline{\text{implicit}} \equiv \oplus(\text{IMP})$

$\underline{\text{dsv3}}\Uparrow\text{M1.R} \equiv \oplus(\text{dsv3}_{\uparrow I})$

STATES

$\underline{\text{dsv3}} \equiv \oplus(\text{SECRET}, \text{IMP})$

Figure 6.13: Example 2 templates (omitted parts are empty)

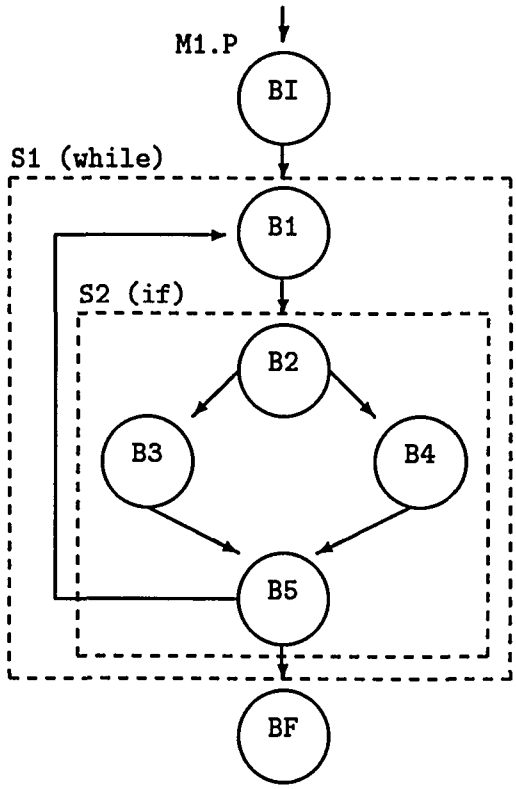

Figure 6.14: Flow graphs for example 2

$$\underline{\text{dsv2}}^{(1)}_{\uparrow 5} \equiv \oplus(\text{CONFIDENTIAL, IMP}^{(1)}_{5})$$
$$\text{IMP}^{(1)}_{5} \equiv \oplus(\text{IMP}^{(1)}_{S})$$
$$\text{IMP}^{(1)}_{S} \equiv \oplus(\text{LOW, IMP})$$

Finally, we get

$$\underline{\text{dsv1}}^{(2)}_{\downarrow 2} \equiv \oplus(\text{CONFIDENTIAL, IMP}).$$

Case 3: Three iterations of S1.

$$\underline{\text{dsv2}}^{(3)}_{\uparrow 2} \equiv \oplus(\text{dsv2}^{(2)}_{\downarrow S1})$$
$$\underline{\text{dsv2}}^{(2)}_{\downarrow S1} \equiv \oplus(\text{dsv2}^{(2)}_{\downarrow 5})$$
$$\underline{\text{dsv2}}^{(2)}_{\uparrow 5} \equiv \oplus(\text{CONFIDENTIAL, impn52})$$
$$\text{IMP}^{(2)}_{5} \equiv \oplus(\text{LOW, IMP})$$

Finally, we get

$$\underline{\text{dsv2}}^{(3)}_{\downarrow 2} \equiv \oplus(\text{CONFIDENTIAL, IMP}).$$

We similarly expand $\underline{\text{dsv1}}_{\uparrow F}$ and $\underline{\text{dsv2}}_{\uparrow F}$ with the following results,

$$\underline{\text{dsv1}}_{\uparrow F} \equiv \oplus(\text{dsv1}^{(k)}_{\downarrow S1}) \text{ and}$$
$$\underline{\text{dsv2}}_{\uparrow F} \equiv \oplus(\text{dsv2}^{(k)}_{\downarrow S1}),$$

where we have

$$\underline{\text{dsv1}}^{(0)}_{\downarrow S1} \equiv \oplus(\text{dsv1}_{\uparrow I}) \text{ and}$$
$$\underline{\text{dsv2}}^{(0)}_{\uparrow S1} \equiv \oplus(\text{LOW, IMP}).$$

Now, assume that M1.P is called under an implicit flow of LOW. Also, assume, in module M1, that dsv2 has an initial class of LOW and a value of 0, that dsv1 has class LOW, and that ssv1 has a nonzero value. In module M2, assume that dsv3 is nonzero with class CONFIDENTIAL.

When the s-check in M1.P is evaluated, it shows a flow from LOW to CONFIDENTIAL, and the call is allowed to proceed. When M2.Q is called from M1.P during the first iteration of S1, the s-definitions with superscript (1) are evaluated. The class of dsv2 is updated to LOW and the class of dsv1 is unchanged and remains LOW. Also, implicit is evaluated to LOW.

M2.Q has no s-checks and proceeds to call M1.R. The implicit flow sent into M1.R is LOW and dsv3 is unchanged and remains CONFIDENTIAL. Dsv3 is also the actual parameter.

M1.R also does not have any s-checks. The value of a is nonzero and R.a is CONFIDENTIAL. The call proceeds and writes into dsv1. Upon returning, it updates the class of dsv1 to CONFIDENTIAL.

Note that the call M1.P is module-recursive since it resulted in the call M1.R which is to the same module. Actually, M1.R does indeed modify dsv1 which is also accessed by M1.P.

When control returns to M2.Q, dsv2 is modified and its class updated to SECRET. Then, a return to M1.P is performed. Upon receiving the return message the s-check in M1.P is evaluated from

$$\underline{ssv1}(\text{CONFIDENTIAL}) \leftarrow \oplus(dsv1 \Downarrow M2.Q(*))$$

to

$$\underline{ssv1}(\text{CONFIDENTIAL}) \leftarrow \oplus(dsv1 \Downarrow M2.Q(*), \text{CONFIDENTIAL}).$$

The s-expression reflects the change in $\underline{\text{dsv1}}$ after the call to M2.Q. No flow violations are detected and the execution of the procedure continues at basic block B_5.

At the second iteration of S1, the call to M2.Q is not made since dsv2 becomes nonzero after the assignment in B_5. However, the probe is sent to M2.Q with an implicit flow of

$$\underline{\text{implicit}} \equiv \oplus(\text{dsv2}^{(2)}_{\uparrow 2}, \text{LOW}, \text{IMP})$$

which evaluates to CONFIDENTIAL according to the s-definition of $\underline{\text{dsv2}}^{(2)}_{\uparrow 2}$.

When the probe is received at M2.Q, dsv3 is s-incremented by CONFIDENTIAL, but remains SECRET. Then, the probe is propagated to M1.R where dsv1 is s-incremented and also remains CONFIDENTIAL. Finally, a probe-return is sent to M1.P. Since dsv1 is still CONFIDENTIAL, the s-check in M1.P detects no violations and the procedure is allowed to resume. Finally, upon returning, M1.P updates the class of dsv2 to CONFIDENTIAL.

In the above scenario, no flow violations were detected. However, with a simple modification of the assumptions, we show an example in which execution of M1.P must halt because of a violation.

Instead of assuming that the initial value of ssv1 is nonzero, let us assume that it is zero. That causes M2.Q to be called during the second iteration of S1.

When M2.Q executes, it calls M2.R with an actual parameter that has class SECRET. M2.R updates the class of dsv1 according to the class of the parameter, resulting in $\underline{\text{dsv1}}$ being SECRET.

When eventually a return is sent to M1.P, the s-check

$$\underline{\text{ssv1}}(\text{CONFIDENTIAL}) \leftarrow \oplus(\text{dsv1}\Downarrow\text{M2.Q}(*), \text{CONFIDENTIAL})$$

is evaluated to

$$\underline{ssv1}(\text{CONFIDENTIAL}) \leftarrow \oplus(\text{dsv1}\Downarrow\text{M2.Q}(*), \text{SECRET}, \text{CONFIDENTIAL}).$$

That is a clear flow violation which causes the execution of the process to halt.

6.6 Summary

In this section, we summarize the new scheme to compute classes of dynamic variables by revisiting the compile-time information flow control mechanism.

As explained in Chapter 4, the main task of the compile-time mechanism is to produce a template for each procedure in an RM. The changes that are made in this chapter are confined to the generation of s-definitions in the templates; they do not affect the generation of s-checks. The changes are summarized as follows.

We assume that the compile-time mechanism is producing a template for a procedure M.P. The compile-time mechanism starts by dividing the code into basic blocks. Any message-sending instruction is assumed to form a basic block by itself. Then, the mechanism proceeds by generating a flow graph for M.P[6].

For a dynamic variable dsv, a dsv basic block is one that s-assigns or may s-assign dsv. Dsv basic blocks could be of any of four types: assignment unambiguous, call unambiguous, call ambiguous, or probe unambiguous.

For each dynamic variable dsv, the compile-time mechanism needs to generate an s-definition that gives $\underline{dsv}$ right prior to the execution of a message-sending instruction. In other words, for a message-sending basic block B_i, the mechanism is to

[6]Note that these first steps must also taken by many optimizing compilers. The various algorithms to generate flow graphs and to determine loops are presented in [1].

generate an s-definition for $\underline{\text{dsv}}_{\uparrow i}$. The s-definition is placed in the template in the INS, OUTS, or STATES part, depending on whether the message is a call, a probe, or a return.

The form of the generated s-definition for $\underline{\text{dsv}}_{\uparrow i}$ depends on the position of B_i in the flow graph. We define the s-definition by structural induction on the type of statements that can logically precede B_i. The statements that precede B_i are the ones that potentially define the class of dsv. Assume that those statements form a region S. Then, we have

$$\underline{\text{dsv}}_{\uparrow i} \equiv \oplus(\text{dsv}_{\downarrow S}).$$

The basis of the induction is when B_i is preceded by S which consists of a single basic block B_j. Then, we have the following cases.

1. If B_j is not a dsv basic block, then we have

$$\underline{\text{dsv}}_{\downarrow j} \equiv \oplus(\text{dsv}_{\uparrow j}).$$

2. If B_j is an assignment dsv basic block, then we have

$$\underline{\text{dsv}}_{\downarrow j} \equiv \oplus(\text{dsv-exp}_j, \text{IMP}_j),$$

   where $\underline{\text{dsv-exp}_j}$ is an s-variable representing the class of the last expression assigned to dsv in B_j.

3. If B_j is a call unambiguous dsv basic block, then we have

$$\underline{\text{dsv}}_{\downarrow j} \equiv \oplus(\text{M.Q.dsv}, \text{IMP}_j)$$

where M.Q is the procedure being called in B_j.

4. If B_j is a call or probe ambiguous dsv basic block, then we have

$$\underline{\text{dsv}}_{\downarrow j} \equiv \oplus(\text{dsv}\Uparrow\text{M.Q})$$

where M.Q is the procedure being called or probed in B_j.

In the induction step, S can be any one of the control structures in our assumed syntax: compound, if-then, if-then-else and while statements. Then, we have the following cases.

1. If S is a compound statement (S1; S2) then we have

$$\underline{\text{dsv}}_{\downarrow S} \equiv \oplus(\text{dsv}_{\downarrow S2}).$$

2. If S is an if-then statement, whose body of code forms statement S1 with header $B_{I_{S1}}$ and whose condition is E_l, then we have

$$\underline{\text{dsv}}_{\downarrow S} \equiv B_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \neg B_{I_{S1}}[\oplus(\text{dsv}_{\uparrow S1})]$$

or

$$\underline{\text{dsv}}_{\downarrow S} \equiv B_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee \neg B_{I_{S1}}[\oplus(\text{dsv}_{\uparrow S1}, \text{IMP}_S, E_l)]$$

depending on whether S1 contains any unambiguous dsv basic blocks.

3. If S is an if-then-else statement, whose bodies of code form statements S1 and S2 and whose condition is E_l, then we have

$$\underline{\text{dsv}}_{\downarrow S} \equiv B_{I_{S1}}[\oplus(\text{dsv}_{\downarrow S1})] \vee B_{I_{S2}}[\oplus(\text{dsv}_{\downarrow S2})]$$

or

$$\underline{\mathrm{dsv}}_{\downarrow S} \equiv \mathrm{B}_{I_{S1}}[\oplus(\mathrm{dsv}_{\downarrow S1})] \vee \mathrm{B}_{I_{S2}}[\oplus(\mathrm{dsv}_{\downarrow S2}, \mathrm{IMP}_S, \mathrm{E}_l)]$$

or

$$\underline{\mathrm{dsv}}_{\downarrow S} \equiv \mathrm{B}_{I_{S1}}[\oplus(\mathrm{dsv}_{\downarrow S1}, \mathrm{IMP}_S, \mathrm{E}_l)] \vee \mathrm{B}_{I_{S2}}[\oplus(\mathrm{dsv}_{\downarrow S2})]$$

depending on whether S1 or S2 contain any unambiguous dsv basic blocks.

4. If S is a level-n while statement, then S is treated as a series of if-then statements, $\mathrm{S}^{(\underline{k})}$. Each $\mathrm{S}^{(\underline{k})}$ has a body of code $\mathrm{S1}^{(\underline{k})}$ with header $\mathrm{B}_{I_{S1}}^{(\underline{k})}$ and a condition $\mathrm{E}_l^{(\underline{k})}$. We have

$$\underline{\mathrm{dsv}}_{\downarrow S}^{(\underline{k})} \equiv \oplus(\mathrm{dsv}_{\downarrow S1}^{(\underline{k})}).$$

When simulating iterations of a loop, S, to generate s-definitions, the algorithm halts when a generated s-definition is equivalent to a previously generated one. The following theorem is proven in [27] and it puts an upper bound on the number of needed simulations.

Theorem: The number of needed simulations of iterations for S has a finite bound which is a function of the number of dynamic and static variables that receive direct flows in S, the number of s-variables in the procedure being processed, and the maximum path length of the security lattice.

6.7 Precision of the Modified Flow Control Mechanism

In this section, we prove some properties of the new flow control mechanism. First, we show that the new scheme for updating dynamic state variables is precise.

In other words, whenever the current class of a dynamic state variable is updated at message-sending time, the new class reflects the precise class of the information contained in the variable.

Second, we show that the new precision in updating classes of dynamic state variable, leads to a more precise security mechanism. In other words, the flow control mechanism rejects less secure programs than without the new class updating scheme.

6.7.1 Precision of the class updating scheme

Let dsv be a dynamic state variable in an RM M. Assume that a procedure M.P is called and that its template is instantiated. Assume that at some point during the execution of P, the current class of dsv, $\underline{dsv}$, is updated by the run-time information flow control mechanism, which uses the new flow graph s-definitions and has the access component providing the run-time support for evaluating those s-definitions.

Theorem:

After the updating, $\underline{dsv}$ reflects the precise security class of the information contained in dsv. In other words, dsv is neither underclassified nor overclassified.

Proof (by construction):

The proof is by induction. It is also very simple and obvious at this point since it follows exactly the construction of the flow graph s-definitions in Section 6.4. The construction is also summarized in the previous section. Here, we will not repeat the details that appeared in those sections.

□

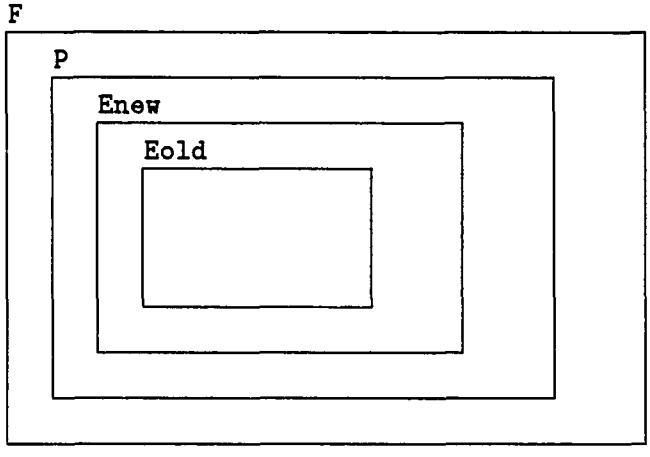

Figure 6.15: Precision of our security mechanism

6.7.2 Precision of the security mechanism

We define the concept of precision as defined in [9]. Let F be the set of all possible information flows in a system striped from an information flow control mechanism. Let P be the F subset of all information flows that are allowed by our flow policy: all flows that follow the lattice structure of the security classes. Let NEWSYS denote our system which includes the flow control mechanism with the new class updating mechanism that uses the flow graph s-definitions and let E_{new} be the set of all allowable flows in NEWSYS. Finally, let OLDSYS denote a system that uses the old form of s-definitions and let E_{old} be the set of all allowable flows in OLDSYS. In Figure 6.15, we show the diagrams for the various sets.

Since the new class updating mechanism does not affect the way s-checks are generated or evaluated, it may be surprising that the figure shows that E_{old} is strictly included in E_{new}. It would be expected that NEWSYS and OLDSYS have the same degree of precision since the same s-checks are being evaluated in both systems.

However, we claim that NEWSYS is strictly more precise than OLDSYS. We prove that fact in the following theorem. The main idea is that since NEWSYS avoids overclassifying dynamic state variables that may appear in s-expression of s-checks, then more programs are accepted by NEWSYS than by OLDSYS.

Theorem: $E_{old} \subset E_{new}$.

Proof:

First, we show that all flows that are allowed by OLDSYS are also allowed by NEWSYS. For a flow to be allowed by either of the two systems, there must be a collection of s-checks that do not detect any flow violations when evaluated. Let

$$\underline{\text{sv}}(\text{SC}) \leftarrow \oplus(\text{s-exp})$$

be such an s-check, where SC is a security class and s-expr is an s-expression. Let $\underline{\text{s-expr}}^n$ and $\underline{\text{s-expr}}^o$ be the class values of s-expr if evaluated by NEWSYS and OLDSYS, respectively.

If s-expr does not contain any dynamic variables, then it must be the case that $\underline{\text{s-expr}}^n$ and $\underline{\text{s-expr}}^o$ represent the same security class. Therefore, if OLDSYS allows the flow represented by the s-check (i.e., it finds $\underline{\text{s-exp}} \leq \text{SC}$), NEWSYS must allow it too.

If s-expr contains a dynamic state variable, dsv, then it must be the case that

$$\underline{\text{s-expr}}^n \leq \underline{\text{s-expr}}^o$$

since the current class of dsv is always precise in NEWSYS and may be overclassifled in OLDSYS. Therefore, again in this case, if OLDSYS allows the flow represented by the s-check, NEWSYS must allow it too.

We conclude that any flow allowed by OLDSYS is also allowed by NEWSYS and that $E_{old} \subseteq E_{new}$.

Next, we show, by a simple example, that there are flows that are allowed by NEWSYS but not allowed by OLDSYS. Consider the following piece of code.

```
int dsv;
int ssv1(LOW);
int ssv2(TOPSECRET);
...

dsv = 0;
if (dsv == 0) then
B3:  dsv = ssv1;
else
B4:  dsv = ssv2;
M.Q();
ssv1 = dsv;
```

Note that ssv2 never flows into dsv. The OLDSYS s-definition for $\underline{\text{dsv}}\Uparrow$M.Q is

$$\underline{\text{dsv}}\Uparrow\text{M.Q} \equiv \oplus(\text{LOW, TOPSECRET, IMP}).$$

When M.Q is called, the current class of dsv is overclassified to TOPSECRET. Assuming that M.Q does not cause an s-assignment of dsv, when the call returns, the s-check

$$\underline{\text{ssv1}}(\text{LOW}) \leftarrow \oplus(\text{dsv}\Downarrow\text{M.Q, IMP}).$$

is evaluated. A flow violation is detected when $\underline{dsv}\Downarrow$M.Q is replaced with TOPSECRET.

On the other hand, the NEWSYS s-definition is

$$\underline{dsv}\Uparrow\text{M.Q} \equiv B_3[\oplus(\text{LOW, IMP})] \vee B_4[\oplus(\text{TOPSECRET, IMP})].$$

Since B_4 is never entered, the class of dsv is updated to LOW (assuming IMP is LOW). When the call returns, the same above s-check is evaluated and no flow violations are detected.

We conclude that the flows specified by the above code are in E_{new} but not in E_{old}. Therefore, $E_{new} \neq E_{old}$.

Since we already proved that $E_{old} \subseteq E_{new}$, we conclude that

$$E_{old} \subset E_{new}.$$

□

Figure 6.15 makes some other claims also. In it, we claim that our information flow control mechanism is secure. In other words, we have

$$E_{new} \subset P.$$

A formal proof of this claim is well beyond the scope of this research. However, we argue that we have constructed the s-checks so as to show all possible flows into a static variable. Therefore, by evaluating them, the mechanism can check all the flows and reject any that are potentially insecure.

Another claim is that our mechanism is not precise. In other words, we have

$$E_{new} \neq P.$$

This is obvious since the s-expression in the s-checks represent all possible flows into a static variable and not just the ones that actually occur during run time. For an example of a program which is in E_{new} but not in P, consider the code of the previous example but with the call to M.Q omitted. Then, the s-check is as follows

$$\underline{\text{ssv1}}(\text{LOW}) \leftarrow \oplus(\text{LOW, TOPSECRET, IMP}).$$

The flow is rejected in NEWSYS, even though, the flow from ssv2 to dsv1 does not occur.

We should note that it has been proven that it is theoretically impossible to have a secure mechanism that is also precise [9, 18].

7. PROBE IMPLEMENTATION ISSUES

In this chapter, we discuss implementation issues that pertain to probes and their efficiency.

7.1 Analysis of Probe Performance

When we discussed our locking system, we explained that the efficiency of probes was a major concern. For that reason, our locking scheme was designed specifically to improve the performance of probes. We allowed (through the probe-locks) different probes to execute concurrently within the same RM.

However, depending on how busy, wide and large a system is, probes may still cause the overall performance of the system to decrease. Probes involve sending messages and they may be numerous.

In the worst case, if a system contains n objects, a single probe may end up visiting each one of those n objects. That may become intolerable if several probes are active under worst case scenarios.

In practice, it may be rare to find objects that call upon *all* other objects. However, it may be common for objects to call upon several distant objects.

We have developed some ideas that may help in further improving the performance of probes. In the rest of this chapter, we present two such ideas.

7.2 Parallel Sending of Probes

In many cases, a single procedure may need to send several probes. For example, consider the following code.

```
while (e) do {
   M1.P();
   M2.Q();
   M3.R();
}
sendprobe(M1.P);
sendprobe(M2.Q);
sendprobe(M3.R);
```

There are several probes that need to be sent at the time control exits the loop. Instead of sending the probes in a synchronous manner one at a time, the sendprobe instructions can be implemented so as to allow successive probes to be sent concurrently. Such a scheme would take full advantage of the locking mechanism allowing probes to execute concurrently.

7.3 Probe Tree Computation

In Chapter 4, we observed that the path of a propagated probe and the procedures of objects it visits form a probe tree. The root of a probe tree is the initial probe recipient. The descendants of the root are probe recipients to which the probe is propagated.

Probe trees are determined by their root. That is because, once an initial probe recipient receives a probe, it always propagates it to the same modules and procedures. Those modules are determined by the INS part of the template of the initial probe recipient. Unless that INS part changes, the paths taken by all probes propagated by a given initial probe recipient are the same.

For a given probe and initial probe recipient, the probe tree is traced when the probe is sent the first time. For that reason, we suggest to implement probes so as to compute the probe tree when it is traced the first time. The result of the probe tree computation should be stored with its root.

A probe checks the implicit flow it carries against all static variables that receive flow in a probe recipient. Also, a probe s-increments dynamic state variables in the module of the probe recipient. Therefore, computing the probe tree involves storing several pieces of information with the initial probe recipient:

1. All the probe recipients that form the tree,
2. the classes of all the static variables in the probe recipients that need to be checked, and
3. all the dynamic state variables that need to be s-incremented in the probe recipients.

Assume that the information of the probe tree of a given initial probe recipient, IR, is computed and stored after the first probe is propagated by IR. Then, once a subsequent probe is received by IR, it can locally perform all flow checks that the probe needs to perform at various probe recipients.

If a flow violation is detected, the probe is sent with a violation message to the probe originator. In that case, we have saved the step of propagating the probes that in any case, would have resulted in a flow violation.

If no flow violations are detected, IR needs to send s-increment messages to have the dynamic state variables (of which it keeps a list) s-incremented. Those messages need only be sent to the modules of the probe recipients that contain dynamic state variables that need to be s-incremented. After a successful completion of the s-incrementing step, a probe-certified message may be sent to the probe originator which can resume execution.

The above scheme works fine unless the probe tree information that is stored becomes outdated. If a change occurs in a template of any probe recipient, the s-checks that need to be evaluated and the dynamic state variables that need to be s-incremented may change also. For the above scheme to continue to perform correctly even in case probe tree information is modified, the keeper of such information needs to be informed when a change occurs in any of the templates of the procedures that form the probe tree. Once a change is detected, the initial probe recipient cannot rely on the its stored information to correctly check a probe locally. The old probe tree needs to be discarded and a new probe tree needs to be computed next time a probe is sent.

In the worst case, the whole of the probe tree needs to be recomputed. However, in certain cases, recomputing the probe tree may not involve updating all of its stored information. The only updates that need to be performed are to the descendant of the probe recipient whose template changes and causes the recomputation of of the probe tree. The other parts of the tree that are not connected to the modified recipient

need not be affected.

The next problem to tackle is explaining how the initial probe tree recipient is informed of a change in a template in its probe tree. It cannot be the responsibility of the modified recipient to inform all of its callers about the change. That is simply because it may not know about all of its callers. Even if it does (if it keeps record of all the modules that were sent any probe tree information), its callers may be too numerous; It may be too inefficient to individually inform them of the change.

A better solution is to have the keeper of the probe tree information responsible for making sure that that information is still valid before it is used. That check can be done at the time the probe is sent.

One way to implement this solution is to assume the existence of a distributed database that is used to look up procedure names and bind them to a remote RM [6]. Then, for each procedure in the database, we associate a monotonically increasing version number. Whenever a procedure template is modified, the version number is incremented in the database.

A keeper of probe tree information can use the version number as follows. It keeps the version number along with the probe tree information. The kept version number corresponds to the number found in the database at the time the stored probe tree is computed. Any time the probe tree information is to be used, the initial probe tree recipient must compare the stored version number against the current one that appears in the database. Only if the two version numbers differ, does the initial recipient decide to rebuild the tree. Otherwise, it can safely use the probe information knowing that it is current.

The above solution may not be the best one. By keeping a version number for

a procedure, we are binding the caller to a certain version of the callee. That runs contrary to keeping all name resolution and binding dynamic. However, at this point, we know of no other solution to solve the problem of template modification after a probe tree is built.

7.4 Example

To end this chapter, we look at an example. Consider the code in Figure 7.1. Assume that the first probe from M1.P (the probe originator) to M2.Q (the initial probe recipient) is sent. That first probe is propagated to other recipients as follows: from M2.Q to M3.R and M4.S, and from M3.R to M5.T. During the message exchange caused by that first probe, the respective modules send back the probe tree information to M2.Q (the root).

The probe tree information consists of the three other probe recipients that form the descendants of the tree root. Along with each of those descendants, the following information are stored with M2.Q.

1. Recipient M3.R:
   (a) Static variables that receive flow in the procedure: ssv1(SECRET).
   (b) Dynamic state variables that need to be s-incremented: dsv1.
2. Recipient M4.S:
   (a) Static variables that receive flow in the procedure: none.
   (b) Dynamic state variables that need to be s-incremented: dsv2.
3. Recipient M5.T:

```
M1 ={ ...

 P()
 {
    ...
    if (a) then
       M2.Q()
    else
       sendprobe(M2.Q);
 }
}

M2 ={ ...

 Q()
 {
    M3.R();
    M4.S();

 }
}
```

```
M3 = { int dsv1;
       int ssv1(SECRET);
  ...

  R()
  {
     M5.T();
     dsv1 = ...;
     ssv1 = ...;
  }
}

M4 = { int dsv2;
  ...

  S()
  {
     dsv2 = ...;
  }
}

M5 = { int ssv2(LOW);
  ...

  T()
  {
     ssv2 = ...;
  }
}
```

Figure 7.1: Example for computing a probe tree

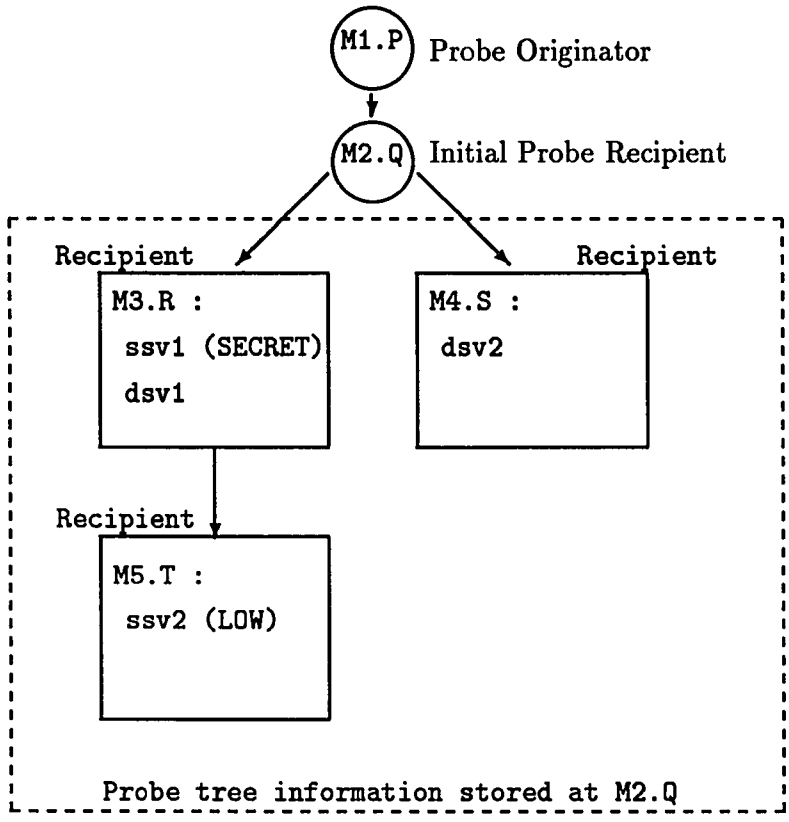

Figure 7.2: Example probe tree information

(a) Static variables that receive flow in the procedure: ssv2(LOW).

(b) Dynamic state variables that need to be s-incremented: none.

Figure 7.2 summarizes the probe tree information that is kept at the initial probe recipient.

Now, assume that M2.Q receives a subsequent probe that is carrying an implicit flow, IMP. Then, M2.Q retrieves its probe tree information and performs the following steps.

1. Locally, check for flow violations by checking that

$$\text{IMP} \leq \underline{\text{ssv1}}(\text{SECRET}) \text{ and } \text{IMP} \leq \underline{\text{ssv2}}(\text{LOW}).$$

2. If no flow violations are detected, send s-increment messages for dsv1 in M3.R and for dsv2 in M4.S.

3. If a flow violation is detected, reject the probe and inform the probe originator. No messages to the recipients are needed.

Notice the savings in sending messages. In case of no violations, s-increment messages are not sent to all recipients. In case of a violation, no message are sent to the recipients. The only extra cost that was incurred is of building and storing the probe tree information.

As long as the probe tree information is not changed, M2.Q can keep using it and avoiding the expensive probe propagation process. Once M2.Q learns of a change in the template of any of the probe recipients (M3.R, M4.S, and M5.T), it needs to recompute the probe tree.

8. CONCLUSION

8.1 Summary

In this dissertation, we started by assuming the existence of a general distributed computing environment. In such an environment, we assumed that all objects are modeled by data type instances. An instance, called a resource module (RM), encapsulates some data items (the state variables of the RM) and their operations (procedures to access the state). RMs in the system may reside on different networked sites and machines. The means of communication between RMs is through message passing. Remote procedure calls (RPCs) are used to request operations on the state of an RM. Processes within an RM service RPC requests. Multiple processes in an RM are allowed to run concurrently.

We, next, suggested an information flow control mechanism to provide multilevel security for the RM system. We called the multilevel secure system the protected resource module (PRM) system. In such a system, users and data items are assigned military-type security classes. The classes are partially ordered and form a lattice. The security requirement of the system is that no information from a class can flow into a lower class.

The information flow control mechanism of the PRM system is a combined compile-time run-time mechanism. (A pure run-time mechanism is rejected on the

basis of its inefficiency.) At compile time, the security of a procedure in a PRM instance is checked. However, since the system is distributed, some run-time checking is also required. So, the compile-time mechanism also builds the necessary structures (the information flow templates) to allow for the needed run-time checks. In addition, if dynamically-bound state variables (variables with security classes that may change at run-time) are allowed, the templates must also contain enough information to allow for computing the classes of those variables.

The run-time flow control mechanism is designed to perform the needed run-time checks and to compute the classes of dynamically-bound variables. The performing of the steps of the run-time mechanism is restricted to only message passing time, again, for efficiency reasons.

By allowing dynamically-bound state variables, we gained flexibility in the sense that a single RM instance and its procedures are able to handle information of various classes. However, we introduce the problem of preserving the consistency of the security information in an RM in the presence of concurrent processes accessing such information. Another problem is that since classes are updated at message passing time only, the computed classes used in those updates may not be precise. In other words, they may not reflect the precise security level of the information contained in the dynamically-bound variables.

We solved the first problem by introducing a concurrency control mechanism for the PRM system. The mechanism is based on a nested transaction model that employs locking techniques to synchronize processes. We actually presented two such concurrency control mechanisms. The first employs five lock types that correspond to all the operations that can be performed on a dynamically-bound variable. Two

locks types are value-locks that handle synchronizing access to the value part of a state variables. The other three lock types are class-locks that handle access to the variable class part of a dynamically-bound variable. By separating locks this way, we maximized concurrency by giving processes independent accesses to values and classes. For example, one process is allowed to read the value of a variable while another is incrementing the class of the variable.

The second locking mechanism is similar but cuts down on overhead by combining some of the five locks. It uses three lock types instead of five, however, we lose some degree of concurrency.

The second problem introduced by the presence of dynamically-bound variables deals with the precision of the computed security classes. One scheme is to simply compute the class of a variable as an upper-bound on the class of the information that possibly flows into the variable. In most of the cases in which information flows only conditionally, such a scheme overclassifies variables.

We solved the overclassification problem by designing a scheme that starts by building a flow graph for each procedure. With the help of the flow graph, the scheme generates, at compile time, some security definitions. A security definition can be used, at run time, to precisely compute the class of the information that flows into a variable. Correctly evaluating security definitions depends on the path taken during the execution of a procedure. As a consequence, some run time support must be available to permit such correct evaluations.

The last contribution of our work deals with the messages (probes) that are sent and propagated, at run time, to RMs to check for inter-module implicit flows that are caused by skipping the execution of procedure calls. Our concern is that probes

may become too numerous in a large system and adversely affect the performance of such a system.

To increase the efficiency of probes, we first, through the design of our locking schemes, allowed them to run concurrently within the same RM. Second, we observed that probes that are propagated to RMs trace a tree structure (probe tree). We devised a scheme in which, for each probe, a probe tree is computed and stored after the first time the probe is sent. Subsequent sending of the same probe can use the probe tree information to check implicit flows without actually having to propagate the probe. In that way, we substantially decreased the probe traffic.

In summary, our system provides multilevel security for a distributed programming environment. The system allows concurrent access to shared data and precisely computes classes of dynamically-bound variables. It also provides for ways to improve the efficiency of inter-module implicit flow checking.

8.2 Areas for Future Research

In this section, we present the areas that were left for future research.

8.2.1 Improvement of the precision of the security mechanisms

In chapter 6, we showed how the precision in checking flows of the security mechanism is improved by precisely computing the classes of dynamic variables using the flow graph s-definitions. Flow graph s-definitions are used to compute the precise flow into dynamically bound variables.

On the other hand, the s-checks in the templates still compute an upper bound on the flow into statically bound variables. This fact implies an overclassification

of flows into statically bound variables and introduces imprecision into the security mechanism.

One area for future work is to investigate the possibility of decreasing the over-classification in s-checks also. To achieve that goal, the s-checks need to be built so as to allow the taking into consideration of the actual execution path. A scheme similar to the one used for s-definitions may be possible. However, the scheme need to carefully handle implicit flows and not allow security to be jeopardized by the reporting of flow violation errors.

8.2.2 The locking mechanism

We presented two locking mechanisms: one using five lock types and the other using three lock types. An investigation needs to be conducted on which of the two schemes is more appropriate in different situations. That also depends on the distributed computational model that is used.

8.2.3 Different concurrency control methods

Locking is the most common way to implement transactions and atomic actions. There are two other ways whose usefulness for our security model can be investigated: *timestamping* [16, 17, 32] and *optimistic* [20] methods.

Each of the methods has some advantages. For example, in a concurrency control mechanism using timestamps, deadlocks cannot occur since no transactions are blocked. In an optimistic concurrency control mechanism, more concurrency is allowed since all transactions are under the initial assumption that they do not interfere with each other.

BIBLIOGRAPHY

[1] Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Andrews, G. R. and R. P. Reitman. "An axiomatic approach to information flow in parallel programs." *ACM Transactions on Programming Languages and Systems* 2, 1 (Jan. 1980): 56-76.

[3] Bell, D. E., and L. J. Lapadula. *Secure Computer Systems: Mathematical Foundations and Model.* MITRE Corp., Bedford, Massachusetts, 1974.

[4] Bernstein, P. A., and N. Goodman. "Concurrency control in distributed database systems." *Computing Surveys* 13, 3 (June 1981): 121-157.

[5] Bernstein, P. A., V. Hadzicolos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[6] Birell, A. D., and B. Nelson. "Implementing remote procedure calls." *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984): 39-59.

[7] Denning, D. E. *Secure Information Flow in Computer Systems.* Ph. D. dissertation. Purdue University, West Lafayette, Indiana, 1975.

[8] Denning, D. E. "A lattice model of secure information flow." *Communications of the ACM* 19, 5 (May 1976): 236-243.

[9] Denning, D. E. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1982.

[10] Denning, D. E., and P. J. Denning. "Certification of programs for secure information flow." *Communications of the ACM* 20, 7 (July 1977): 504-513.

[11] Denning, D. E., and P. J. Denning. "Data security." *Computing Surveys* 11, 3 (Sept. 1979): 227-249.

[12] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger. "The notions of consistency and predicate locks in a database system." *Communications of the ACM* 19, 11 (Nov. 1976): 624-633.

[13] Feiertag, R. J., K. N. Levitt, and L. Robinson. "Proving multilevel security of a system design." *Proc. 6th Symposium on Operating Systems, Operating Systems Review* 11, 5 (Nov. 1977): 57-65.

[14] Fenton, J. S. *Information Protection Systems.* Ph. D. dissertation. University of Cambridge, Cambridge, England, 1973.

[15] Fenton, J. S. "Memoryless subsystems." *Computer Journal* 17, 2 (May 1974): 143-147.

[16] Jefferson, D. R. "The time warp mechanism for database concurrency control." *U.S.C. Technical Report.* Dept. of Computer Science, Univ. of Southern California, Los Angeles, June 1983.

[17] Jefferson, D. R. "Virtual time." *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985): 404-425.

[18] Jones, A. K., and R. J. Lipton. "The enforcement of security policies for computation." *Proceedings 5th Symposium on Operating Systems, Operating Systems Review* 9, 5 (Nov. 1975): 197-206.

[19] Kohler, W. H. "A survey of techniques for synchronization and recovery in decentralized computer systems." *Computing Surveys* 13, 2 (June 1981): 149-183.

[20] Kung, H. T., and J. T. Robinson. "On optimistic methods for concurrency control" *ACM Transactions on Database Systems* 6, 2 (June 1981): 213-226.

[21] Lampson, B. W. "A note on the confinement problem." *Communications of the ACM* 16, 10 (Oct. 1973): 613-615.

[22] Landwehr, C. E. "Formal models of computer security." *Computing Surveys* 13, 3 (Sept. 1981): 247-278.

[23] Lipner, S. B. "A comment on the confinement problem." *Proceedings 5th Symposium on Operating Systems, Operating Systems Review* 9, 5 (Nov. 1975): 192-196.

[24] Liskov, B. "Distributed programming in Argus." *Communications of the ACM* 31, 3 (March 1988): 300-312.

[25] Liskov, B., and R. Scheifler. "Guardians and actions: linguistic support for robust, distributed programs." *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983): 381-404.

[26] Liskov, B., D. Curtis, P. Johnson, and R. Scheifler. "Implementation of Argus." *ACM SIGOPS Operating Systems Review* 21, 5 (Nov. 1987): 111-122.

[27] Mizuno, M. *Highly-Structured Software for Network Systems and its Protection.* Ph. D. dissertation. Iowa State University, Ames, Iowa, 1987.

[28] Mizuno, M. and A. E. Oldehoeft. "Information in a distributed object-oriented system with statically bound object variables." *Proceedings of the 10th National Computer Security Conference,* 1987: 56-67.

[29] Mizuno, M., and A. E. Oldehoeft. *Information Flow Control in a Distributed Object-Oriented System, Part I.* Technical Report TR-CS-88-9. Kansas State University, Manhattan, Kansas, 1988.

[30] Moss, J. E. B. "Nested transactions and reliable distributed computing." *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems,* 1982: 33-39.

[31] Moss, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[32] Reed, D. R. "Implementing atomic actions on decentralized data." *ACM Transactions on Computer Systems* 1, 1 (Feb. 1983): 3-23.

[33] Tanenbaum, A. S., and R. Van Renesse. "Distributed operating systems." *Computing Surveys* 17, 4 (Dec. 1985): 419-470.

[34] Traiger, I. L., J. Gray, C. A. Galtieri, and B. G. Lindsay. "Transactions and consistency in distributed database systems." *ACM Transactions on Database Systems* 7, 3 (Sept. 1982): 323-342.