APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

Ramesh V. Peri

This dissertation has been read and approved by the Examining Committee:

William A. Wulf (Dissertation Advisor)

Paul F. Reynolds (Committee Chair)

Donald A. Brown (Committee Member)

John C. Knight (Committee Member)

John McLean (Committee Member)

Gabriel Robins (Committee Member)

Accepted for the School of Engineering and Applied Science

Dean Richard W. Miksad
School of Engineering and Applied Science

January 1996

Specification and Verification of Security Policies

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Ramesh V. Peri

January 1996

Abstract

Computer security is the protection of the resources of a computer against events that can result in loss of availability, unauthorized access, or modification of data. The techniques devised to protect computers from these attacks are general purpose in nature and therefore cannot directly enforce security that has no universal definition. The high degree of assurance in security properties of systems used in security-critical areas such as military and commerce is usually achieved by verification. In this thesis we develop a framework for carrying out this verification in a formal and systematic manner.

Security verification in information systems consists of — specification of a security policy, specification of the system that must enforce the policy, and verification of policy against system specification. We develop a comprehensive framework for performing all these tasks.

We propose a temporal logic based language that can be used to specify the security policies in a succinct and unambiguous manner. We show that the protection features of information systems can be specified using the specification language Z and develop a method to concisely describe the set of traces generated by such a specification. We develop a set of rules, which can be applied inductively, to verify that the set of traces generated by a specification have the properties specified by the policy. These rules can also be used for verifying that the functionality of a system is not restricted more than required to enforce a given security policy.

Acknowledgments

I thank my advisor, Bill Wulf, for his support and guidance in this endeavour. His patience, compassion and belief in me were instrumental in the completion of this journey that I embarked half a decade back. Bill, thanks for everything.

I thank all my teachers Prof. Govindarajulu, Prof. Nori, Prof. Sangal, Prof. Knight, Prof. Reynolds who taught me many things about Computer Science. I am very grateful to Dr. McLean for his comments and suggestions on many of the ideas in this dissertation.

I would like to thank Bill's students Darrell Kienzle, Sally McKee, Brett Tjaden, Chenxi Wang, Alec Yasinsac for their input at various stages of my research. Darrell deserves special thanks for his very insightful comments about the dissertation.

I would like to thank Ravi and Sukirti who made me a part of their family and provided me moral and emotional support. Sanjay and Indira were always there for me when I needed them. I thank many other friends who made my life at UVa very enjoyable and fulfilling.

I would like to thank my wife, Satya, who played a pivotal role in helping me complete this dissertation through her constant support and encouragement. I would like to thank my parents and my brother whose love and affection are the reason for my success in life until now. I dedicate this dissertation to them.

To my family

Contents

List of Figures

Private information is practically the source of every large modern fortune

- Oscar Wilde

Chapter 1

Introduction

In this dissertation we develop and demonstrate a framework for verifying the security properties of information systems. This framework models

- an information system as a generator of a set of traces where a trace is a sequence of states,
- a security policy as a temporal property that the set of traces satisfy, and
- verification as the process of proving that temporal properties are satisfied by the traces generated by the system.

1.1 Introduction

Computer security includes the protection of the resources of a computer against malicious attacks that can result in loss of availability, unauthorized access, or modification of data. A number of techniques have been devised to protect computers from these attacks that include operating system protection mechanisms and cryptographic techniques. These techniques, which are general purpose in nature, cannot directly enforce security that has

no universal definition. Moreover, the use of computers in security-critical areas such as military and commerce requires a high degree of assurance in the mechanisms designed for their protection that is usually achieved by verification of security properties. Hence, the field of security verification involves

- the study of security models used to describe the protection mechanisms,
- the mechanisms to precisely specify the meaning of security in the form of security policy specifications, and
- security verification as a way of increasing the confidence in the security of the system under consideration.

Although considerable work has been done in all three areas individually, there is no comprehensive framework that allows one to carry out the task of building and verifying security-critical systems. In this thesis we will attempt to develop such a comprehensive framework, by using the existing body of work where possible and developing new concepts when necessary.

1.1.1 Security Models

Security models are used to precisely and formally describe the security relevant features of information systems that allow one to reason about their behavior. These models can be broadly categorized into access control models and information flow models.

1.1.1.1 Access Control Models

Access control models, as the name suggests, control access to information. These models are used primarily for describing the protection features of operating systems. There are four important components in an access control model:

- a set of subjects,
- a set of objects,

- an access matrix that maintains the protection state of the system, and
- a set of rules for changing the protection state of the system.

A number of access control models have been proposed that differ from each other in the state changing rules. These models include HRU model [HAR76], SPM model [SAN88], take-grant model [SNY81], ESPM model [AMM92], grammatical protection models [SNY81], CPD model [BAA90], Typed Access Matrix model [SAN92], and Transform model [SAN89]. An important goal of modeling a system using an access control model is to answer the safety question:

'is a particular state reachable from the initial state'

In most of the models listed earlier it was shown that this question of safety is undecidable.

Here we describe the take-grant model that is used in most of the examples in subsequent chapters. A state in take-grant model [SNY81] is represented as a graph, referred to as the tg-graph. The nodes in the tg-graph, referred to as entities, represent either subjects (active entities like users) or objects (passive entities like files) and the labeled arcs are the rights that the entities have over other entities. The tg-model that we consider has four rights, called *take(t)* and *grant(g)* (which control the propagation of rights) and *read(r)* and *write(w)*.

In a tg-graph, an unfilled circle denotes a subject, a lightly shaded circle denotes an object and a darkly shaded circle denotes either a subject or an object. Also, an entity s having a right x to an entity o is represented by a directed arc labeled x from the node representing s to the node representing o. Changes in the system can occur as a result of rules referred to as *take-rule*, *grant-rule* and *create-rule* which are modeled as graph rewriting rules. These rules are described below (note that in the figures shown, the solid

part of the graph represents the state of the system before the application of the rule, and the dashed part represents the state of the system after the application of the rule).

take-rule:

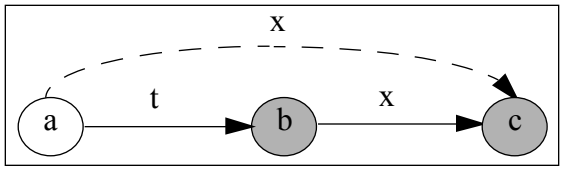


Figure 1.1 : The take-rule of take-grant model

Here *x* is any of the *t*, *g*, *r* or *w* rights.

This rule states that if there is a *t* right from a subject *a* to entity *b*, then *a* can get any right that *b* has.

grant-rule:

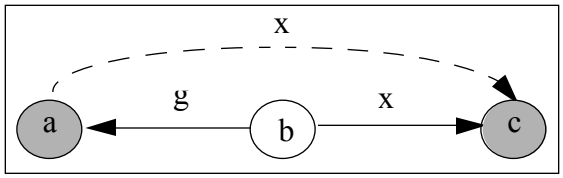


Figure 1.2 : The grant-rule of take-grant model

Here *x* is any of the *t*, *g*, *r* or *w* rights.

This rule states that if there is a *g* right from a subject *b* to entity *a*, then *b* can give any right that it has to *a*.

create-rule:

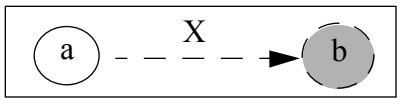


Figure 1.3 : The create-rule of take-grant model

Here X is a subset of the set {*t*, *g*, *r*, *w*}.

This rule states that any subject *a* can create a new entity *b* with any set of rights.

In the tg-model, the path between two nodes is represented in a notation similar to that of regular expressions. For example, a path from entity a to entity b such that there are some arbitrary number of entities (x_1, x_2, ..., x_n) where an arc labeled *'t'* is present

- from a to x_1,
- from each x_i to x_{i+1}, and
- from x_n to b.

can be represented by $\overrightarrow{t*}$ where the 't' denotes the right, the '*' denotes that 't' can be present any number of times and the arrow distinguishes the source and the destination of the right. Any path can be easily described by this notation.

These access control models are useful for modeling the protection features of operating systems but are not useful for reasoning about flow of information.

1.1.1.2 Information Flow Models

Information flow models deal with flow of information rather than access to information. This notion of information flow was first proposed by Denning [DEN76] whose definition can be phrased as '*information is said to flow from object a to object b if the value of object b some how depends on the value of object a'*.

Denning also developed a procedure, called *flow analysis*, for enumerating all the information flows between the variables in the program that can be used to check whether any illegal flows can occur.

The definition of information flow in Denning's model was later formalized using state machines, resulting in a number of *information flow predicates*. These predicates are defined in terms of the *traces* of a system, where a *trace* is a possible sequence of inputs and outputs that characterize the operations of the system. One of the earliest and most

popular information flow predicates is *Noninterference* [GM84] which states that information flows from entity *a* to entity *b* if the inputs of entity *a* can affect either the inputs or the outputs of entity *b*.

A number of other information flow predicates such as Restrictiveness [MCC87], Nondeducibility [SUT86], Separability [MCL94], Nondeducibility on Strategies [WJ90] have been defined that capture different notions of information flow. An important aspect in which these predicates differ from each other is their composition i.e., the properties of the system obtained by composing two systems that satisfy one of the above predicates.

1.1.1.3 The Composition Principle

A composition construct is a way of connecting the outputs of one system to the inputs of another. Considerable work has been done to investigate the properties of information flow predicates under different composition constructs.

An important work in this area is that of Alpern and Schneider [AS85] who define both the property and a system as a set of behavior sequences (referred to as traces). In this work, a property is said to hold for a system only if the set of traces of the system is a subset of the set of traces that characterize the property. Based on this notion of a system and its property, Abadi and Lamport [AL90] gave a composition principle that can be used to determine whether a composite system satisfies some property from the properties of its components. An important restriction on this composition principle is that it is applicable only to properties of individual traces. Unfortunately, the Abadi-Lamport composition is not applicable to information flow predicates since they are not properties of traces but properties of *sets* of traces [MCL94]. Therefore a composition principle, or at least a framework, that can be used to reason about composite systems built out of components that satisfy some information flow property is required.

1.1.2 Security Policies

A security policy is a set of properties that are stated in terms of some model of the system. These properties can be either access control properties that are stated in terms of access control model or information flow properties that are stated in terms of information flow model of the system. A policy that consists of just access control properties is called an access control policy and a policy that consists of just information flow properties is called an information flow policy. Real world security policies usually contain both access control and information flow properties.

A number of security policies have been proposed and studied extensively. A widely known policy is Multi-Level Security (MLS) policy in which requirements are stated in terms of information flow relations. In this policy, every entity in the system is associated with a level from a set of partially ordered levels, and information is allowed to flow from one entity to another at the same or higher level. Another policy that is common in commercial applications is called the Chinese-Wall policy [BN89]. In this policy, entities are divided into conflict of interest class sets, and information is allowed to flow from an entity to at most one entity belonging to a conflict of interest class set.

In order to specify the security policies precisely and unambiguously some formal specification frameworks have been developed [FOL89a][CUP93]. These frameworks, although useful, are not general enough to express the policies that one might encounter in practice. Moreover these specification languages do not have an associated verification framework to prove that a system correctly enforces these policies.

1.1.3 Security Verification

Security verification is the process of proving that the properties specified in a security policy are enforced in the information system. Security policies are global properties that specify the legal behaviors of a system whereas the system specification is usually in the

form of a state machine where the transitions are local to each state. This makes the verification process difficult since it involves characterizing all global behaviors generated by the state machine and checking whether they possess the properties specified by the policy.

Security verification is influenced by the work in program verification. A number of systems that resemble programming languages have been proposed to specify the design of information systems. Some such systems are Gypsy [CHE81], InaJo [CHE81], and SPECIAL [CHE81]. These languages are part of larger environments that contain

- verification condition generators that perform information flow analysis on the specification and generate conditions for enforcement of MLS Policy, and
- a theorem prover used to prove the conditions generated by the verification condition generator.

These systems support only the MLS policy which makes their use in areas other than military systems, difficult.

1.2 The Security Verification Problem

The information systems used in critical areas require a high degree of assurance in their security and safety properties. This high degree of assurance about the security of such systems can be achieved by independent verification of security properties apart from good design practices and testing processes.

In this dissertation we investigate the problem of verifying the security properties of information systems. These properties can be either information flow properties or access control properties, and the information system can be an operating system or a database system that controls and manipulates either the accesses to data or the data itself. In order to demonstrate the utility of the verification techniques that are developed in this

thesis we use a system whose protection features can be modeled by the take-grant model. However, the techniques developed here are equally applicable to any system that can be modeled as a state transition system.

1.3 Motivation

Consider the task of developing the specification of a system similar to the take-grant system specified earlier that has to enforce Multi-Level security policy. In such a system, each entity is assigned a security level and the information flow between entities of different levels is restricted by the partial order relation ≤ defined on the levels. Before we can proceed any further, the definition of information flow has to be given in the context of the take-grant model. For the purpose of this example, assume that information flows from A to B if there is a read or a take right from B to A or a write or a grant right from A to B.

Consider a system similar to the take-grant model in which the initial state is restricted so that there are no rights between any of the entities present in the system. Although this does enforce the MLS policy, the requirement is overly restrictive. It reduces the functionality of the system; an entity at a higher level will not be able to get any information from an entity at a lower level, even though this behavior is allowed by the MLS policy. Therefore a weaker restriction on the behavior of the subjects is needed in order to increase the functionality of the system.

Consider another system in which the initial state satisfies the following restrictions (note that these restrictions might seem to enforce MLS policy but in reality they *do not* for reasons given below).

- A take or a read right can exist from an entity at a higher level to an entity at a lower level.
- A write right can exist from an entity at a lower level to an entity at a higher level.

- A grant right cannot exist between any two entities.

A system that satisfies all the above restrictions has more functionality than the one specified earlier because it is possible for information to flow from an entity at a lower level to an entity at a higher level. But this system does not enforce MLS policy because the tg-graph shown in the following figure

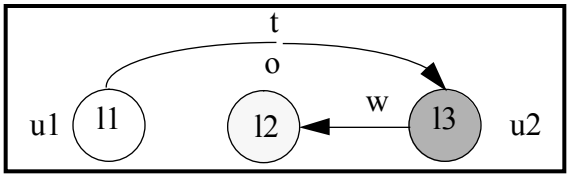


Figure 1.4 : The tg-state that might violate MLS requirement

where *l1*, *l2* and *l3* are the levels of *u1*, *u2* and *o* respectively and *l1 > l2 > l3* can be transformed into the graph shown in the following figure by the application of take-rule.

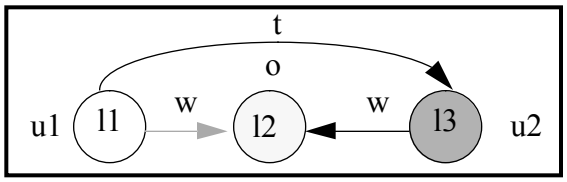


Figure 1.5 : The tg-state that violates MLS requirement

The state in the above figure violates the requirement of MLS since it is possible to for the higher level subject *u1* to write to the lower level object *o.* The fact that the above specification does not enforce MLS was only clear after some analysis. This gives credence to our argument that a formal proof of the security properties of the specification is essential to increase our confidence in it.

In the above example, any of the following additional restrictions on the specification of the take-rule will make the system enforce the MLS policy.

- The take rule is modified so that it can be used to copy only read rights.
- The take rule is modified so that it can be used to copy read rights and only write rights to objects at higher level.

Of course, a formal proof that the above specification enforces the MLS policy is required to increase our confidence in the system more than just an informal argument. In order to develop this proof we need a comprehensive framework to formally specify a security policy and formally describe the system that enforces the policy. Another thing to notice is that while both modifications to the take rule given above make the system enforce the MLS policy, the latter is less restrictive than the former one. We need a way to formally characterize this feature of the specifications of systems and prove that a given specification is not unnecessarily restrictive.

1.4 Organization of the Thesis

The main goal of this dissertation is to develop a comprehensive framework that can be used for security verification. For this purpose we investigate the following four topics:

- definition of the information flow relation,
- specification of security policies,
- specification of the system, and
- verification of security properties.

The first task has been extensively dealt with in the literature resulting in a number of information flow predicates. An important aspect that has been extensively studied is the composition of each predicate with itself. In [MCL94], McLean developed a framework based on set theory for reasoning about the composition of an information flow predicate with any other predicate. In Chapter 2 we present an alternative way of reasoning about the composition properties of these predicates that is simple, elegant, and amenable for automatic verification as compared to McLean's approach.

Informal specification of security policies can result in ambiguities and misinterpretations. Therefore, formal specification methods based on predicate calculus

[FOL88a] and modal logic [CUP93] have been developed for precisely specifying these policies. In Chapter 3 we develop an alternate framework based on partial order temporal logic for specifying the security policies that is more expressive than the ones found in the literature. In this chapter we also give the formal semantics of this language in terms of a state machine model.

Traditionally, the protection mechanisms of information systems were specified using access control models. In Chapter 4 we explore why access control models are not adequate for describing the security-relevant features of modern information systems, and propose that a specification language such as Z is more suitable for this task. We demonstrate this by developing a Z specification of the security features of the Unix file system. We also develop a method to characterize a set of traces that can be generated by a system specified in Z, which results in a description suitable for verification.

In Chapter 5, we develop a framework for verifying that the set of traces generated by a system specified in Z satisfy the security properties specified as temporal formulas. We develop a set of rules that can be applied inductively, based on the structure of the temporal formula to be proved, to carry out the verification process. We demonstrate the utility of this framework by proving that a system similar to the take-grant system specified in Z satisfies the MLS policy. In this chapter we also identify different kinds of information flow security policies and give a method to enforce them based on explicit labeling of entities.

The specification of a system that enforces a security policy is usually a restricted form of a more general specification referred to as the functional specification. One of the dangers in developing a system specification from its functional specification and the security policy is over-restriction, meaning that the functionality of the system might be restricted unnecessarily to enforce the policy. In Chapter 6 we develop a framework, which given a security policy and a specification of the system that enforces this policy, makes it possible to prove that the system is not over restricted.

Chapter 7 gives the conclusions that we draw from our experience in trying to develop a comprehensive framework for security verification and suggests some directions in which this work can be extended.

The last thing one discovers in composing a work is what to put first

- Blaise Pascal

Chapter 2

Composition Properties of Information Flow Predicates

A number of predicates have been proposed that capture the notion of information flow in the specification of security policies. In this chapter a comprehensive framework based on many-sorted predicate logic is developed for specifying and reasoning about the composition properties of these information flow predicates under a variety of composition constructs. We also show that this reasoning process can be automated in the PVS theorem proving system.

2.1 Introduction

One of the goals of protection mechanisms in a system is to control the flow of information between the entities in it. In order to achieve this, the notion of *information flow* has to be made precise. A number of attempts have been made to formulate the notion of information flow based on Shannon's information theory that resulted in a number of definitions usually referred to as the information flow predicates.

For example, Noninterference is an information flow predicate proposed by Goguen and Meseguer [GM82] that has been used quite extensively in security models. It defines a system to be secure if high-level users cannot influence the behavior of low-level users when the policy mandates that information should not flow from high-level users to low-level users. The simple and elegant specification made Noninterference a basis for a general theory of security.

Later, a number of other information flow predicates were proposed such as Nondeducibility [SUT86], Restrictiveness [MCC87], Noninference [MCL94]. These predicates tried to take into account non-deterministic systems, more prevalent in the real world. An important difference between these predicates is their composition property i.e., whether a system obtained by composing two systems satisfying the security property also satisfies that property. This question of composition is of great importance since the properties of complex systems need to be derived from the composition of the properties of its constituents. Unfortunately many of the security properties mentioned earlier only compose under special conditions, and one of the fertile areas of research is finding these conditions.

Although considerable work has been done on the composition properties of each of these information flow predicates with themselves, few results exist about their composition with each other. An important work in this area is that of McLean [MCL94]. In this chapter we develop an alternative method to the one presented in [MCL94] for specifying and reasoning about the composition properties of the information flow predicates that is amenable to automatic theorem proving.

2.2 Composition Properties of Information Flow Predicates

In this section we present the trace model [MCL84] and describe McLean's approach to specifying and reasoning about the composition properties of information flow predicates.

2.2.1 Trace Model

A system S can be specified as a state transition system that consists of

- a set of states, Σ, where the states are described by the values of a set of state variables.
- and a transition relation $T \subseteq (\Sigma \times \Sigma)$.

Definition: Given an information system S = (Σ, T), its trace set denoted by *trace(S)* is the set

$$\{ <s_1, s_2, \ldots> \mid (s_i \in \Sigma) \}$$

Definition: A trace $<s_1, s_2, ...>$ is said to be a *valid trace* in a system S = (Σ, T), if for all i, $(s_i, s_{i+1}) \in T$.

It should be noted that a system can be completely described by specifying its valid traces.

Definition: The state space of a system is defined as a set of input variables $<in_1, in_2, ... in_m>$ where each input variable in_i ranges over some set I_i and a set of output variables $<out_1, out_2, ..., out_n>$ where each output variable ranges over some set O_i. The state space Σ of such a system is the set

$$\{<<in_1, in_2, ..., in_m>, <out_1, out_2, ..., out_n>> \mid (in_i \in I_i) \wedge (out_i \in O_i)\}.$$

Without loss of generality, we will consider a simpler system where there are only two inputs lin and hin (which denote low-input and high-input) and two outputs lout and hout (which stands for low-output and high-output) [MCL94]. This assumption means that the state space is a set of a tuples of the form (*lout*, *lin*, *hout*, *hin*) where *lout*, *lin*, *hout* and *hin* are the values of the state variables hin, lin, hout and lout that range over {0, 1}. A trace in such a system is denoted by the sequence

$$<(hin_1, lin_1, hout_1, lout_1), (hin_2, lin_2, hout_2, lout_2), ...>$$

These traces can be described by a set of functions F = {HIN, LIN, HOUT, LOUT}, whose values together uniquely define a trace. These functions are defined as:

- HIN is a function that gives the sequence of all hin's in every state of the given trace,
- LIN is a function that gives the sequence of all lin's in every state of the given trace,
- HOUT is a function that gives the sequence of all hout's in every state of the given trace, and
- LOUT is a function that gives the sequence of all lout's in every state of the given trace.

For instance, given a trace t = <(1,1,0,0), (1,0,1,1), (0,0,1,1)>, HIN(t) is <1,1,0> and HOUT(t) is <0,1,1>. That is, HIN(t) is the sequence consisting of all hin's of the trace t and HOUT(t) is the sequence consisting of all hout's of the trace t.

Definition: A system property is a property of the set of all valid traces of the system.

The system properties can be classified into two kinds:

- Functional Property — a property that a single valid trace, in isolation, is required to satisfy, and
- Possibilistic Property — a property that the set of all valid traces is required to satisfy.

For example, a property that every trace in the set of valid traces of a system described earlier, satisfy the requirement that ($hout_i = hin_i + lin_i$) is a functional property since this is the property of an individual trace. The safety (i.e., something bad does not

happen) and liveness (i.e., something good eventually happens) properties fall under the category of functional properties. The average response time over all possible executions of a system is not the property of a single trace and is therefore a possibilistic property. The information flow properties fall in this category.

The important difference between the functional and the possibilistic properties is that the former are closed under refinement i.e., if system S satisfies a functional property then any system whose set of traces is a subset of the valid traces of S also satisfies that property while the latter are not.

Unfortunately, the Abadi and Lamport composition principle mentioned in Section 1.1.1.3 cannot be applied to information flow properties, since they do not fall in the class of functional properties [PW94] [MCL94]. Therefore a different composition principle needs to be developed that can be applied to possibilistic properties in general and information flow properties in particular.

2.2.2 McLean's Framework

McLean [MCL94] developed a framework to reason about the composition properties of information flow predicates based on an informal notation. In this approach, information flow predicates, which fall under the category of possibilistic security properties, are expressed as closure properties of functions called *selective interleaving functions.* This was motivated by the observation that security properties are closure properties of functions that take two traces and produce a third one.

Let S be a system whose state space is $\{<<in_1,...,in_m>, <out_1,...,out_n>>\}$. A selective interleaving function of type $F_{i,j}$ (where $i \in \{0, 1, 2\}^n$ and $j \in \{0, 1, 2\}^m$) takes two traces as its arguments and produces a third trace that agrees with the first argument trace with respect to input (output) such that corresponding element of i (j) is 1 and with the second argument trace with respect to input (output) such that the corresponding

element of i (j) is 2. Different selective interleaving functions differ on what they assign to input (output) when i (j) is 0.

For example, the output of a function f of type $F_{<1,\ 2>,\ <1,\ 2>}$ with the inputs <<1, 2>, <3, 4>> and <<a, b>, <c, d>> is <<1, b>, <3, d>>.

The information flow predicates are then defined such that a set of traces is said to satisfy an information flow predicate if it is closed under some selective interleaving function. These selective interleaving functions are used to specify a variety of information flow properties like Separability ($F_{<1,2>,\ <1,2>}$) and Generalized Noninference ($F_{<1,\ 2>,\ <0,\ 2>}$). The theory of selective interleaving functions was used to provide a uniform framework for specifying the information flow security predicates and a methodology for reasoning about their behavior under different kinds of composition constructs. The composition results are of the form:

the composition S of S1 and S2, which are closed under functions of types $F_{i1,j1}$ and $F_{i2,j2}$, is closed under a function of type $F_{i,j}$ under some conditions that involve i1, i2, i, j1, j2 and j.

In the above statement, the conditions that involve i1, i2, j1, j2, i, j is of the form (i = i1 = i2) and (j = j1 = j2). In [MCL94], McLean enumerates these conditions for the composition properties of the information flow predicates.

Also, a partial order can be defined on the selective interleaving functions as follows:

$$F_{i1,j1} \leq F_{i2,j2} \quad \text{iff} \quad \begin{pmatrix} \forall x \times ((1 \leq x \leq m) \Rightarrow ((i1[x] = i2[x]) \vee (i2[x] = 0))) \\ \wedge \\ \forall x \times ((1 \leq x \leq m) \Rightarrow ((j1[x] = j2[x]) \vee (j2[x] = 0))) \end{pmatrix} \begin{matrix} \div \\ \div \\ j \end{matrix}$$

Intuitively, $(F_{i1,j1} \leq F_{i2,j2})$ means that every system that is closed under $F_{i1,j1}$ is also closed under $F_{i2,j2}$.

This partial order on the selective interleaving functions in turn imposes a partial order among security predicates that are expressed using these functions. This makes it easy to observe the relative strengths of these information flow predicates. From the above definition it is easy to see that Separability ($F_{<1,2>, <1,2>}$) is stronger than Generalized Noninference ($F_{<1, 2>, <0, 2>}$) since

$$F_{(1,2), (1,2)} \leq F_{(1,2), (0,2)}$$

according to the above definition.

We propose a different framework for specifying the possibilistic security properties based on many-sorted predicate logic; this framework can also be used to specify the composition constructs. A drawback of selective interleaving functions is that they cannot be used to express security predicates such as McCullough's Generalized Noninterference [MCC87] in an elegant manner [MCL87]. This is because the interleaved trace in Generalized Noninterference is constructed by different functions for different parts of the input traces. In our method, reasoning about composition of different security properties is straightforward and intuitive since we can use the rich and powerful logical formalism of predicate logic. This new formalism also has the advantage that the process of proving the composition properties of information flow predicates can be automated.

2.3 Logic Based Specification of Information Flow Predicates

A many-sorted logic for specification of information flow predicates is given in Section 2.3.1 and its semantics is given in Section 2.3.2. Section 2.3.3 gives the specification of a number of information flow predicates in this formalism and Section 2.3.4 shows how this specification formalism can be used to specify a variety of composition constructs.

2.3.1 The Specification Framework

An information flow security property for a system S can be specified as a formula over a many-sorted predicate logic consisting of:

- the symbols *true* and *false.*
- a set of *sorts*.
- a set of *variables* x, y, z each of which has an associated sort.
- a set of *constants* X, Y, Z each of which has an associated sort.
- a set of functions F where each function has a signature of the form

  $$[sort_0, sort_1, \ldots, sort_{n-1} \rightarrow sort_n]$$

  where $sort_i$ ($i < n$) denotes the sort of i^{th} argument of the function and $sort_n$ denotes the sort of its return value.

- a set of predicates P where each predicate has an arity that denotes the number of its arguments each of which has an associated sort. These predicates return either *true* or *false*.
- the Boolean connectives $\wedge$, $\vee$, $\Rightarrow$ and $\neg$.
- the quantifiers $\forall$ and $\exists$.

Definition: A *term* is defined as

- a *variable* or a *constant* is a term.
- $f(t_1, \ldots t_n)$ is a term where f is a function from the set F and t_i is a term of appropriate sort.

Definition: An *atomic formula* is defined as $p(t_1, \ldots t_n)$ where

- p is a predicate symbol from the set of predicate symbols P of arity n and

- t_i is a term of appropriate sort.

Definition: A *formula* is defined to be

- an atomic formula is a formula.
- If F1 and F2 are formulas then so are (F1 $\wedge$ F2) , (F1 $\vee$ F2) and ($\neg$ F1).
- If x is a variable and F is a formula then (($\forall$x) F) and (($\exists$x) F) are formulas.

Definition: A *subformula* of a formula F is a consecutive sequence of symbols from F that is a formula.

Definition: An occurrence of a variable x in a formula F is *bound* if there is a subformula G of F containing that occurrence of x that begins with ($\forall$x) or ($\exists$x). An occurrence of x in F is *free* if it is not bound.

Definition: A formula is said to be a *ground formula* if it does not contain any variables.

Definition: Given a formula (($\forall$x) F) or (($\exists$x) F) , $F \mid_{x = X}$ denotes the formula obtained by replacing all occurrences of variable x with constant X in F.

2.3.2 Semantics

Given a formula, its truth value can be inductively defined as follows:

Definition: An information system S is said to satisfy a security property F, written as S |- F (read |- as satisfies) if

- F is a ground atomic formula and F is assigned the value *true*.
- F is of the form (($\forall$x) G) and x is a variable of some sort and for every constant X of the same sort as x, S |- $G \mid_{x = X}$.
- F is of the form (($\exists$x) G) and x is a variable of some sort, if there is a constant X of

the same sort as x, S |- G | $_{x = X}$.

- F is of the form (G ∧ H) and S |- G and S |- H.
- F is of the form (G ∨ H) and S |- G or S |- H.
- F is of the form (¬G), and S |- G is *false*.

2.3.3 Specification of Information Flow Predicates

In order to describe the information flow predicates in the many sorted predicate logic described above we assume that there are two sorts {trace, valseq}. Also the set of functions that describe the traces are F = {HIN, LIN, HOUT, LOUT} as defined in Section 2.2.1 — denoting the inputs and outputs of high-level and low-level respectively with signature [trace → valseq]. Also S, S1 and S2 are members of the set of predicates P that specify whether a trace is present in the corresponding system or not, i.e., each of these predicates has one argument of sort trace.

The information flow predicates, can be expressed as an implication of the form $(A \Rightarrow B)$ where A and B are formulae as defined earlier (ignoring the universal quantifiers).

In fact, a selective interleaving function of the type $F_{(<1,2>, <1,2>)}$ of [MCL94] can be expressed in this algebraic notation as

$$\forall x\ \forall y \left(\begin{pmatrix} S(x) \\ \wedge \\ S(y) \end{pmatrix} \Rightarrow \exists p \begin{pmatrix} S(p) \wedge (HOUT(p) = HOUT(x)) \wedge \\ (HIN(p) = HIN(x)) \wedge (LIN(p) = LIN(y)) \wedge \\ (LOUT(p) = LOUT(y)) \end{pmatrix} \right)$$

The above formula means that, if there are two traces x and y in a system S then there is a trace p in S where trace p has the same high-level behavior as trace x and the same low-level behavior as trace y.

Next we show that the above language can be used to express different kinds of information flow predicates.

Noninference

This predicate specifies that the high-level events should not influence the low-level events in the system. A system S is said to satisfy Noninference if

$$\forall x\Big((S(x)) \Rightarrow \exists y\Big(\begin{matrix} S(y) \wedge (LIN(x) = LIN(y)) \wedge (LOUT(x) = LOUT(y)) \wedge \\ (HIN(y) = \lambda) \wedge (HOUT(y) = \lambda) \end{matrix} \Big)\Big)$$

That is, if there is a trace x in the system then there should be another trace y that has the same low-level events as x even when the high-level events are non-existent (here a value λ implies that the corresponding event is non-existent).

Generalized Noninference

This predicate specifies that the high-level input event should not influence the low-level events in the system. A system S is said to satisfy Generalized Noninference if

$$\forall x\Big((S(x)) \Rightarrow \exists y\Big(\begin{matrix} S(y) \wedge (LIN(x) = LIN(y)) \wedge (LOUT(x) = LOUT(y)) \wedge \\ (HIN(y) = \lambda) \end{matrix} \Big)\Big)$$

The meaning of the above statement is that, if there is a trace x in the system then there should be another trace y that has the same low-level events as x even if the high-level inputs of y are non-existent.

Separability

This predicate specifies that it should not be possible to deduce anything about the high-level events by observing the low-level events. A system S is said to satisfy Separability if

$$\forall x \forall y \Big(\Big(\begin{matrix} S(x) \\ \wedge \\ S(y) \end{matrix} \Big) \Rightarrow \exists p \Big(\begin{matrix} S(p) \wedge (HIN(p) = HIN(x)) \wedge \\ (HOUT(p) = HOUT(x)) \wedge \\ (LIN(p) = LIN(y)) \wedge (LOUT(p) = LOUT(y)) \end{matrix} \Big) \Big)$$

That is, if there are two traces *x* and *y* in the system then there is a trace p whose high-level and low-level events are an interleaving of the high-level and low-level events of the traces *x* and *y.*

Generalized Noninterference

Generalized Noninterference specifies that it should not be possible to deduce anything about the high-level inputs by observing the low-level events. A system S is said to satisfy Generalized Noninterference if

$$\forall x \forall y \left(\begin{pmatrix} S(x) \\ \wedge \\ S(y) \end{pmatrix} \Rightarrow \exists p \left(\begin{array}{c} S(p) \wedge (LOUT(p) = LOUT(x)) \wedge \\ (LIN(p) = LIN(x)) \wedge (HIN(p) = HIN(y)) \end{array} \right) \right)$$

That is, if there are two traces *x* and *y* in a system then there is a trace *p* that is an interleaving of the high-level inputs and low-level events of traces *x* and *y.*

Definition: A property *X* is said to be weaker than the property *Y* if $(X \Rightarrow Y)$ is a valid formula.

From the above definition we can see that Generalized Noninterference is weaker than Separability. In fact, the following partial order specified in [MCL94] can be easily derived from the above definitions.

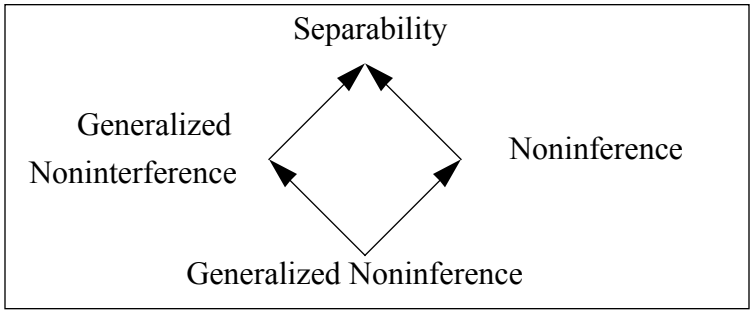


Figure 2.1 : Partial Order of Information Flow Predicates

In this section we showed that information flow predicates can be expressed in a two-sorted predicate logic based language. In the following section we will show that this formalism can be used to express the composition constructs.

2.3.4 Specification of Composition Constructs

Information systems can be composed in a variety of ways. Typically, a composition of two systems is defined to be a new system whose behaviors are defined in terms of the behaviors of its constituent systems. In our framework, a composition construct can be specified by the relationship between the set of traces in the constituent systems S1 and S2 with the set of traces in their composition S. This formulation of composition constructs as relations between traces transforms the problem of proving the security properties of composed systems to theorem proving in predicate logic.

Depending on the systems that are involved, the composition constructs can be divided into internal and external composition constructs.

2.3.4.1 Internal Composition

The internal composition constructs are typically Boolean constructs such as union and intersection that are used to compose similar kinds of systems. Hence the traces of all the systems involved in internal composition are defined by the same set of characterizing functions.

Union

The union of S1 and S2 accepts the input that is acceptable to either S1 or S2 and produces the output that is same as one of the outputs produced by S1 or S2. It is formally specified as

$$\forall x\,(S(x) \Leftrightarrow (S1(x) \vee S2(x)))$$

Intersection

The intersection of S1 and S2 accepts the input that is acceptable to both S1 and S2 and produces the output that can be produced by both. It can be formally specified as

$$\forall x\,(S(x) \Leftrightarrow (S1(x) \wedge S2(x)))$$

Difference

The difference of two systems S1 and S2 accepts the input that is acceptable to both S1 and S2 but produces output that is acceptable to either S1 or S2 but not both. It can be formally specified as

$$\forall x\left(S(x) \Rightarrow \exists y, z\left(\begin{array}{c} S1(y) \wedge (IN(x) = IN(y)) \wedge S2(z) \wedge (IN(x) = IN(z)) \wedge \\ (OUT(x) = OUT(y)) \vee (OUT(x) = OUT(z)) \wedge \\ (OUT(y) \neq OUT(z)) \end{array}\right)\right)$$

and also

$$\forall y, z\left(\left(\begin{array}{c} S1(y) \wedge (IN(y) = IN(z)) \wedge \\ S2(z) \wedge (OUT(y) \neq OUT(z)) \end{array}\right) \Rightarrow \exists x\left(\begin{array}{c} S(x) \wedge (IN(x) = IN(y)) \wedge \\ \left(\begin{array}{c} (OUT(x) = OUT(y)) \vee \\ (OUT(x) = OUT(z)) \end{array}\right) \end{array}\right)\right)$$

2.3.4.2 External Composition Constructs

The external composition constructs are cross product, cascade and feedback and these are used to construct a system from a number of other systems. In the formulation of the these composition constructs we use the simple system definition given in Section 2.2.1 to make the definitions easier to understand. However, it should be noted that these definitions can be easily extended to handle the general definition of a system given in Section 2.2.1.

Cross Product

Here two systems are run in parallel and any trace of the combined system is a combination of the traces from the individual systems. A system S is said to be the cross-product of two systems S1 and S2 if it satisfies the following condition.

$$\forall x \left(S(x) \Rightarrow \begin{pmatrix} \exists y \, (S1(y) \wedge (IN1(x) = IN1(y)) \wedge (OUT1(x) = OUT1(y))) \\ \wedge \\ \exists z \, (S2(z) \wedge (IN2(x) = IN2(z)) \wedge (OUT2(x) = OUT2(z))) \end{pmatrix} \right)$$

The meaning of the above statement is that every trace in S is obtained by concatenating a trace from S1 and another trace from S2.

Also,

$$\forall x \forall y \left(\begin{pmatrix} S1(x) \wedge \\ S2(y) \end{pmatrix} \Rightarrow \exists z \begin{pmatrix} (S(z) \wedge (OUT1(z) = OUT1(x))) \wedge \\ (OUT2(z) = OUT2(y)) \wedge \\ (IN1(z) = IN1(x)) \wedge (IN2(z) = IN2(y)) \end{pmatrix} \right)$$

That is, for every trace *x* in S1 and every trace *y* in S2 there is a trace in S that is obtained by concatenating *x* and *y*.

The cross-product operation can be pictorially depicted as shown below.

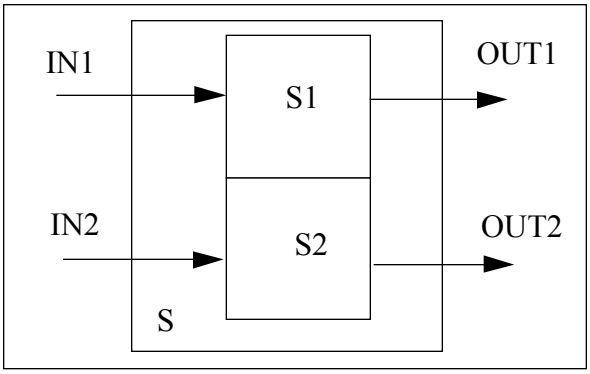


Figure 2.2 : Cross-product Composition of S1 and S2

Cascade

In this kind of composition, the output of one system is fed as input to another system. A system S is said to be the cascade of two systems S1 and S2 if it satisfies the following condition

$$\forall x \left(S(x) \Rightarrow \exists y \exists z \begin{pmatrix} S1(y) \wedge S2(z) \wedge (HIN(x) = HIN(y)) \wedge \\ (LIN(x) = LIN(y)) \wedge (HOUT(x) = HOUT(z)) \wedge \\ (LOUT(x) = LOUT(z)) \wedge (LOUT(y) = LIN(z)) \wedge \\ (HOUT(y) = HIN(z)) \end{pmatrix} \right)$$

The meaning of the above statement is that, every trace *x* in S is built out of a trace *y* in S1 and a trace *z* in S2 where the outputs of *y* are same as the inputs of *z* and inputs of *x* are same as inputs of *y* and outputs of *x* are same as the outputs of *z*.

Also,

$$\forall y \forall z \left(\left(\begin{array}{c} S1(y) \wedge S2(z) \wedge \\ (LOUT(y) = LIN(z)) \wedge \\ (HOUT(y) = HIN(z)) \end{array} \right) \Rightarrow \exists x \left(\begin{array}{c} S(x) \wedge (HIN(x) = HIN(y)) \wedge \\ (LIN(x) = LIN(y)) \wedge \\ (LOUT(x) = LOUT(z)) \wedge \\ (HOUT(x) = HOUT(z)) \end{array} \right) \right)$$

That is, if there are two traces *y* and *z* in S1 and S2 where outputs of *y* are same as the inputs of *z* then there will be a trace *x* is S such that the inputs of *x* are same the inputs of *y* and the outputs of *x* are same as the outputs of *z*.

This composition can be pictorially represented as

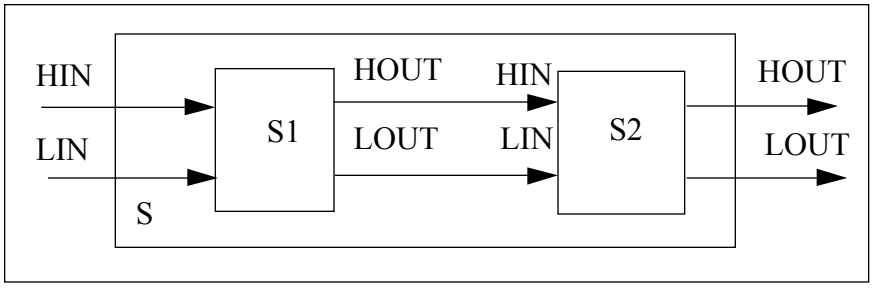


Figure 2.3 : Cascade Composition of S1 and S2

Feedback

Here a system acts as front end to another system i.e., the input and output of a system is presented to the outside world through a front-end. A system S is said to be feedback composition of two systems S1 and S2 if it satisfies the conditions shown below where *f* and *f'* are some arbitrary functions with signature [valseq → valseq].

Typically *f* and *f'* are such that given a sequence *x*, an interleaving, denoted by the function symbol *C*, of *f(x)* and *f'(x)* produces the sequence *x* i.e., (*C(f(x), f'(x)) = x)*.

$$\forall x\left(S(x) \Rightarrow \exists y \exists z\left(\begin{array}{c} S1(y) \wedge S2(z) \wedge (HIN(x) = f(HIN(y))) \wedge \\ (LIN(x) = f(LIN(y))) \wedge (HOUT(x) = f'(HOUT(y))) \wedge \\ (LOUT(x) = f'(LOUT(y))) \\ (LIN(z) = f(LOUT(y))) \wedge (HIN(z) = f(HOUT(y))) \wedge \\ (HOUT(z) = f'(HIN(y))) \wedge (LOUT(z) = f'(LIN(y))) \end{array}\right)\right)$$

The meaning of the above statement is that every trace *x* of S is obtained from a trace *y* in S1 by applying some function on it provided there is a trace *z* in S2 that can be obtained from the trace *y* by applying some other function.

Also,

$$\forall y \forall z\left(\begin{array}{l} \left(\begin{array}{c} S1(y) \wedge S2(z) \wedge (LIN(z) = f(LOUT(y))) \wedge \\ (HIN(z) = f(HOUT(y))) \wedge \\ (HOUT(z) = f'(HIN(y))) \wedge (LOUT(z) = f'(LIN(y))) \end{array}\right) \Rightarrow \\ \exists x\left(\begin{array}{c} S(x) \wedge (HIN(x) = f(HIN(y))) \wedge (LIN(x) = f(LIN(y))) \wedge \\ (LOUT(x) = f'(LOUT(y))) \wedge (HOUT(x) = f'(HOUT(y))) \end{array}\right) \end{array}\right)$$

That is, for every trace *y* in S1 and *z* in S2 that satisfy a certain property there is a trace *x* in S that is obtained from *y* by applying some function to it.

This operation can be pictorially represented as

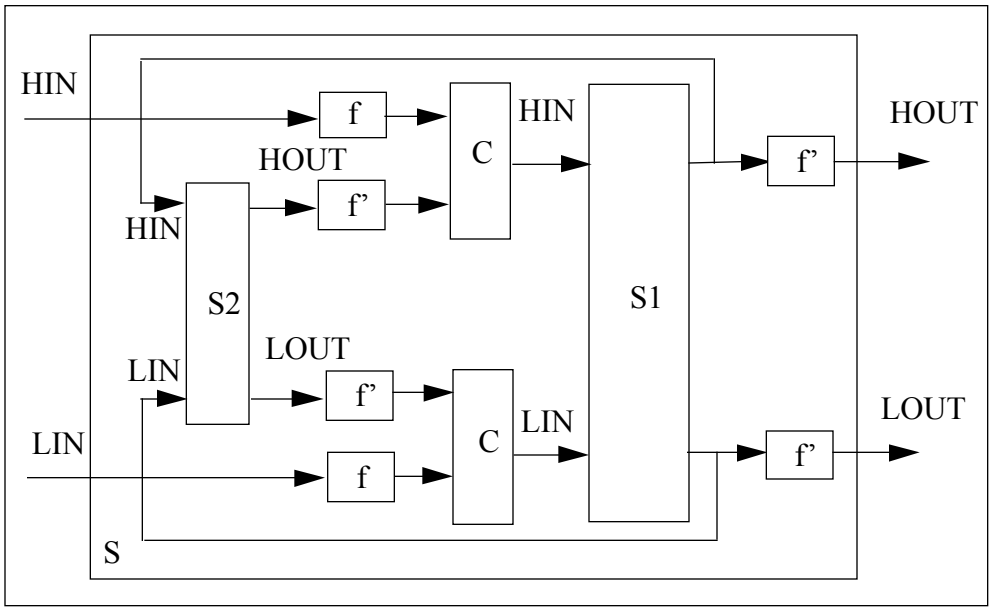


Figure 2.4 : Feedback Composition of S1 and S2

2.4 Proving Composition Properties of Information Flow Predicates

In this section we will develop a general method for proving the composition properties of information flow predicates that are expressed in the notation developed in earlier sections.

Let S1 be a system that satisfies the property $(A \Rightarrow B)$ and S2 be another system that satisfies the property $(C \Rightarrow D)$. Also, let $(R \Rightarrow T)$ be some property satisfied by both the systems S1 and S2. In order to prove that the composition S of S1 and S2 satisfies the property $(P \Rightarrow Q)$, the following approach can be used.

- Assume that $(P \Rightarrow Q)$ is true in S, which implies that S contains some set of traces *S* that satisfy the property P (if there are no such traces then the theorem is trivially true).
- From the definition of composition construct deduce that a set of traces *S1* are present in S1 and another set of traces *S2* are present in S2 from the set of traces *S* present in S derived in the previous step.
- From the property $(A \Rightarrow B)$ and the set of traces *S1* prove that there is a set of traces *S1'* in S1 and similarly from the property $(C \Rightarrow D)$ and the set of traces *S2* prove that the is a set of traces *S2'* in S2.
- If required use the additional property $(R \Rightarrow T)$ of the systems S1 and S2 and the set of traces *S1'* and *S2'* in S1 and S2 respectively to deduce the presence of the set of traces *S1"* in S1 and *S2"* in S2.
- Now, again from the definition of the composition construct prove that the set of traces *S1"* in S1 and *S2"* in S2 can be composed to produce a set of traces *S"* in S. Also prove that the set of traces *S"* in S implies that Q is true in system S.

It should be noted that the above method is a generalization of a number of proofs given in [MCL94] for composition theorems for a variety of composition constructs and

that they can all be derived using the above approach. In the following example we will show that Generalized Noninterference is composable under certain conditions using this method.

Example 2.1: Let S1 be a system that satisfies Generalized Noninterference i.e.,

$$\forall x \forall y \left(\begin{pmatrix} S1(x) \\ \wedge \\ S1(y) \end{pmatrix} \Rightarrow \exists p \begin{pmatrix} S1(p) \wedge (LOUT(p) = LOUT(x)) \wedge \\ (LIN(p) = LIN(x)) \wedge (HIN(p) = HIN(y)) \end{pmatrix} \right)$$

and S2 is another system that also satisfies the Generalized Noninterference i.e.,

$$\forall x \forall y \left(\begin{pmatrix} S2(x) \\ \wedge \\ S2(y) \end{pmatrix} \Rightarrow \exists p \begin{pmatrix} S2(p) \wedge (LOUT(p) = LOUT(x)) \wedge \\ (LIN(p) = LIN(x)) \wedge (HIN(p) = HIN(y)) \end{pmatrix} \right)$$

Let the cascade of S1 and S2 be S which means that it satisfies the properties

$$\forall x \left(S(x) \Rightarrow \exists y \exists z \begin{pmatrix} S1(y) \wedge S2(z) \wedge (HIN(x) = HIN(y)) \wedge \\ (LIN(x) = LIN(y)) \wedge (HOUT(x) = HOUT(z)) \wedge \\ (LOUT(x) = LOUT(z)) \wedge (LOUT(y) = LIN(z)) \wedge \\ (HOUT(y) = HIN(z)) \end{pmatrix} \right)$$

and

$$\forall y \forall z \left(\begin{pmatrix} S1(y) \wedge S2(z) \wedge \\ (LOUT(y) = LIN(z)) \wedge \\ (HOUT(y) = HIN(z)) \end{pmatrix} \Rightarrow \exists x \begin{pmatrix} S(x) \wedge (HIN(x) = HIN(y)) \wedge \\ (LIN(x) = LIN(y)) \wedge \\ (LOUT(x) = LOUT(z)) \wedge \\ (HOUT(x) = HOUT(z)) \end{pmatrix} \right)$$

Also let S1 and S2 satisfy the following property:

$$\forall x \forall y \left(\begin{pmatrix} S1(x) \wedge \\ S2(y) \end{pmatrix} \Rightarrow \exists z \begin{pmatrix} S2(z) \wedge (LOUT(x) = LIN(z)) \wedge \\ (LOUT(y) = LOUT(z)) \wedge (HOUT(x) = HIN(z)) \end{pmatrix} \right)$$

Now prove that the Cascade S of S1 and S2 satisfies Generalized Noninterference i.e.,

$$\forall x \forall y \left((S(x) \wedge S(y)) \Rightarrow \exists p \begin{pmatrix} S(p) \wedge (LOUT(p) = LOUT(x)) \wedge \\ (LIN(p) = LIN(x)) \wedge (HIN(p) = HIN(y)) \end{pmatrix} \right)$$

Proof:

Step1:

Assume that there are two traces x1 and y1 that satisfy the property $(S(x1) \wedge S(y1))$.

Step 2:

Now, from the definition of cascade and S(x1) we can deduce that there are traces x11 and x12 that satisfy the following property:

$$\left(\begin{array}{c} S1(x11) \wedge S2(x12) \wedge (HIN(x1) = HIN(x11)) \wedge \\ (LIN(x1) = LIN(x11)) \wedge (HOUT(x1) = HOUT(x12)) \wedge \\ (LOUT(x1) = LOUT(x12)) \wedge (LOUT(x11) = LIN(x12)) \wedge \\ (HOUT(x11) = HIN(x12)) \end{array}\right)$$

Again from the definition of cascade and S(y1) we can deduce that there are traces y11 and y12 that satisfy the following property:

$$\left(\begin{array}{c} S1(y11) \wedge S2(y12) \wedge (HIN(y1) = HIN(y11)) \wedge \\ (LIN(y1) = LIN(y11)) \wedge (HOUT(y1) = HOUT(y12)) \wedge \\ (LOUT(y1) = LOUT(y12)) \wedge (LOUT(y11) = LIN(y12)) \wedge \\ (HOUT(y11) = HIN(y12)) \end{array}\right)$$

Step 3:

Now the traces x11 and y11 in S1 and the definition of Generalized Noninterference of S1 we can deduce that there are two traces p and q in S1 that satisfy the following properties:

$$\left(\begin{array}{c} S1(p) \wedge (LOUT(p) = LOUT(y11)) \wedge \\ (LIN(p) = LIN(y11)) \wedge (HIN(p) = HIN(x11)) \end{array}\right)$$

and

$$\left(\begin{array}{c} S1(q) \wedge (LOUT(q) = LOUT(x11)) \wedge \\ (LIN(q) = LIN(x11)) \wedge (HIN(q) = HIN(y11)) \wedge \end{array}\right)$$

Step 4:

Now the trace p in S1 and the trace y12 in S2 and the property of S1 and S2 imply that there is a trace p' in S2 that satisfies the following property:

$$\begin{pmatrix} S2(p') \wedge (LOUT(p) = LIN(p')) \wedge (HOUT(p) = HIN(p')) \\ (LOUT(p') = LOUT(y12)) \end{pmatrix}$$

Also the trace q in S1 and the trace x12 in S2 and the property of S1 and S2 imply that there is a trace q' in S2 that satisfies the following property:

$$\begin{pmatrix} S2(q') \wedge (LOUT(q) = LIN(q')) \wedge (HOUT(q) = HIN(q')) \\ (HOUT(q') = HIN(x12)) \end{pmatrix}$$

Step 5:

Now, from the trace p in S1 and the trace p' in S2 and the definition of cascade operation we can deduce that there is a trace r in S that satisfies the following property:

$$\begin{pmatrix} S(r) \wedge (HIN(r) = HIN(p)) \wedge (LIN(r) = LIN(p)) \wedge \\ (LOUT(r) = LOUT(p')) \wedge (HOUT(r) = HOUT(p')) \end{pmatrix}$$

Also from the trace q in S1 and the trace q' in S2 we can deduce form the definition of cascade operation that there is a trace s in S that satisfies the following property:

$$\begin{pmatrix} S(s) \wedge (HIN(s) = HIN(q)) \wedge (LIN(s) = LIN(q)) \wedge \\ (LOUT(s) = LOUT(q')) \wedge (HOUT(s) = HOUT(q')) \end{pmatrix}$$

Now, substituting the values for HIN(q), LIN(q), LOUT(p) and HOUT(p) from the properties derived before, the above statements reduce to:

$$\begin{pmatrix} S(r) \wedge (HIN(r) = HIN(x1)) \wedge (LIN(r) = LIN(y1)) \wedge \\ (LOUT(r) = LOUT(y1)) \wedge (HOUT(r) = HOUT(p')) \end{pmatrix}$$

and

$$\begin{pmatrix} S(s) \wedge (HIN(s) = HIN(y1)) \wedge (LIN(s) = LIN(x1)) \wedge \\ (LOUT(s) = LOUT(x1)) \wedge (HOUT(r) = HOUT(q')) \end{pmatrix}$$

Hence the cascade S of S1 and S2 as specified above satisfies Generalized Noninterference.

The above proof process can be summarized by the following figure.

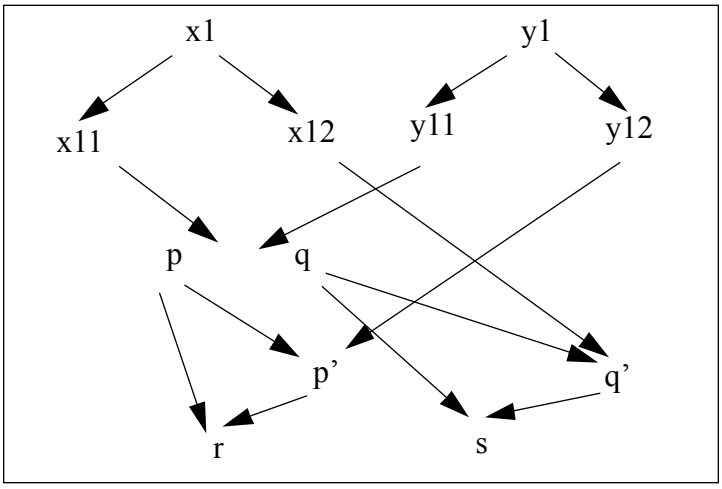


Figure 2.5 : Proof Procedure for Composition Properties

Here the property to be proved about the composed system S is $(P \Rightarrow Q)$. From P, it has been deduced that there are two traces x1 and y1 in S. From x1 in S, it was deduced that there are traces x11 and x12 in S1 and S2 respectively and from y1 in S, it was deduced that there are traces y11 and y12 in S1 and S2 respectively by the definition of composition construct. From the definition of Noninterference and the traces x11 and y11 in S1 it was deduced that there is a trace p in S1. Now from the property of S1 and S2 and the trace p in S1 and y12 in S2, it was deduced that the trace p' is present in S1. Also, from the property of S1 and S2 and the traces q in S1 and x12 in S2, it was deduced that the trace q' is present in S2. Now from the definition of composition construct and the traces

- p in S1 and p' in S2 we can deduce that there is a trace r in S and
- q in S1 and q' in S2 we can deduce that there is a trace s in S.

The traces r and s in S imply that it satisfies Generalized Noninterference.

All the other theorems about the properties of composition constructs in [MCL94] can be proved in a systematic manner using the above approach.

2.5 Automated Proof of the Composition Properties

The framework given in the previous section can be easily automated since each of the steps correspond to standard theorem proving rules as shown below:

- step 1 involves skolemizing (replacing a universally quantified variable with a constant usually referred to as skolem constant) the property to be proved.
- step 2 involves quantization (replacing an existentially quantified variable with a unique constant that has not appeared before) followed by skolemization of the definition of the composition operation.
- step 3 involves quantization followed by skolemization of the information flow properties of the systems S1 and S2.
- step 4 involves quantization followed by skolemization of the property satisfied by S1 and S2.
- step 5 is the quantization followed by skolemization of the definition of the composition operation.

In Appendix A, we give the specification and the proof of the above theorem in the PVS theorem proving system.

2.6 Conclusions

We gave a logical framework for specifying and reasoning about the possibilistic security properties that can be expressed in the form ($A \Rightarrow B$). We also give a framework for reasoning about the composition of these properties. This approach is not just limited to information flow security properties that are expressed as interleaving functions on traces but can be used to reason about any property that is in the appropriate form. Moreover, the rich logical formalism underlying predicate logic makes the reasoning process simple and elegant and facilitates use of automatic theorem proving systems.

A policy is a temporary creed liable to be changed, but while it holds good it has to be pursued with apostolic zeal

- Mohandas K . Gandhi

Chapter 3

Formal Specification of Information Flow Policies

Unambiguous and precise specification of security policies is an important element of the design of security critical systems. Towards this end, a formal specification method based on temporal logic is developed and its semantics is given in terms of a state machine model. The utility of this method is demonstrated by using it to specify some common security policies.

3.1 Introduction

Security policies are important to security-critical systems. A formal and unambiguous specification of the security policy can help the system designers understand the policy, and enable the evaluators to verify that the system enforces it correctly using formal analysis techniques.

A number of security policies have been proposed in the literature such as the MLS policy [BL73], the Chinese Wall security policy [BN89] and the Clarke-Wilson

Commercial security policy [CW87]. These policies, as defined, are representatives of a class of policies. For example, the MLS policy is a generic policy that is used primarily in military systems — but no system enforces it exactly as defined in the literature. Major military information systems, such as CMW [CFG87], MMS [MCL84] and LOCK [BKY85], enforce policies that resemble the MLS policy at a very high level of abstraction. In these systems, the specification of the security policy is not independent of functional description of the system. As a result:

- the security policy is more difficult to identify and understand, and
- the verification process is more difficult.

For these reasons, we develop a formal specification language that can be used to specify the information flow requirements of the security policy independent of the system specification. The formal specification of the access control requirements of the security policies are dealt with in the next chapter.

3.2 Specification of Information Flow Security Policies

An information flow security policy consists of a set of information classes and a set of restrictions on the way in which information can flow among these classes. At the level of policy specification, the meaning of the term *information flow* is not important and is treated as an uninterpreted predicate (in fact the information flow predicates defined in the previous chapter are used to assign meaning to this term). Of interest, however, is the effect of different information flows on the state of the system.

Information systems are usually represented using state transition systems, making security policies properties of sequences of transitions. The fact that security policies are properties of sequences of transitions suggests that they can be specified using temporal logic.

3.2.1 Temporal Logic

Temporal logic was proposed by Pneuli [PN77] as a tool for specifying and verifying the properties of concurrent programs. Considerable work has been done on the theory of temporal logic, and it has been applied in several areas other than concurrent program verification. In our work we use temporal logic in the formal specification of information flow policies.

Temporal logic differs from traditional logic by including temporal operators in addition to the traditional logical operators. These temporal operators are used to express the properties of the system in relation to time. Different forms of temporal logic exist based on different interpretations of time. A common version is Linear Time Temporal Logic (LTL) which views each time instant as having a unique successor. This logic includes the temporal operators $\Box$ (always) and $\Diamond$ (eventually) and can be used to reason about structures that are linear sequences. Another form of temporal logic is Branching Time Temporal Logic (BTL), which assumes that each instant in time can have several successors, each corresponding to a different future. This logic has the path quantifiers **A** (for all paths) and **E** (for some path) in addition to the temporal operators. This logic can be used to reason about structures that are infinite trees such as the computation tree of a non-deterministic program.

Another kind of temporal logic is Partial Order Temporal Logic — used to deal with structures that are partial orders. These partial orders often arise in the execution of concurrent programs. This logic contains past operators $\overline{\Diamond}$ (at some point in the past) and $\overline{\Box}$ (at all times in the past) in addition to the temporal operators and path quantifiers. It can be seen that POTL is at least as powerful than LTL and BTL by the fact that it subsumes both LTL and BTL.

Temporal logic is used in the verification of properties of programs in the following manner [WOL87]:

- an axiomatic system is specified to relate the program constructs and the temporal formulas,
- the property to be verified is specified as a temporal formula, and
- the axiomatic system is applied to the program, proving the desired formula.

We intend to use a similar approach to verify that system specifications enforce security policies.

The LTL and BTL are used for specifying the properties of a system with respect to the events occurring in the current state or some future state (hence the linear structure and infinite tree structure of interpretation for LTL and BTL respectively). Security policies which are properties of a system that depend on the events in the *past*, *present,* and *future,* cannot be expressed using LTL or BTL. POTL, on the other hand, with its past and future temporal operators, is ideally suited for the specification of such security policies.

3.2.2 A Specification Language for Security Policies

A security policy, SP, can be specified as a four-tuple, (C, Op, P, A), where

- C is a finite set of information classes,
- Op is a set of relations, including the equality relation (=), on information classes in C,
- P is a finite set of primitive propositions. These propositions are the information flow relations over the information classes. For example, a proposition may be of the form (A $\rightarrow$ B) where A and B are information classes and $\rightarrow$ is an information flow operator which denotes a predicate discussed in Chapter 2.

- A is the set of policy statements as defined below.

Policy statements are temporal logic formulas containing:

- the constants *true* and *false*.
- A set of variables ranging over the set of information classes C.
- The Boolean connectives $\neg$, $\wedge$, $\vee$ and $\Rightarrow$.
- The Boolean quantifiers $\forall$ and $\exists$ used to quantify over information classes.
- The temporal operators $\bar{o}$ (previous) and $\overline{\cup}$ (since), o (next), $\cup$ (until).
- The path quantifiers **A** and **E**.

The policy statements also contain several other operators that can be defined in terms of the basic temporal operators listed above. These operators are:

- $\Diamond$p (eventually in future p) abbreviation for (true $\cup$ p)
- $\Box$ p (always in the future p) abbreviation for $\neg\Diamond\neg$p
- $\bar{\Diamond}$p (sometime in the past p) abbreviation for (true $\overline{\cup}$ p)
- $\overline{\Box}$ p (always in the past p) abbreviation for $\neg\bar{\Diamond}\neg$p.

Definition: A state formula is defined as follows:

- A primitive formula is a state formula.
- If A and B are information classes or variables then A $\times$ B is a state formula where $\times \in$ Op i.e., $\times$ is an operation defined on the information classes.
- If X and Y are state formulas then so are $\neg$X, X $\wedge$ Y, X $\vee$ Y and X $\Rightarrow$ Y.
- If X and Y are state formulas then so are ($\bar{o}$ X) and (X $\overline{\cup}$ Y).
- If X is a state formula and v is a class variable then (($\forall$v) X), (($\exists$v) X) are state for-

mulas.

- If X is a path formula then **A**X and **E**X are state formulas.

Definition: A path formula is defined as follows:

- If X and Y are state formulas then (o X) and (X $\cup$ Y) are path formulas.

Definition: The policy statements are a set of state formulas that the initial state must satisfy.

An information flow policy that does not specify any restrictions on the flow of information between different information classes is the least restrictive policy since any implementation trivially satisfies it. This least restrictive policy is denoted by $\perp$.

3.2.3 Formal Specification of Security Policies

In this section we formally specify some common security policies using the language that was developed in the previous section.

Example 3.1: In the classic Multi-Level security policy [LAN81], there are some number of information classes over which a partial order is defined. Information is permitted to flow between two classes only if they are properly ordered with respect to one another. A typical MLS policy used by the military is:

- The set of classes C is {Top-Secret, Secret, Classified, Un-Classified}.
- The set Op is { $\leq$, = } where
  - the relation $\leq$ imposes a total order on these information classes, and
  - = has the usual meaning.
- The set of primitive propositions are (X $\rightarrow$ Y) for every information class X and Y.

- The policy statement is:

$$\mathbf{A}\Box\ (\forall X, Y \times (\ (X \rightarrow Y)\ \Rightarrow\ (X \leq Y)\ )\ )$$

That is, information can flow from a Class to the same or a higher Class; it does not say anything about whether an implementation must allow that flow; thus a system that does not permit any information flow trivially satisfies this policy. To disallow such trivial implementations we can add a statement specifying that the information flow between two information classes be permitted in at least one path from the initial state.

$$\forall X, Y \times (\ (X \leq Y)\ \Rightarrow \mathbf{E}\ \Diamond\ (X \rightarrow Y)\ )$$

That is, the initial state should be such that for every information class X which is less than or equal to another class Y, there should be a path containing a state in which information flows from X to Y.

Alternatively, we can impose a more restrictive condition requiring that every reachable state satisfy the above condition i.e.,

$$\mathbf{A}\Box\ (\forall X, Y \times (\ (X \leq Y)\ \Rightarrow \mathbf{E}\ \Diamond\ (X \rightarrow Y)\ )\ )$$

Finally, a different sort of requirement might be that information flow between two classes always be possible. This can be specified as follows:

$$\mathbf{A}\Box\ (\forall X, Y \times (\ (X \leq Y)\ \Rightarrow \mathbf{A}\ \Diamond\ (X \rightarrow Y)\ )\ )$$

It should be noted that the above specification of MLS is something that an implementor or verifier of an information system would prefer over the informal specifications usually found in the literature. Also, the statements about the additional functional requirements imposed on the information flows cannot usually be expressed using other formalisms in such a compact and precise manner.

Example 3.2: The Chinese Wall security policy [BRE89] groups entities into *conflict of interest classes* and states that information can flow from at most one entity belonging to a conflict of interest class to any given entity. Given two banks, Bank1 and Bank2, and a consultant Cons information can flow either from Bank1 to Cons or from Bank2 to Cons *but not both* (here Bank1 and Bank2 belong to a conflict of interest class). This policy can be specified as:

- The set of information classes, C, is {Bank1, Bank2, Cons}.
- The set of relations, Op, is $\{=_c, =\}$ where
  - $=_c$ holds between two classes if they belong to the same conflict of interest class. Here (Bank1 $=_c$ Bank2 and Bank2 $=_c$ Bank1). Note that for any class X, it is not the case that (X $=_c$ X).
  - = has the usual meaning.
- The primitive propositions are of the form (X $\rightarrow$ Y) for every X and Y in C.
- The policy statements are:

$$\mathbf{A}\Box\ (\ (Bank1 \rightarrow Cons) \Rightarrow \neg(\mathbf{E}\ \Diamond\ (Bank2 \rightarrow Cons))\ )$$

$$\mathbf{A}\Box\ (\ (Bank2 \rightarrow Cons) \Rightarrow \neg(\mathbf{E}\ \Diamond\ (Bank1 \rightarrow Cons))\ )$$

In the most general case the above policy statements can be expressed as:

$$\mathbf{A}\Box\ (\forall X, Y, Z \times (\ (\ (X \rightarrow Y)\ \wedge\ (Z =_c X)\ ) \Rightarrow \neg(\mathbf{E}\ \Diamond\ (Z \rightarrow Y))\ )\ )$$

where X, Y and Z are information class variables and $=_c$ is as defined above.

This example shows that it is easy to express security policies that are discretionary in nature, where information can flow between different classes only under some dynamic conditions, through the use of existential path quantifier and the eventuality temporal operator.

Example 3.3: Consider a system that charges its users for access to a data-base. In this case the information flow from the data-base to the user is allowed only if the service administrator has access to the credit card information of the user. This policy can be expressed as:

- The set of information classes C is {database, user, usercc, administrator}.
- The set of Operators Op is {=} where = has the usual meaning.
- The primitive propositions are $(X \rightarrow Y)$ for every X and Y in C.
- The policy statement is:

$$\mathbf{A}\Box\ (\ (database \rightarrow user) \Rightarrow (usercc \rightarrow administrator)\ )$$

That is, it should always be the case that if information flows from the database to the user, then information should flow from usercc to the administrator at the same time.

Example 3.4: Consider a system where a government department is allowed to give the information about a contract to at most one of the many competing agencies, $Agency_i$ where i ranges from 1 to n. Such a policy can be expressed in the above language as:

- The set of information classes C is $\{Agency_1, Agency_2, ..., Agency_n, Govt\}$.
- The set of relations, Op, is $\{=_c, =\}$ where
  - $=_c$ specifies that two information classes belong to the same conflict of interest class, here $Agency_i =_c Agency_j$ for all i and j, $i \neq j$, and
  - = has the usual meaning.
- The primitive propositions P are of the form $(X \rightarrow Y)$ for every X and Y in C.
- The policy statement is:

$$\mathbf{A}\Box\ (\forall X, Y \times (\ (\ (Govt \rightarrow X) \wedge (X =_c Y)\ ) \Rightarrow \mathbf{A}\Box \neg (Govt \rightarrow Y)\ )\ )$$

That is, if information flows from *Govt* to $Agency_i$ then information cannot flow from *Govt* to a different $Agency_j$ ever.

Example 3.5: A MLS policy that includes a trusted entity that is allowed to violate the MLS restrictions can be specified as:

- The set of classes C is {Top-Secret, Secret, Classified, Un-Classified, Trusted}.
- The set Op is { ≤, = } where
  - ≤ imposes a total order on the information classes except the class Trusted, and
  - = has the usual meaning.
- The set of primitive propositions are (X → Y) for every information class X and Y.
- The policy statement is:

$$\mathbf{A}\Box\ (\forall X, Y\ (\ (X \rightarrow Y) \Rightarrow\ (\ (X \leq Y) \vee (X = Trusted) \vee (Y = Trusted)\ )\ )\ )$$

That is, information is allowed to flow from lower class to a higher class or in either direction between any class and the class *Trusted*.

Note that the above policy allows information to flow indirectly between information classes in a manner that violates the MLS policy conditions — as long as they occur through an entity belonging to the *Trusted* information class. This specification reveals yet another variation on the ubiquitous MLS policy.

Example 3.6: Consider a system consisting of three information classes {BkGrnd, Cur, User}. The security policy is that information of class *Cur* can be given to a *User* only if *User* has the *BkGrnd* information. Such a policy can be specified as follows:

- The set of classes, C, is {BkGrnd, Cur, User}.
- The set Op {=}, where = has the usual meaning.

- The set of primitive propositions are (X → Y) for every information class X and Y.
- The policy statement is:

$$\mathbf{A} \quad ((Cur \rightarrow User) \Rightarrow \overline{\Diamond} (BkGrnd \rightarrow User))$$

The above expression states that information is allowed to flow from *Cur* to *User* only if at some time in the past information flow occurred from *BkGrnd* to the *User*. This example demonstrates the use of past operators in expressing security policies. Note that this policy cannot be expressed with future operators since the information flow from *BkGrnd* to *User* only *enables* the information to flow from *Cur* to *User* but does not say whether it actually occurs or not.

The preceding examples show that the policy specification language developed here is expressive enough to specify a large class of security policies and formal enough to do so in a precise and unambiguous manner.

3.3 A Model for Information Flow Policies

An information system that has to enforce a security policy can be modeled as a state machine that constitutes an interpretation of the formulas in that policy. If the temporal formulas of the security policy hold under this interpretation, then the corresponding system satisfies the policy. In this section we present the semantics of the language specified in the previous section over a system modeled as a state machine.

3.3.1 State Machine Model

An information system that enforces a security policy, SP = (C, Op, P, A), can be specified as a state transition machine M = (S, E, τ, s_0) where

- S is the set of all states described by means of state variables,
- E is the set of entities in the system,

- τ is the state transition relation, $\tau \subseteq S \times S$, and
- s_0 is a state in S that is the starting state.

Definition: Given a security policy SP = (C, Op, P, A) and a state machine M = (S, E, τ, s_0) an *Interpretation* I is of the form (η, π), where

- η is a class assignment function of the form η: E $\rightarrow$ C that assigns an information class to every entity.
- π : S $\rightarrow$ **P** (P^{η}) is an interpretation function that gives the validity of all the propositions that can be derived from the primitive propositions in a given state. Here P^{η} denotes the set of all propositions obtained by replacing every information class c in the proposition p $\in$ P by an entity e where η(e) = c.

In Chapter 4 we will show that the formal specification language Z can be used to specify a state machine that includes the interpretation of the security policy it has to enforce.

Definition: Given a state machine M = (S, E, τ, s_0), a security policy SP = (C, Op, P, A) that must be enforced by M, and an interpretation I = (η, π), an *interpretation instance* is defined as (η', π) where η' is the class assignment function (η' : E $\rightarrow$ C) such that for every information class c there is at most one entity e such that, η'(e) = η(e) = c.

The *interpretation instance* of an interpretation defines another interpretation where each information class has just one entity associated with it, permitting the truth value of any formula that is specified in terms of information classes to be determined. It should be noted that an interpretation I of a security policy SP in a state machine M has a number of instances with different class assignment functions.

Definition: Given a state machine $M = (S, E, \tau, s_0)$ that satisfies the security policy $SP = (C, Op, P, A)$, a *path* is defined to be a state machine $M' = (S, E, \tau', s_0)$ where τ' is a mapping of the form $\tau' : S \rightarrow S$ such that $(\tau'(s) \in \tau(s))$ for all s in S.

A path in a state machine gives the linear sequence of state transitions that are possible from the initial state. Note that every state in the transition function of a *path* has only one next state.

The truth value of a formula can be inductively defined as shown in the following definition.

Definition: A formula X of a security policy is said to be true in a state machine $M = (S, E, \tau, s_0)$ under an interpretation instance $I' = (\eta', \pi)$ of an interpretation $I = (\eta, \pi)$, written as $M_{I'} \vdash X$ (read the $\vdash$ symbol as *satisfies*) if X is

- $(A \rightarrow B)$ and there exist entities a and b such that $(a \rightarrow b) \in \pi(s)$ where $(\eta'(a) = A)$ and $(\eta'(b) = B)$.
- (A op B) where op $\in$ Op and (A op B) is *true*.
- $(P \wedge Q)$ and $M_{I'} \vdash P$ and $M_{I'} \vdash Q$.
- $(P \vee Q)$ and $M_{I'} \vdash P$ or $M_{I'} \vdash Q$.
- $((\forall v)P)$ and for every information class c, $M_{I'} \vdash P \mid_{v = c}$.
- $((\exists v)P)$ and there is an information class c such that, $M_{I'} \vdash P \mid_{v = c}$.
- $\neg P$ and $M_{I'} \vdash P$ is not true.
- **A**P and for any path M' derived from M, $M'_{I'} \vdash P$.
- **E**P and for some path M' derived from M, $M'_{I'} \vdash P$.
- $(\bar{o}\ P)$ and for all states s'' such that $\tau(s'') = s$ and $M''_{I'} \vdash P$ where

M'' = (S, E, τ, s'').

- (P $\overline{\cup}$ Q) and for every sequence

$$\left(s_0, \tau(s_0), \tau(\tau(s_0)), \ldots, \tau^k(s_0) \right)$$

where ($\tau^k(s_0) = s$) and ($\tau^0(s_0) = s_0$) the following conditions are true:

- There is a j where $0 \leq j < k$, such that
  - $M^i_{I'} \vdash P$ for all $0 \leq i \leq j$, and
  - $M^l_{I'} \vdash Q$ for all $j < l \leq k$.

where M^p is (S, E, τ, $\tau^p(s_0)$) and $\tau^p(s)$ is the p^{th} element in the sequence

$$(s, \tau(s), \tau(\tau(s)), \ldots)$$

Definition: A path formula X is said to be true in the path M' = (S, E, τ', s) of the machine M = (S, E, τ, s_0) under an interpretation instance I' = (η', π) written as $M_{I'} \vdash X$ if X is of the form:

- (o P) and $M''_{I'} \vdash P$ where M'' = (S, E, τ', τ'(s)).
- (P $\cup$ Q) and for some $j \geq 0$, $M^j_{I'} \vdash Q$ and for all $0 \leq k < j$, $M^k_{I'} \vdash P$ where M^i is a machine (S, E, τ', $\tau'^i(s)$) and $\tau'^i(s)$ is the i^{th} element in the sequence

$$(s, \tau'(s), \tau'(\tau'(s)), \ldots)$$

Definition: A state machine M is said to enforce a security policy SP under an interpretation I if the policy statements of SP are true in M under every interpretation instance of I.

It should be noted that the interpretation function π is critical. Depending on the kind of security that is desired this function is selected appropriately.

Definition: A Security Policy is said to be *consistent* if there exists a state machine and an interpretation in which the policy axioms are true.

In this section we gave the formal semantics of the information flow policy specification language proposed in the previous section. It should be noted that the semantics given here are useful in understanding the precise meaning of a specification but do not provide an efficient method for verifying whether a state machine correctly enforces the security policy. This issue of providing an efficient method for verifying that a state machine actually enforces a policy will be dealt with in Chapter 4.

3.4 Related Work

Existing information flow policy specification formalisms can be broadly categorized into two classes:

- Modal logic based specifications,
- Operator based specifications.

In this section we compare these formalisms with our temporal logic based approach.

3.4.1 The Modal Logic Approach

In a modal logic approach, information is modeled as knowledge, as specified by modal operators whose semantics are given in terms of the behavior of the system. This approach has been used in [CUP93] for specifying security policies.

3.4.1.1 The Model

A system is modeled as <O, D, T, S, *A*> where

- O is the set of Objects.
- D is the domain of the values of objects.
- T is the set of time points.
- S represents the set of possible traces in the system where a trace specifies the value of all the objects at all time instants.
- A is the set of subjects in the system. A is a subset of the set R, called the roles, where R is the set of total functions from S into (**P** (O × T)). That is, the subjects are modeled by their knowledge of the values of the objects in different traces of the system.

Definition: Given a trace s and a subject A, ($s \lceil A_t$) denotes the set of values of objects known by subject A in s until time t.

3.4.1.2 Security Policy Specification

A security policy is a formula that is built out of a set of primitive formulas which are of the form Val(o, t, d) where (o, t, d) ∈ (O × T × D) and Val(o, t, d) indicates that the value of object o at time t is d and a set of operators:

- The usual Boolean connectives ¬, ∧, and ∨.
- Modal operator $K_{A, t}$ where A is a role and t is a time. $K_{A, t}\ \psi$ indicates that A, at time t, *knows* that ψ is true.
- Modal operator $R_{A, t}$ where A is a subject and t is a time. $R_{A, t}\ \psi$ indicates that A has *permission*, at time t, to know that ψ is true.

The semantics of the propositional and the Boolean formulas are specified in the usual way. The semantics of the modal operators are specified as follows:

- The formula $K_{A, t}\ \psi$ is true in a trace s iff ψ is true in every trace s' where ($s \lceil A_t$)

$= (s' \lceil A_t)$. That is, a role knows that ψ is true at some time t in a trace s if ψ is true in every trace s' that is indistinguishable from s.

- The formula $R_{A,t}\ \psi$ is true in a trace s if a subject A is assigned a role X such that $K_{X,t}\ \psi$ is true in s. This statement means that a user is *permitted* to know that ψ is true if there is a role X which knows that ψ is true.

The above logic can be used to express different kinds of security policies by identifying a group of formulas that possess certain properties. Some of the groups identified in [CUP93] are:

- Right(A, s, t) contains formulas whose truth value must be known to role A — i.e., for every formula ψ in Right(A, s, t), $K_{A,t}\ \psi$ should be true in s. This is used to express policies like the MLS and Chinese Wall policies.
- Prohib(A, s, t) contains formulas whose truth value should not be known to role A — i.e., for every formula ψ in Prohib(A, s, t), $(\neg K_{A,t}\ \psi \wedge \neg K_{A,t}\ (\neg\ \psi))$ should be true in s.

3.4.1.3 Comparison

In the modal logic approach of [CUP93], the notion of time is explicitly present in the model, whereas in a temporal logic based approach it is captured by the temporal operators. None of the security policies in the literature use absolute time and therefore this feature of modal logic approach is of questionable value in policy specifications. Also we observed that the abstraction provided by the temporal operators is adequate to represent the notion of relative ordering of events occurring in policy specifications. These abstractions also help the developer of a policy by eliminating the need to worry about extraneous details such as representation and inference properties of time. Moreover, the fact that the Right and Prohib formulas mentioned in the previous section are specified for every trace at each

point in time makes them quite large and unwieldy. In the case where the lifetime of the system is infinite it may even be impossible to enumerate them.

One of the drawbacks of the approach in [CUP93] is that it cannot be used to specify security policies where only part of a trace must satisfy a certain property. For example, it is difficult to specify a policy containing a requirement such as:

> *every legal state has the property that if information flows from A to B then information cannot flow from B to C in the future.*

In our formalism the above policy requirement can be easily specified as

$$\mathbf{A}\Box\ (\ (A \rightarrow B) \Rightarrow \mathbf{A}\Box\ (\neg\,(B \rightarrow C)\ )\ )$$

In [CUP93], policies are specified by grouping formulas into sets that have to satisfy some modal properties. For example, the sets Right and Prohib mentioned in the previous section are used to specify the policies that contain explicit permission and prohibition of information flow requirements. This sort of specification is more difficult to develop since the specifier has to be cognizant of the subtleties of the underlying trace semantics of the modal operators.

Another approach based on modal logic is the 'logic of security' [GMP93], in which policies are specified by the modal operators P (permission) and O (obligation) as well as the afore mentioned K (knowledge) operator. Here, P is used to specify the secrecy policies and O is used to specify the integrity policies. Once again, since the modal operators work on complete traces rather than the intermediate states in a trace, it is difficult to specify properties that need to be satisfied by some part of a trace.

3.4.2 Operator Based Model

A model for specifying information flow policies that contain information classes and a set of information flow operators, has been proposed in [FOL89a]. Based on this model a taxonomy of information flow policies is developed in [FOL89b].

3.4.2.1 The Model

The security model consists of a set of

- Information classes C and
- Information flow operators FO = {→, ý, }
  - (a → b) means information can flow from class a to class b,
  - (A ýa) means information can flow from any subset of A to class a, and
  - (A a) means information can flow from all the classes of the set A to the class a.

As in the temporal logic based approach that we proposed earlier, the meaning of the information flow operation is left uninterpreted.

3.4.2.2 The Policy Specification

Security policies are specified as a set of formulas where a formula is as follows:

Definition: A formula is of the form (A ýa) or (A a) where A is a set of information classes and a is an information class.

Definition: A security policy is specified as a set of formulas.

Example 3.7: The Multi-Level security policy can be specified as follows:

- The information classes are {Top-Secret, Secret, Classified, Un-Classified}.
- The policy formulas are:

- ({Un-classified} ýClassified)
- ({Un-classified, classified} ýSecret)
- ({Un-classified, Classified, Secret} ýTop-Secret)

[FOL89b] identifies different classes of information policies such as

- an *separation policy* that satisfies the following condition

$$\forall A1, A2, a \times((((A1 \quad a) \quad \in P) \wedge ((A2 \quad a) \quad \in P)) \Rightarrow ((A1 \cup A2) \quad a) \quad \in P))$$

- an *aggregation policy* that satisfies the following condition

$$\forall A1, A2, a \times((A1 \cup A2) \quad a) \quad \in P)) \Rightarrow (((A1 \quad a) \quad \in P) \wedge ((A2 \quad a) \quad \in P))$$

Other kinds of policies which are referred to as quasi-ordered policies and reflexive policies are identified in [FOL89b]. [FOL89b] also gives a high water mark mechanism for implementing different kinds of policies by converting the policies into a lattice.

3.4.2.3 Comparison

The work in [FOL89b] is the first attempt at classifying the information flow policies and proposing a taxonomy for them. However, the class of information flow policies that can be expressed using the above approach is limited due to the small number of information flow operators. For example, there is no way of expressing a policy requirement such as

$$\mathbf{A}\Box\ (\ (A \rightarrow B) \Rightarrow \mathbf{A}\Box\ (\neg\ (B \rightarrow C)\ )\ )$$

using the two information operators defined in [FOL89a].

An underlying assumption in the approach of [FOL89b] is that information flow policies are specified as sets of allowed information flows. In practice, this is not usually the case. Most of the security policies in the literature — MLS policy, Chinese Wall policy — are specified in terms of disallowed information flows. In many cases it is left to the

discretion of the implementor whether to actually allow all the information flows that do not violate the security policy. For example, the Chinese Wall policy only says that information should not flow from entities belonging to the same conflict of interest class to another entity. It does not say what information flows need actually occur to start with. Hence a formalism such as the one proposed earlier in this chapter, which includes the complete first order logic, is more expressive than the operator based model.

3.5 Conclusions

A Partial Order Temporal logic based specification mechanism, which can be used in the specification of information flow security policies, has been developed. This mechanism is successfully used to specify some common information flow security policies in a succinct and unambiguous manner. Formal semantics for this language are provided using a state machine model that allows one to precisely understand the meaning of the policy. This logic based approach is more expressive than other formalisms proposed in the literature for specifying security policies.

A commercial society whose members are essentially ascetic and indifferent in social rituals has to be provided with blue prints and specifications for evoking the right form for every occasion

- Marshall McLuhan

Chapter 4

Specification of Security Critical Systems

Access control models have been proposed for describing and analyzing the protection mechanisms of information systems. However, these models have rarely been used in practice for either purpose. In this chapter we take a critical look at existing access control models and identify some reasons for their lack of use. We propose that the specification language Z [DIL90] is more suitable for describing the protection mechanisms of information systems and show that temporal logic can be used to specify access control policies. We also develop a language for specifying the set of traces generated by a formal specification that can be used in the verification of its security properties.

4.1 Introduction

Access Control models played an important role in modeling and analyzing the security of computer systems in the 1970's. These models were used in the design of secure operating systems such as the UCLA Data Secure Unix [WAL80], and PSOS [LAN81]. They were also used in detecting security holes in extant operating systems such as Multics, where a

card reader daemon could install a Trojan horse in a user file space [JON78]. Although security holes similar to the one in Multics have been found in modern operating systems, access control models had very little role in discovering them. This is unfortunate, since a simple analysis of the protection mechanisms most likely could have found most of these holes in a systematic manner.

Access control models are state machines consisting of a set of

- entities,
- access rights between the entities, and
- transitions — the state changing operations.

The state transitions occur non-deterministically provided the appropriate access rights are present. Security analysis in these systems is concerned with the reachability of a state from the initial state. This is referred to as the *Question of Safety* and was shown to be undecidable in the general case [HAR76].

Access control models were first described by Lampson in [LAM74]. Later, a number of models were proposed that differ in expressive power and decidability of safety; these include the HRU model [HAR76], take-grant model [SNY81], Grammatical protection models [SNY81], Schematic Protection model [SAN88], Extended Schematic Protection model [AMM92], and Typed Access Matrix model [SAN92]. Although a number of interesting results about the *question of safety* exist with respect to these models, none of them have been used in practice to analyze the safety of the protection mechanisms of large scale systems.

The HRU model is the first and most general model. It consists of a set of Subjects and Objects and the protection state of the system is described by means of a two dimensional access matrix indexed by the entities. This matrix contains the set of rights

between the Subjects and Objects. Six primitive operations are defined on this access matrix (enter a right, delete a right, create a subject, create an object, destroy a subject, destroy an object) that can be combined in arbitrary ways to create new commands. The safety question in this model is the reachability of a specified state from the initial state with a given set of commands. It was shown in [HAR76] that this safety question is undecidable. The Schematic Protection model (SPM) is a restricted form of HRU model where the entities are associated with types (not just Subject and Object as in HRU model) and the operations are limited to creation of new entities and copying of rights. Although it is not as general as the HRU-model, SPM has better safety analysis techniques. The Typed Access Matrix model [SAN92] is similar to HRU except that the entities have types associated with them. The conditions for making the safety question decidable are tighter in this model than the corresponding ones of the HRU model.

In this chapter we investigate the reasons for the reluctance on the part of security practitioners to use access control models in security verification and propose an alternative approach for modeling the protection features of information systems.

4.2 Access Control Models and Security Verification

The general approach mandated by the *Trusted Computer Systems Evaluation Criteria* [ORA85] for building secure systems is illustrated by the following figure.

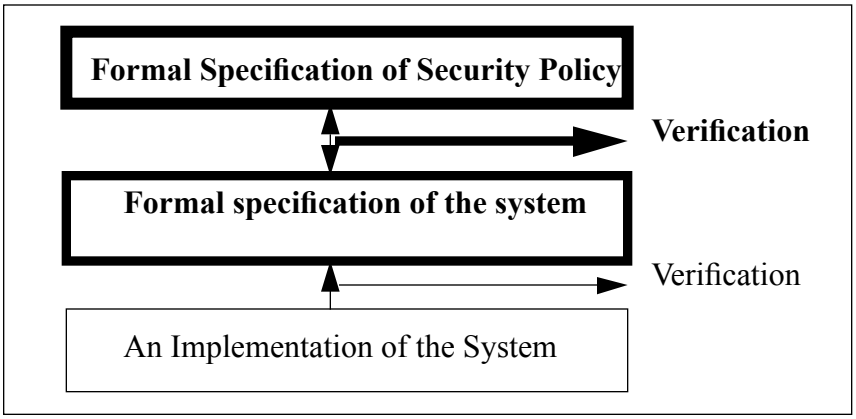


Figure 4.1 : The Security Verification approach

In the above figure, two separate verification processes are carried out:

- one between the security policy and the abstract specification of the system, and
- the other between the abstract specification of the system and the software or hardware that actually implements it.

This approach is both practical and productive since it decomposes the task of security verification into more manageable sub-tasks. Since most security holes are design flaws, we are interested in the verification of the security policy with respect to system specifications that contain the design aspects of the system. The verification between the actual implementation of the system and its specification is also important, of course; conventional program verification can be used for this purpose.

The formal specification of a system is useful in many ways. In fact, many security holes can be found just by writing the specification down. The use of a second, qualitatively different paradigm forces the specifier to have a deeper understanding of the system. It reveals things that would otherwise be obscure and tend to be ignored. This statement is supported by the experience in a large number of areas such as authentication protocols, network protocols, software engineering, formal methods. As an added benefit, if the specification is in a user friendly language, it can even be used as a reference manual to communicate the features of the system to its users.

In order to apply the approach shown in fig 4.1 to model and verify computer security, three steps need to be performed:

- the *security policy* must be specified as properties of the behavior of the system,
- the *specification of the system* must developed using a model such as an access control model
- the *verification* consisting of checking whether all the reachable states do indeed

possess the required properties, must be performed.

Traditional access control models are lacking in three important aspects that are crucial for specification and verification of security policies. These are:

- Abstraction mechanisms,
- Security Analysis, and
- Policy Specification.

In the following sections we elaborate on each of these issues by means of examples.

4.2.1 Abstraction Mechanisms

Access control models were initially proposed to model the protection of main memory between processes in multi-programmed systems [LAN81]. The advent of timesharing and networking changed the protection landscape drastically, but access control models have not changed to accommodate these developments.

Modern operating systems are complicated, with diverse features that affect the protection state in a number of ways. The abstraction mechanisms provided by the traditional access control models are primitive, making the task of modeling and analyzing protection features of information systems extremely difficult. For instance,

- There is no natural way to model the infamous suid (set user id) bit, or the environment variables of the Unix operating system — both of which have been the *root* of many a security problem.
- There is no natural way to enforce the concept of groups where the entities belonging to the same group possess the same access privileges. This notion of groups is important in secure systems since the Computer Systems Evaluations Criteria

[ORA85] of the Department of Defense mandates that systems provide facilities so that groups can be realized and their affect on the security of the system can be analyzed.

The above examples show that access control models cannot easily describe many real world systems making the task of using them to analyze security properties difficult.

4.2.2 Security Analysis

One of the primary goals of access control models is to enable formal analysis of the safety of systems that they model. As mentioned before, in the general case the question of safety is undecidable for these models. Therefore, these models make some assumptions about the system to make the safety question decidable. These assumptions typically are:

- restrictions on the form of the transition relation — so that the size of the protection state of the system is bounded. For example, in the SPM [SAN88], a newly created entity must be *less powerful* than its parent so that the number of possible states is finite in number.
- the set of transitions in the model need to satisfy the restoration principle; that is, any of the access rights that might be deleted can be reinstated. It means that the deletion of access rights does not have any affect on the protection state of the system. All the access control models mentioned earlier make this assumption.

Clearly the above assumptions are not realistic. For example, few systems satisfy the restoration principle — in Unix a user can change the ownership of a file he owns to another user but cannot get the ownership changed back without the cooperation of the new owner or the root.

The restrictions on the form of operations for bounding the size of the protection state of the system are also not very realistic. In fact, the problem it purports to address

doesn't really exist — in real systems it is usually not possible to indefinitely increase the size of protection state by creating an unbounded number of new entities as assumed by the create operation in many of the access control models.

Access control models contain two types of rights:

- *normal rights* used to perform operations such as the read, write, etc., and
- *control rights,* which effect the protection state of the system, such as take and grant in tg-model.

An assumption made by access control models is that the control rights can be copied just like normal rights. This assumption makes the security analysis quite difficult. But again this assumption is not true in practice; in most systems, a user is not allowed to copy control rights. For example, in Unix one cannot copy the control rights like ownership but can only transfer it. This kind of realistic restriction on the control rights has good potential to simplify security analysis.

4.2.3 Policy Specification

The security policies for which access control models provide useful analysis techniques are usually of the form that certain set of states, labeled unsafe, should not be reachable from the initial state. Unfortunately, these kinds of policies are not representative of the real world since the safety of a state is often dependent on how it is reached from the initial state. A state that is safe when reached through one sequence of actions may be unsafe when reached through a different sequence. For example, a state where user A has read access to user B's file may or may not be safe. A security policy might state that such a state is safe if B explicitly authorizes the read access for A, otherwise it is not.

It is widely accepted that security policies and the mechanisms used to enforce them are orthogonal [WUL80] and therefore should be treated as such. Although a given access

control model can answer questions about the reachability of certain unsafe states, ideally we would like it to answer questions about an arbitrary policy. For example, the take-grant model can be used to determine whether an entity can obtain certain rights to another entity but cannot address whether information can flow between two entities. This is because the take-grant model does not contain an interpretation for the term *information flow*. Once this term is assigned a meaning in the domain of the take-grant model, it is possible to derive the truth value of predicates that depend on this term in any state of the system. Ideally, the policy specification language should be independent of the access control model and the mapping between them should define the security requirements of the system under consideration.

The Bell and LaPadula model (BLP) [BL74] is an access control security model that enforces the MLS policy by the following two restrictions on the actions of the entities:

- An entity cannot read an entity at a higher level than itself (no *read-up*) and
- An entity cannot write to an entity at a lower level than itself (no *write-down*),

This model was criticized by McLean [MCL87] because it includes systems that are insecure according to the MLS policy — a system containing an operation that changes the levels of all the entities to a single level. In response [BEL88], it was argued that the BLP model is in fact correct as defined due to the presence of the *tranquility principle* which states that the levels of active entities cannot change. This confusion is due to the fact that the BLP model has become synonymous with the MLS policy, while in reality BLP is simply an access control model that attempts to enforce the MLS policy. We feel that it is both a prerogative and a responsibility of the security policy to specify whether or not the classifications associated with the entities can be altered. Therefore we contend that the BLP model with the tranquility principle and the same model without it satisfy different policies — both unfortunately referred to as the MLS policy.

In this section we argued that access control models are not suitable for developing system specifications that can be used in security verification. In the following sections we will propose a framework for specifying the protection features of information systems that is amenable to verification.

4.3 A New Approach to System Specification

A major limitation of traditional access control models is the underlying assumption that the world is composed of only subjects, objects, access rights and the protection state access matrix. These abstraction mechanisms provided by the access control models are inadequate for modeling all the protection features of modern information systems. A richer set of abstraction mechanisms, which can also be used in the verification process, is required for describing the protection features of these systems. In this section we propose that the specification language Z, with its rich set of abstraction mechanisms based on set theory and predicate calculus, can be used to describe the protection features of information systems.

Formal specification notations can be classified into two categories[WIN90]:

- model based specification and
- property based specification.

In a model-based approach, the behavior of the system is specified by first constructing a model of the system in terms of abstract structures such as sets, relations, functions, sequences, etc., and then describing the operations by their affect on the system state. In property-based methods, a system is described indirectly by means of a set of properties that it is required to satisfy. These two approaches, although quite different at the syntactic level, are equivalent in expressive power at the semantic level. Model-based notations include Z, VDM, InaJo, Petrinets, CSP and Unity and property based languages include Larch, Clear and Act One.

The protection features of information systems can be modeled quite naturally as state machines, where the state description consists of information such as labels regarding the security characteristics of entities or access rights in tables like access matrices. These structures can be described conveniently by the abstraction mechanisms provided by a model based specification language such as Z [DIL90]. The operations in the system can be modeled as state transitions in the system.

It is easy to see how this approach can be used in describing the protection features of an operating system. For example, in Unix:

- Users can be modeled by entities with properties that describe their environment and
- Files can be described by entities with properties that specify the access rights, owner id, suid bit, etc.

In order for this form of specification to be useful in security verification, we need to develop a framework in which security properties, expressed in temporal logic, can be verified. This will be dealt with in the next chapter. In the following sections we show how the specification language Z can be used to specify the protection features of information systems and also specify the behaviors of interest in a compact way over which the temporal properties can be interpreted.

4.3.1 Overview of Z

Z is a specification language based on first-order predicate calculus and typed set theory. The model of computation is a non-deterministic state transition machine. The state in this model consists of a set of state variables and the transitions are described as the changes that can occur to the values of these variables.

The specification of a system in Z consists of two parts:

- State Description — The state, in Z, is described using a set of state variables. These variables have a type associated with them where the type can be a basic one such as integer or a more complicated one such as a relation or a function. The values of the state variables describe the state of the system at any instant.
- Operation Description — The operations are described by specifying their effect on the state variables of the system. These operations describe the conditions under which they can occur, referred to as pre-conditions, and their affect on the state of the system, referred to as post-conditions.

The basic unit of specification in Z is a schema that consists of:

- A declaration part where state variables are declared and
- A body part where the pre-conditions and affects of the operation described by the schema are listed in a declarative manner.

By convention, in schemas describing operations, the unprimed variables (e.g. user) represent are the state variables before the operation occurs and primed ones (e.g. user') represent the variables after the occurrence of the operation. Hence a condition in a schema that does not include any primed variables is a pre-condition and any condition that contains a primed variable is a post-condition.

The computational model in Z is non-deterministic — any operation whose pre-condition is true may be executed. Note that a non-deterministic model of computation does not mean that the affect of a given operation is non-deterministic. For more details about the features of Z see Appendix B.

An important feature of security policies of information systems is that the ability to perform operations is bestowed upon only a select set of objects in the system. In order to capture this feature in specifications, we extend Z, by defining a variable that ends in a

'#' to denote an input variable not under the control of the entity invoking the operation. It is assumed that such a variable is assigned a value by the underlying system at the time of operation invocation.

4.3.2 An Example

The take-grant model [SNY81] is an access control model used to describe the protection mechanisms in operating systems. This model can be specified in Z by giving schemas for the state description, the state changing operations, and the initial state. The schema for state description is given below.

$$
\begin{array}{|l}
\hline
tg - model \\
\hline
Entities : \mathbb{P}\, NAME \\
Subjects : \mathbb{P}\, NAME \\
Objects : \mathbb{P}\, NAME \\
AM : NAME \times NAME \pfun \mathbb{P}\, RIGHT \\
\hline
Subjects \cup Objects = Entities \\
Subjects \cap Objects = \phi \\
\forall\, x, y \bullet (((x, y) \in \operatorname{dom} AM) \\
\qquad \Leftrightarrow ((x \in Entities) \land (y \in Entities))) \\
\hline
\end{array}
$$

This schema defines the state as consisting of four state variables:

- *Entities*, *Subjects,* and *Objects* which are each a set of NAME (the emphasized P denotes power set)
- *AM* is a partial function mapping pairs of names to sets of rights (the cross is the cross-product and the arrow with a slash in the middle denotes a partial function).

Here, NAME is assumed to be a string of characters and RIGHT is the set {t, g, r, w} denoting the *take*, *grant*, *read* and *write* rights respectively.

The meaning of each line in the body part of the schema is:

- An entity must be either a *Subject* or an *Object*.

- An entity cannot be both a *Subject* and an *Object.*
- AM is a two-dimensional array consisting of a row and a column for every entity and also every row and column in this AM corresponds to some entity. (The dot is the separator between the quantified variables and the expression).

The three operations take-rule, grant-rule and create-rule of this take-grant model can be specified as follows:

$take - rule$

$\Delta tg - model$
$Invoker\#, From?, To? : NAME$
$r? : RIGHT$

$Invoker\# \in Subjects$
$From? \in Entities$
$To? \in Entities$
$'t' \in AM(Invoker\#, From?)$
$r? \in AM(From?, To?)$
$AM' = AM \oplus$
$\quad ((Invoker\#, To?) \mapsto (AM(Invoker\#, To?) \cup \{r?\}))$

The first line in the declaration part includes the declarations and conditions of the schema *tg-model* described earlier (delta indicates a change in the state and declares the unprimed and the primed versions of the variables). The second line declares the variable set by the system, Invoker#, to be of type NAME and also two input variables (ending in question marks) of type NAME. The third line declares another input variable of type RIGHT.

The meaning of each line in the body part of the schema is described below:

- This operation can be invoked only by an entity belonging to Subjects.
- The name from which a right is obtained must belong to Entities.
- The name to which the right refers must belong to Entities.

- A take right needs to be present in the access matrix between the entity that invokes this schema and the entity from which the right is being taken.
- The requested right must be possessed by the entity from which it is being taken.
- After the operation the access matrix is updated so that the entity that invoked this operation has the appropriate right to the requested entity (the circle with a plus sign in it is the function override operator).

The following schema describes the grant-rule.

$$
\begin{array}{|l}
\hline
grant - rule \\
\Delta tg - model \\
Invoker\#, To?, For? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Subjects \\
For? \in Entities \\
To? \in Entities \\
'g' \in AM(Invoker\#, To?) \\
r? \in AM(Invoker\#, For?) \\
AM' = AM \oplus ((To?, For?) \mapsto (AM(To?, For?) \cup \{r?\})) \\
\hline
\end{array}
$$

This schema similar to the take-rule schema except for the following two aspects.

- A grant right needs to be present for it to be invoked.
- The effect on the state variables is different to reflect the grant rule of the take-grant model.

The create rule is:

$$
\begin{array}{|l}
\hline
create - rule \\
\Delta tg - model \\
Invoker\#, New? : NAME \\
r? : \mathbb{P}\, RIGHT \\
\hline
Invoker\# \in Subjects \\
New? \notin Entities \\
(Subjects' = Subjects \cup \{New?\}) \\
\quad \lor (Objects' = Objects \cup \{New?\}) \\
\forall\, x, y \bullet (((x \in Entities) \land (y \in Entities) \land \\
\qquad (x \neq Invoker\#) \land (y \neq New?)) \\
\qquad \Rightarrow (AM'(x, y) = AM(x, y))) \\
\forall\, x \bullet (((x \in Entities) \land\ (x \neq Invoker\#)) \\
\qquad \Rightarrow (AM'(x, New?) = \phi)) \\
AM'(Invoker\#, New?) = r? \\
\hline
\end{array}
$$

The declaration part of this schema is similar to the earlier schemas. The meaning of the statements in the body part is as shown below.

- An entity belonging to Subjects can invoke this operation.
- The name of the new entity needs to be unique.
- The newly created entity can be either a subject or an object.
- The set of entity names is updated with the name of the newly created entity.
- The rows and columns of AM that do not correspond to *Invoker#* or *New?* are unchanged.
- All the entries in the column corresponding to the newly created entity, except that for the entity that invokes this operation are empty.
- The creator of the new entity gets the rights that it requested.

The initial state is:

$$\begin{array}{|l} \hline Init \\ \Delta mtg - model \\ \hline \end{array}$$

The initial state of the system specifies that any set of values to the state variables which satisfy the invariants specified in the tg-model schema constitutes a valid initial state.

4.3.3 Specification of Protection Features of Unix File System

In order to demonstrate the ability of Z to specify the protection mechanisms of large systems we used it to describe a subset of the protection features of the Unix file system. This specification is given in Appendix C. This exercise demonstrated that Z is indeed suitable for specifying the protection features of real systems. In this process we discovered that the natural language description of the semantics of certain features of the file system (access bits of the directories) were not precise and in fact differ between different implementations.

4.4 Specification of Access Control Policies

Security policies usually consist of both access control requirements and information flow requirements. In order to completely specify a security policy we must be able to specify both these requirements. In Chapter 3 we developed a temporal logic based framework for specifying information flow requirements of security policies. In this section we will show that temporal logic can be used to specify access control policies as well.

Access control requirements are expressed as properties that must be satisfied by a set of states reachable from the initial state. These properties are specified as conditions on the values of the state variables in the description of the system. Hence these requirements can be expressed as temporal properties of the sequence of state transitions that can occur in the system. The temporal logic framework developed in Chapter 3 can be used to specify

these access control policies where the primitive propositions are expressions that describe a state in Z.

For example, the requirement that entity A cannot obtain a right to read an entity B in the take-grant model specified in Section 4.3.2 can be expressed as

$$\mathbf{A}\Box\ (\text{'}r\text{'} \notin AM(A, B))$$

That is, every state reachable from the initial state should be such that AM(A, B) does not contain *'r'* right.

Access control policies usually consist of:

- Mandatory access requirements and
- Discretionary access requirements.

Mandatory requirements, as the name suggests, must be true for each and every state of the system, whereas discretionary requirements need only be true for some specified subset of system states. A mandatory access requirement is usually expressed as $(\mathbf{A}\Box\ (P))$ where P is some property of a state. For example, a statement such as *'there must be no read from an entity of lower level to an entity of higher level'* in Bell-LaPadula model is a mandatory requirement. This requirement, usually referred to as the *simple-security property,* can be expressed in temporal logic as

$$\mathbf{A}\Box\ (\ (Class(A) \leq Class(B)) \Rightarrow (\text{'}r\text{'} \notin AM(A, B))\ )$$

where *Class*, *AM* and *'r'* are described by the state description of the system.

A discretionary access requirement is usually expressed as $(\mathbf{E}\Diamond\ (P))$ where P is some property. For example, a statement such as *'it should be possible for an entity A to acquire a read access to an entity B'* is a discretionary requirement that can be expressed as

$$\mathbf{E}\Diamond\ (\text{'}r\text{'} \in AM(A, B))$$

where *AM* and *'r'* are in the state description of the system.

4.5 Trace Specifications

The Z specification of a system does not include any notion of control flow. The schemas in the specification only enumerate all possible conditions under which the operations can take place and specify their effect on the state of the system. In order to reason about the properties of a specification, some compact way of referring to the set of traces that can be generated by a system needs to be devised. In this section we develop such a framework.

Definition: Given a system whose operations are described by the schemas Op_1, Op_2,..., Op_n, a *trace formula* is defined as follows:

- Any of the operations Op_i of the given system is a *trace formula.*
- If *I* is a state expression that does not contain any primed variables and *X* is a *trace formula* then so is $(I \wedge X)$.
- If $X_1, X_2, ..., X_m$ are *trace formulas* then so are
  - **C**($X_1, X_2, ..., X_m$), and
  - **D**($X_1, X_2, ..., X_m$).

where **C** and **D** are trace operators whose semantics will be given later in this section.

The importance of trace formula is that it can compactly represent the set of all traces that are of interest in a given system. In order to give the semantics of trace formulas, we first need some definitions on sequences.

Definition: Given a trace t which is a sequence of states of the form ($s_0, s_1, ..., s_m$)

- *first*(t) is the state s_0.

- *rest*(t) is the sequence $(s_1, ..., s_m)$
- *concat*(t, t') is the sequence $(s_0, s_1, ..., s_m, s'_0, s'_1, ..., s'_n)$ where t' is another trace of the form $(s'_0, s'_1, ..., s'_n)$.
- *last*(t) is the state s_m.

The set of traces generated by a trace expression can be characterized as shown below.

Definition: The set of traces generated by a trace expression X of a system S whose specification is (s, Init, $\{Op_1, Op_2, ..., Op_n\}$) is denoted as T_X and is

- If X is of the form Op_i then T_X contains a trace of the form (s0, s1) where s0 and s1 are states in the system S and s0 satisfies the preconditions of Op_i and s1 satisfies the post-conditions of Op_i.
- If X is of the form $(I \wedge Y)$ where I is a pre-condition then T_X is such that
  - every trace of T_X is in T_Y, and
  - for every trace t in T_X, I is true in *first*(t).
- If X is of the form **C**$(X_1, X_2, ..., X_m)$ then every trace t in T_X consists of sequences $t_1, t_2, ..., t_p$ such that
  - t is equal to *concat*$(t_1,$ *rest*$(t_2), ...,$ *rest*$(t_p))$,
  - for all i, $1 \le i < p$, *last*$(t_i) =$ *first*(t_{i+1}), and
  - for all i, $1 \le i \le p$, t_i belongs to one of T_{X_j} for $1 \le j \le m$.
- If X is of the form **D**$(X_1, X_2, ..., X_m)$ then every trace t in T_X consists of sequences $t_1, t_2, ..., t_m$ such that

- t is equal to *concat*(t_1, *rest*(t_2), ..., *rest*(t_m)),
- for all i, $1 \leq i < m$, *last*(t_i) = *first*(t_{i+1}), and
- for all i, $1 \leq i \leq m$, t_i belongs to T_{X_i}.

Note that the **C** is a closure operator and **D** is a sequencing operator. Any trace that is generated by the **C** operator with some parameters is constructed by concatenating a set of sub-traces where each sub-trace is generated by any of its parameters. Similarly, any trace generated by the **D** operator with a sequence of parameters is constructed by concatenating the ordered sequence of sub-traces where each sub-trace is generated by its corresponding parameter.

The above definition inductively defines the set of all traces generated by a trace expression. The trace specification operators **C** and **D** can be used to concisely specify any set of traces that are of interest in security policies. For instance, the set of *all* traces that can be generated by a system whose initial state is described by the schema *Init* and whose operations are described by the schemas Op_1, Op_2,..., Op_n is represented by the trace expression

$$Init \wedge \mathbf{C}\,(Op_1, Op_2, \ldots, Op_n)$$

Example 4.1: The set of all traces generated by the take-grant model specified in Section 4.3.2 is as follows:

$$Init \wedge \mathbf{C}\,(take-rule, grant-rule, create-rule)$$

The above expression denotes the set of all traces that can be generated by applying any of the *take-rule*, *grant-rule* and *create-rule* to the initial state *Init*.

Example 4.2: The set of all traces generated by the take-grant model where the take-rule is applied at least once can be represented as

$$Init \wedge \mathbf{D}\left(\begin{array}{c} \mathbf{C}\,(take-rule, grant-rule, create-rule)\,, \\ take-rule, \\ \mathbf{C}\,(take-rule, grant-rule, create-rule) \end{array}\right)^{\div}_{\div}$$

In this section, given a state machine specification of an information system, we developed a compact way of representing the required set of traces generated by it. Security policies, being the properties that the set of traces need to satisfy, can be verified against these trace specifications. A framework for carrying out this verification will be developed in next chapter.

4.6 Related Work

The framework that we presented in this chapter for specification of the traces has some similarities to the Temporal Logic of Actions (TLA) [LAM93] that is used for specifying and reasoning about concurrent systems. In this section we compare our framework with TLA.

4.6.1 Temporal Logic of Actions (TLA)

TLA is a logic for specifying and reasoning about concurrent programs. In this framework, both programs and properties are represented in a logic that is similar to the temporal logic, and verification is the process of showing that the program implies the property. A program is modeled as a state machine consisting of

- a set of states where a state is described by the values of the state variables and
- a set of actions where an action is a Boolean-valued expression formed from variables, primed variables and constant symbols. An action represents a relation between old states and new states where the un-primed variables refer to the old state and the primed variables refer to the new state.

In this model, an algorithm is considered to be the collection of all possible execution sequences that can be specified and reasoned about in temporal logic. The program descriptions are of the form

$$Init \wedge \Box [N]_f \wedge F$$

where

- *Init* is a description of the initial state,
- *N* is the operations in the program,
- *f* is a tuple of all variables that may not change, and
- *F* is a conjunction of the formulas specifying the fairness conditions.

The usual temporal operators □ (always) ◊ (eventually) are used to specify the properties of programs. In order to prove that a temporal property P is true about a program it is required to prove that

$$(Init \wedge \Box [N]_f \wedge F) \Rightarrow P$$

If this statement is valid then the program which represents the antecedent satisfies the property P. A set of rules were given in [LAM93] that can be used to prove statements of the above form depending on the structure of property P.

4.6.2 Comparison

TLA uses temporal logic to specify both programs and properties which is very appealing from the point of view of verification but it allows one to assume properties that may not be true about a system. For example, the fairness conditions of TLA are just assumptions and one needs to independently verify that these assumptions are indeed really enforced by the system. In our framework, the system specification cannot make any assumptions about the temporal behavior — every temporal property needs to be explicitly proved.

Careful comparison of TLA and our framework shows that the operator of TLA has the same semantics as the **C** operator of our framework but there is no equivalent for the **D** operator. This makes it impossible to specify and verify some properties like functionality requirements in TLA. This is not a drawback of TLA since it was intended only for concurrent programs where functionality is not an issue unlike security.

TLA is based on linear time temporal logic whereas our framework is based on the more expressive partial order temporal logic which allows us to specify and reason about a larger class of properties.

4.7 Conclusions

In this chapter, a model based specification approach to describe the protection mechanisms has been advocated. The specification language Z with minor extensions has been used to describe the take-grant system. We also developed a specification of the protection features of Unix file system that is more complete and comprehensive than any other such formal description using an access control model. We showed that the access control requirements can be expressed using the state description mechanisms of Z and temporal logic. We also developed a mechanism for a compact description of a set of traces that can be useful in the verification process.

Man associates ideas not according to logic or verifiable exactitude, but according to his pleasures and interests. It is for this reason that most truths are nothing but prejudices.

- Remy deGourmont

Chapter 5

Security Policy Verification

It was shown in earlier chapters that the formal specification language Z can be used for specifying the protection features of information systems and temporal logic can be used for specifying their information flow and access control security policies. In this chapter we propose a framework for verifying that the specification of an information system enforces a given security policy. We demonstrate this framework by verifying an access control policy and the MLS policy in a system similar to the take-grant model. We also investigate the labeling mechanisms used for enforcing different kinds of information flow policies.

5.1 Introduction

The semantics of the temporal statements, which make up a security policy specified in Chapter 3, provide a framework for verification. But unfortunately, the temporal operators in the formulas require that the reasoning be carried out over long sequences of state transitions. In the case of finite state machines, the verification process explores all possible sequences of state transitions to verify the security policies. This method, usually referred to as model checking, has been extensively used in network protocol verification and

hardware design verification among others. Security policies and their models are not finite state machines and so the model checking approach cannot be used. In this chapter we propose a set of rules that can be used inductively based on the structure of the temporal formulas to verify that a system specification enforces the security policy.

5.2 Verification of Security Policies

An important goal of modeling a system from the security point of view is to answer the question of safety — that is, whether a particular state is reachable from the initial state with the provided set of operations. In most access control models this safety question is undecidable in general but is decidable when the system is suitably restricted. For example,

- it has been shown [SNY81] that the question of safety in take-grant model can be decided in linear time.
- it was proved that the question of safety is undecidable for SPM [SAN88] in the general case but it is decidable when the instance of the model is acyclic (the create graph is acyclic) and attenuating (the newly created entities do not have more rights than their creators).

A similar kind of analysis methodology is desirable for the specification method that we proposed in the earlier chapter that can handle both access control and information flow requirements. But unfortunately the generality of the framework makes it very difficult to design an algorithm as in [SNY81] and [SAN88]. Instead we propose a set of rules that can be applied inductively to prove that a system enforces a security policy.

Given a system specified as $(s, I, \{Op_1, ..., Op_n\})$ we can use the following set of rules to carry out the verification process. A verification rule is specified as

$$\frac{\begin{array}{c} Cond_1 \\ \cdots \\ Cond_n \end{array}}{Theorem}$$

where $Cond_1$, ..., $Cond_n$ are the conditions that need to be proved in order to deduce that the *Theorem* is valid. In the following rules the statement P is assumed to be free of temporal operators.

- Rule 1:

$$\frac{(I \Rightarrow P) \vee (I \wedge Op) \Rightarrow P'}{(I \wedge Op) \Rightarrow \mathbf{A} \Diamond P} \qquad and \qquad \frac{(I \Rightarrow P) \vee (I \wedge Op) \Rightarrow P'}{(I \wedge Op) \Rightarrow \mathbf{E} \Diamond P}$$

  where Op is an operation from the set of operations $\{Op_1, Op_2, ..., Op_n\}$.

- Rule 2

$$\frac{(I \Rightarrow P) \wedge (I \wedge Op) \Rightarrow P'}{(I \wedge Op) \Rightarrow \mathbf{A} \Box P} \qquad and \qquad \frac{(I \Rightarrow P) \wedge (I \wedge Op) \Rightarrow P'}{(I \wedge Op) \Rightarrow \mathbf{E} \Box P}$$

  where Op is an operation from the set of operations $\{Op_1, Op_2, ..., Op_n\}$.

- Rule 3:

$$\frac{\begin{array}{c}(I \Rightarrow Inv) \\ ((Inv \wedge X_1) \Rightarrow \mathbf{A} \Box (Inv)) \wedge \ldots \wedge ((Inv \wedge X_n) \Rightarrow \mathbf{A} \Box (Inv)) \\ (Inv \Rightarrow P)\end{array}}{(I \wedge \mathbf{C}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A} \Box P}$$

In order to prove that P is always true in all the traces generated by the expression $(I \wedge \mathbf{C}(X_1, \ldots, X_n))$ find a condition *Inv* such that it

  - implies that P is true and
  - is invariant in all the trace expressions X_i.

- Rule 4:

$$\frac{\begin{array}{c}((I \wedge \neg P) \wedge \mathbf{C}((X_1 \wedge \neg P'), \ldots, (X_n \wedge \neg P'))) \Rightarrow \mathbf{A} \Box (Inv) \\ (Inv \wedge Op \Rightarrow P')\end{array}}{(I \wedge \mathbf{C}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A} \Diamond P}$$

In order to prove that P is true at least once in every trace of a system of traces described by $(I \wedge \mathbf{C}(X_1, \ldots, X_n))$, prove that every trace where P is not true in every state of that trace can be extended by executing an operation Op from the set of operations to reach a state where P is true.

- Rule 5:

$$\frac{\begin{array}{c} I \Rightarrow Inv \\ (Inv \wedge \mathbf{D}(x1, x2, \ldots, xm)) \Rightarrow \mathbf{E}\Box(Inv) \\ Inv \Rightarrow P \end{array}}{(I \wedge \mathbf{C}(X_1, \ldots, X_n)) \Rightarrow \mathbf{E}\Box P}$$

where each of {x1, x2, ..., xm} is from the set $\{X_1, X_2, \ldots, X_n\}$.

In order to prove that there is a trace in the set of traces described by $(I \wedge \mathbf{C}(X_1, \ldots, X_n))$ where property P holds for every state in that trace, find a sequence (x1, x2, ..., xm) such that *Inv* is true in every intermediate state. Then prove that *Inv* implies that P is true.

- Rule 6:

$$\frac{(I \wedge \mathbf{D}(x1, x2, \ldots, xm)) \Rightarrow \mathbf{E}\Diamond P}{(I \wedge \mathbf{C}(X_1, \ldots, X_n)) \Rightarrow \mathbf{E}\Diamond P}$$

where each of {x1, x2, ..., xm} are from the set $\{X_1, X_2, \ldots, X_n\}$.

In order to prove that there is a trace in the set of traces described by $(I \wedge \mathbf{C}(X_1, \ldots, X_n))$ where property P holds for some state in that trace, find a sequence (x1, x2, ..., xm) which when applied to the initial state will result in a state where P is true.

- Rule 7:

$$\frac{\begin{array}{c}(I \Rightarrow I_1) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{A}\Box I_1) \wedge \ldots \wedge ((I_n \wedge X_n) \Rightarrow \mathbf{A}\Box I_n) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{A}\Diamond I_2) \wedge \ldots \wedge ((I_{n-1} \wedge X_{n-1}) \Rightarrow \mathbf{A}\Diamond I_n) \\ (I_1 \Rightarrow P) \wedge \ldots \wedge (I_n \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A}\Box P}$$

where $\{I_1, I_2, ..., I_n\}$ are formulas that do not contain any temporal operators.

In order to prove that every state in every trace in the set of traces described by $(I \wedge \mathbf{D}(X_1, \ldots, X_n))$ has the property P, find a set of intermediate conditions $(I_1, I_2, ..., I_n)$ such that each of I_i is invariant when the trace X_i is applied to a state described by I_{i-1}. Also prove that every I_i implies P.

When $(I_1 = I_2 = ... = I_n = Inv)$, the above rule reduces to

$$\frac{\begin{array}{c}(I \Rightarrow Inv) \\ ((Inv \wedge X_1) \Rightarrow \mathbf{A}\Box(Inv)) \wedge \ldots \wedge ((Inv \wedge X_n) \Rightarrow \mathbf{A}\Box(Inv)) \\ (Inv \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A}\Box P}$$

- Rule 8:

$$\frac{\begin{array}{c}(I \Rightarrow I_1) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{E}\Box(I_1)) \wedge \ldots \wedge ((I_n \wedge X_n) \Rightarrow \mathbf{E}\Box(I_n)) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{E}\Diamond I_2) \wedge \ldots \wedge ((I_{n-1} \wedge X_{n-1}) \Rightarrow \mathbf{E}\Diamond I_n) \\ (I_1 \Rightarrow P) \wedge \ldots \wedge (I_n \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{E}\Box P}$$

where $\{I_1, I_2, ..., I_n\}$ are formulas that do not contain any temporal operators.

In order to prove that every state in every trace in the set of traces described by $(I \wedge \mathbf{D}(X_1, \ldots, X_n))$ has the property P, find a set of intermediate conditions $(I_1, I_2, ..., I_n)$

such that there is a trace in the sequence of traces described by $(I_{i-1} \wedge X_i)$ such that each state in it satisfies I_i. Also prove that every I_i implies P.

When ($I_1 = I_2 = ... = I_n = Inv$), the above rule reduces to

$$\frac{\begin{array}{c}(I \Rightarrow Inv) \\ ((Inv \wedge X_1) \Rightarrow \mathbf{E}\Box (Inv)) \wedge \ldots \wedge ((Inv \wedge X_n) \Rightarrow \mathbf{E}\Box (Inv)) \\ (Inv \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{E}\Box P}$$

- Rule 9:

$$\frac{\begin{array}{c}(I \Rightarrow I_1) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{E} \Diamond (I_2)) \wedge \ldots \wedge ((I_n \wedge X_n) \Rightarrow \mathbf{E} \Diamond (I_{n+1})) \\ (I_1 \Rightarrow P) \vee \ldots \vee (I_{n+1} \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{E} \Diamond P}$$

where $\{I_1, I_2, ..., I_n\}$ are formulas that do not contain any temporal operators.

In order to prove that some state in some trace in the set of traces described by $(I \wedge \mathbf{D}(X_1, \ldots, X_n))$ has the property P, find a set of intermediate conditions ($I_1, I_2, ..., I_n$) such that there is a trace in the sequence of traces described by $(I_i \wedge X_i)$ where the final state satisfies I_{i+1}. Also prove that any of I_i implies that P is true.

- Rule 10:

$$\frac{\begin{array}{c}(I \Rightarrow I_1) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{A} \Diamond (I_2)) \wedge \ldots \wedge ((I_n \wedge X_n) \Rightarrow \mathbf{A} \Diamond (I_{n+1})) \\ (I_1 \Rightarrow P) \vee \ldots \vee (I_{n+1} \Rightarrow P)\end{array}}{(I \wedge \mathbf{D}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A} \Diamond P}$$

where $\{I_1, I_2, ..., I_n\}$ are formulas that do not contain any temporal operators.

In order to prove that some state in every trace in the set of traces described by $(I \wedge \mathbf{D}(X_1, \ldots, X_n))$ has the property P, find a set of intermediate conditions ($I_1, I_2, ..., I_n$)

such that every the final state in trace in the sequence of traces described by $(I_i \wedge X_i)$ satisfies I_{i+1}. Also prove that any of I_i implies that P is true.

The rules given above deal with a temporal statement of the form (**X** P) where **X** is a temporal operator and P does not contain any temporal operators. When P contains temporal operators, the verification process gets more complicated. But fortunately most of security policy statements are usually in a standard form which is

$$\mathbf{A}\Box\ (P1 \Rightarrow \mathbf{X}P2)$$

where **X** is a temporal operator (including the past operators) and P1, P2 are formulas that do not contain any temporal operators in a system which is described by $(I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n))$. In order to handle these cases the following rules can be used.

- Rule 11:

$$\frac{(I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)) \Rightarrow\ \mathbf{A}\Box\ (Inv) \qquad ((Inv \wedge P1) \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)) \Rightarrow \mathbf{X}P2}{(I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)) \Rightarrow\ \mathbf{A}\Box\ (P1 \Rightarrow \mathbf{X}P2)}$$

  where **X** is a temporal operator from the set $\{\mathbf{A}\Box\ , \mathbf{A}\Diamond\ , \mathbf{E}\Box\ , \mathbf{E}\Diamond\ \}$.

In order to prove that every trace in a set of traces described by $I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)$ is such that if P1 is true in a state of that trace then some temporal formula $\mathbf{X}P2$ will be true in that state if the system satisfies $\mathbf{X}P2$ starting from the state that satisfies P1.

- Rule 12 a:

$$\frac{(I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)) \Rightarrow \mathbf{A}\Box\ (Inv) \qquad (Inv \wedge P1) \Rightarrow P2 \qquad ((Inv \wedge Op_1 \wedge P1') \Rightarrow P2) \wedge \ldots \wedge (Inv \wedge Op_n \wedge P1') \Rightarrow P2}{(I \wedge \mathbf{C}\ (Op_1, \ldots, Op_n)) \Rightarrow \mathbf{A}\quad (P1 \Rightarrow \overline{\ }P2)}$$

In order to prove that every trace in a set of traces described by $I \wedge \mathbf{C}\,(Op_1, \ldots, Op_n)$ is such that every state leading to a state that satisfies P1 satisfies P2, prove that there is an invariant *Inv* which is satisfied by every state in the system such that

- if *Inv* satisfies P1 then it satisfies P2, and
- if an operation transforms a state satisfying *Inv* into a new state satisfying P1 then the original state also satisfies P2.

- Rule 12 b:

$$\frac{((I \wedge \neg P2) \wedge \mathbf{C}\,((Op_1 \wedge \neg P2'), \ldots, (Op_n \wedge P2'))) \Rightarrow \mathbf{A}\Box\,(\neg P1)}{(I \wedge \mathbf{C}\,(Op_1, \ldots, Op_n)) \Rightarrow \mathbf{A} \quad (P1 \Rightarrow \overline{\Diamond}\, P2)}$$

In order to prove that every trace in a set of traces described by $(I \wedge \mathbf{C}\,(Op_1, \ldots, Op_n))$ is such that P2 is true in some state before P1 is true in a subsequent state, prove that in any trace where P2 is not true in any of its states cannot have P1 to be true in any of its states.

We have specified a set of rules that can be applied inductively to prove that the set of traces generated by a given trace expression possess certain temporal properties. In the subsequent sections we will illustrate the use of the framework developed in this section to prove security properties.

It must be noted that the set of rules is not complete — it means that not all statements that are true can be proved using the above set of inference rules. This is not a major drawback since all the security policies that we have dealt with so far can be handled by the above rules.

5.3 Verification of Access Control Policies

To illustrate the above framework we first use it to prove the conditions developed for the take-grant model [SNY81] for the safety question i.e., given any initial state, whether an entity A can get *x* right to another entity B — usually referred to as *can-share*(A, B, x). It

was proposed [SNY81] that *can-share*(A, B, x) is true only if *connected*(AM, A, B, x) is true where the predicate *connected* is defined as shown below.

Definition: The predicate *connected*(AM, A, B, x) is true if there are subjects $s_1, s_2, ..., s_n$ in AM such that (the path description notation is given in Section 1.1.1.1)

- either ($A = s_1$) or there is an object only path from s_1 to A of the form $\overrightarrow{t^*}\,\overrightarrow{g}$ in AM.
- there is an object only path from s_n to B of the form $\overrightarrow{t^*}\,\overrightarrow{x}$ in AM.
- each (s_i, s_{i+1}) is such that
  - there is either $\overrightarrow{t}$ or $\overrightarrow{g}$ or $\overleftarrow{t}$ or $\overleftarrow{g}$ between them in AM or
  - there is an object only path between them in AM of the form $\overrightarrow{t^*}$ or $\overleftarrow{t^*}$ or $\overrightarrow{t^*}\,\overrightarrow{g}\,\overleftarrow{t^*}$ or $\overrightarrow{t^*}\,\overleftarrow{g}\,\overleftarrow{t^*}$.

The above statement is a reformulation of the conditions given in [SNY81] where only an informal argument for the proof of the above statement was given. This theorem can be formally proved in the verification framework that we proposed earlier in this chapter. The predicate *can-share*(A, B, x) can be represented as $\mathbf{E} \Diamond (x \in AM(X, Y))$ in temporal logic.

Theorem: $\mathbf{E} \Diamond (x \in AM(X, Y))$ *iff connected(AM, A, B, x) is true in the initial state Init in a system whose set of traces is described by the trace expression*

$$Init \wedge \mathbf{C}(take-rule, grant-rule, create-rule)$$

where Init, take-rule, grant-rule and create-rule are the schemas defined in Section 4.3.2

Proof:

If Part:

It is easy to see that there is a sequence of operations starting from the initial state *Init* which depend on the subjects s_1, s_2,..., s_n in the definition of the predicate *connected*(A, B, x) which makes the statement ($x \in$ AM(A, B)) true in a derived state.

Only If Part: Prove that

$$\neg connected\,(AM, A, B, x) \wedge \mathbf{C}\,(take-rule, grant-rule, create-rule) \Rightarrow \neg\,(\mathbf{E} \Diamond\ (x \in AM\,(A, B))\,)$$

By distributing the negation over the existential path quantifier and the eventuality temporal operator we get

$$\neg connected\,(AM, A, B, x) \wedge \mathbf{C}\,(take-rule, grant-rule, create-rule) \Rightarrow \mathbf{A}\Box\ (\neg\,(x \in AM\,(A, B))\,)$$

The above statement can be proved by applying the rule 3 given in previous section which is

$$\frac{(I \Rightarrow Inv)\quad (\,(Inv \wedge X_1) \Rightarrow \mathbf{A}\Box\ (Inv)\,) \wedge \ldots \wedge (\,(Inv \wedge X_n) \Rightarrow \mathbf{A}\Box\ (Inv)\,)\quad (Inv \Rightarrow P)}{(I \wedge \mathbf{C}\,(X_1, \ldots, X_n)\,) \Rightarrow \mathbf{A}\Box P}$$

Here I is the initial state which is $\neg\,(connected\,(AM, A, B, x)\,)$ and the invariant *Inv* is selected to be predicate $\neg\,(connected\,(AM, A, B, x)\,)$.

Case 1: *Prove that* $(I \Rightarrow Inv)$.

This is trivially true since *I* and *Init* are same.

Case 2: Prove that $(Inv \wedge take-rule \Rightarrow \mathbf{A}\Box\ (Inv)\,)$

By applying rule 1 this reduces to $(Inv \wedge take-rule \Rightarrow Inv')$

Expanding *Inv* and *take-rule* from its schema, we get

$$\neg\,(connected\,(AM, A, B, x)\,) \tag{5.1}$$

$$'t' \in AM\,(\,(Invoker\#, From?)\,) \tag{5.2}$$

$$r? \in AM\,(\,(From?, To?)\,) \tag{5.3}$$

$$AM' = AM \oplus \begin{pmatrix} (Invoker\#, To?) \rightarrow \\ AM\,(\,(Invoker\#, To?)\,) \cup \{r?\} \end{pmatrix} \tag{5.4}$$

Now prove that *Inv'* is true i.e.,

$$\neg\,(connected\,(AM', A, B, x)\,)$$

Substituting the value of AM' from equation 5.4 we get

$$\neg\left(connected\left(AM \oplus \begin{pmatrix} (Invoker\#, To?) \rightarrow \\ AM\,(\,(Invoker\#, To?)\,) \cup \{r?\} \end{pmatrix}, A, B, x \right)\right)$$

Here it should be noted that the entities Invoker# and To? are already connected by appropriate path which is clear from the equations 5.2 and 5.3. Hence the above equation can be reduced to

$$\neg\,(connected\,(AM, A, B, x)\,)$$

This is true from the equation 5.1.

Hence *case 2* is proved.

Case 3: Prove that $(Inv \wedge grant - rule \Rightarrow \mathbf{A}\Box\,(Inv)\,)$

Similar to the proof in case 2.

Case 4: Prove that $(Inv \wedge create - rule \Rightarrow \mathbf{A}\Box\,(Inv)\,)$

Similar to the proof in case 2.

Case 5: Prove that $\neg\,(connected\,(AM, A, B, x)\,) \Rightarrow \neg\,(x \in AM\,(A, B)\,)$

From the definition of connected predicate it can be seen that if $(x \in AM(A, B))$ then the sequence of subjects can be selected to be the empty sequence to satisfy all the required conditions. Hence $(x \in AM(A, B)) \Rightarrow (connected(AM, A, B, x))$ from which we can deduce the above statement.

Therefore we can conclude that $(\mathbf{A}\Box (\neg (x \in AM(A, B))))$.

Hence it can be concluded that the statement of the theorem is true.

This proof shows that the rules of inference given in Section 5.2 can be used to prove that the system specification correctly enforces the access control policies.

5.4 Verification of Information Flow Policies

In this section we develop a formal specification of the protection features of an information system and show that it satisfies MLS policy which is an information flow policy. The system that we consider is similar to the take-grant model [SNY81] with a few changes so that the MLS policy can be enforced.

5.4.1 The Security Policy Specification

The MLS policy to be enforced by the system is (C, Op, P, A) where

- The set of information classes C is {l1, l2,..., ln}.
- The set Op is {=, ≤} where
  - ≤ which imposes a partial order on these information classes and
  - = has the usual meaning.
- The set P is (X → Y) for every information class X and Y.
- The policy statement is

$$\mathbf{A}\Box (\forall X, Y \times ((X \rightarrow Y) \Rightarrow (X \leq Y)))$$

5.4.2 Specification of the Information system

The system is specified as (S, π) where:

- S is the system description which among other things consists of
  - a state variable *Entities* that denotes the set of entities in the system,
  - a state variable η that gives the class assigned to each entity.
  - a state variable AM which gives the rights that each entity has to other entities. In a manner similar to the traditional take-grant model, the entities and their rights are represented as nodes and (labeled) arcs of a graph

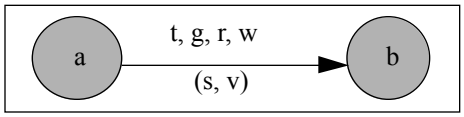


Figure 5.1 : The state of the system in mtg-model

  where t (take), g (grant), r (read), w (write) denote the function of this right and (s, r) is a label that denotes how this right can be used — s is an information class, v is a set of information classes. The label (s, v) is used to control the application of the state changing commands as shown later.

  - the state changing commands — *mtake-rule*, *mgrant-rule* and *mcreate rule*; these commands are defined in the following sections. In addition there are the obvious read and write-rules; if entity *a* has an arc to entity *b* labeled with an *r*, it can read the contents of *b* and if entity *a* has an arc to entity *b* labeled with a *w*, it can write into *b*.

- π gives the truth value of the relations ($l1 \rightarrow l2$) where $l1, l2 \in C$; ($l1 \rightarrow l2$) is true in a state if there exist two entities *a* and *b* where ($\eta(a) = l1$) and ($\eta(b) = l2$) and there is a read/take right from *b* to *a* or a write/grant right from *a* to *b*. It should be noted that this definition of information flow may be different depending on the situation and the system specifier has the prerogative to choose an interpretation

appropriate to the application at hand. Formally the information flow relation can be stated as:

$$\left(\begin{array}{l} ((a, b, r) \in AM(y, x)) \vee ((a, b, t) \in AM(y, x)) \vee \\ ((a, b, w) \in AM(x, y)) \vee ((a, b, g) \in AM(x, y)) \end{array} \right) \Rightarrow (x \rightarrow y)$$

5.4.2.1 The State Description

The state of the system can be described in Z as follows:

$$\begin{array}{|l} \hline mtg - model \\ \hline Entities : \mathbb{P}\, NAME \\ Subjects : \mathbb{P}\, NAME \\ Objects : \mathbb{P}\, NAME \\ Class : \mathbb{P}\, NAME \\ \eta : NAME \nrightarrow NAME \\ AM : NAME \times NAME \nrightarrow \mathbb{P}(NAME \times \mathbb{P}\, NAME \times RIGHT) \\ \hline Subjects \cup Objects = Entities \\ Subjects \cap Objects = \phi \\ Class \cap Entities = \phi \\ \mathrm{dom}\, \eta = Entities \\ \mathrm{ran}\, \eta = Class \\ \forall x, y \bullet (((x, y) \in \mathrm{dom}\, AM) \\ \qquad \Rightarrow ((x \in Entities) \wedge (y \in Entities))) \\ \forall x, y, z \bullet (((x, y, z) \in \mathrm{ran}\, AM) \\ \qquad \Rightarrow ((x \in Class) \wedge (y \in \mathbb{P}\, Class))) \\ \hline \end{array}$$

The declaration part of the above schema introduces a set of state variables whose purpose is as described below.

- *Entities* is a set of names in the system.
- *Subjects* are the active entities that can change the state of the system through the transition rules.
- *Objects* are the passive entities in the system.
- *Class* is the set of information classes.
- η is a function that assigns an information class to each entity.

- *AM* is the data structure that records the protection state of the system.

The body of the schema specifies the restrictions on state variables. The explanation for each statement is given below.

- The set Entities is made up of the set of Subjects and Objects.
- A name cannot be a Subject and an Object at the same time.
- A name cannot be both an information class and an Entity.
- The class assignment function assigns an information class to only Entities.
- The class assignment function assigns only class names to an Entity.
- The rows and columns of access matrix are indexed by Entities.
- The contents of the access matrix is a three tuple where first is a class name, the second is a set of information classes and the third is a right.

5.4.2.2 The Transition Rules

The Take-Rule:

$$
\begin{array}{|l}
\hline
mtake - rule \\
\Delta mtg - model \\
Invoker\#, From?, To? : NAME \\
r? : (NAME \times \mathbb{P}\, NAME \times RIGHT) \\
\hline
Invoker\# \in Subjects \\
From? \in Entities \\
To? \in Entities \\
\exists\, s1, v1, s2, v2, s, v, x \bullet \\
\quad (((s1, v1,' t') \in AM(Invoker\#, From?)) \land \\
\quad (r? = (s, v, x)) \land \\
\quad ((s2, v2, x) \in AM(From?, To?)) \land \\
\quad (s1 \in v2) \land \\
\quad (s = max(s1, s2)) \land \\
\quad (v \subseteq v2)) \\
(AM' = AM \ \oplus \\
\qquad ((Invoker\#, To?) \mapsto (AM(Invoker\#, To?) \cup \{r?\}))) \\
\hline
\end{array}
$$

This schema states the conditions for an entity to copy a right from another entity. It states that only a Subject can copy a right, the right to be copied should already exist at the entity from which it is being copied. The changes to the state of the system can be pictorially represented as shown below (dashed part shows the effect of transition):

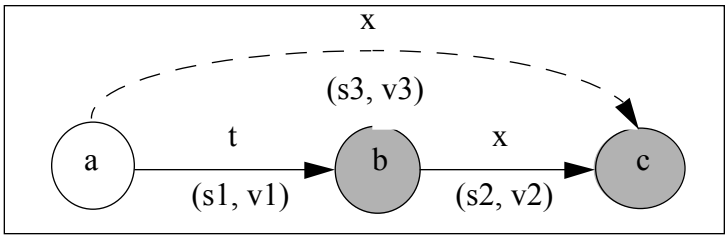


Figure 5.2 : The mtake-rule of mtg-model

Here (s1 ∈ v2) and (s3, v3) is such that

$$s3 = max(s1, s2) \qquad and \qquad v3 \subseteq v2$$

where

$$max(s1, s2) = \begin{pmatrix} s1 & if & s2 \le s1 \\ s2 & if & s1 < s2 \end{pmatrix}$$

The Grant-Rule:

$$\begin{array}{|l} mgrant - rule \\ \hline \Delta mtg - model \\ Invoker\#, To?, For? : NAME \\ r? : (NAME \times \mathbb{P}\, NAME \times RIGHT) \\ \hline Invoker\# \in Subjects \\ For? \in Entities \\ To? \in Entities \\ \exists\, s1, v1, s2, v2, s, v, x \bullet \\ \quad (((s1, v1,' g') \in AM(Invoker\#, To?)) \land \\ \quad (r? = (s, v, x)) \land \\ \quad ((s2, v2, x) \in AM(Invoker\#, For?)) \land \\ \quad (s1 \in v2) \land \\ \quad (s = max(s1, s2)) \land \\ \quad (v \subseteq v2)) \\ (AM' = AM \oplus \\ \qquad ((To?, For?) \mapsto (AM(To?, For?) \cup \{r?\}))) \\ \hline \end{array}$$

This schema states the conditions for an entity to copy a right that it has to another entity. It states that only a Subject can give away a right that it already has. The changes to the state of the system can be pictorially represented as shown below,

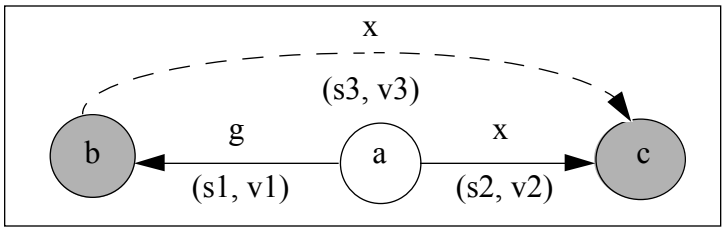


Figure 5.3 : The mgrant-rule of mtg-model

Here (s1 ∈ v2) and (s3, v3) is such that

$$s3 = max(s1, s2) \qquad and \qquad v3 \subseteq v2$$

where max is as defined before.

The Create Rule:

mcreate − rule

$$\Delta mtg - model$$
$$Invoker\#, New? : NAME$$
$$r? : \mathbb{P}(NAME \times \mathbb{P}\, NAME \times RIGHT)$$

$$Invoker\# \in Subjects$$
$$New? \notin Entities$$
$$Subjects' = Subjects \cup \{New?\}$$
$$\quad \lor\ Objects' = Objects \cup \{New?\}$$
$$Entities' = Entities \cup \{New?\}$$
$$\eta' = \eta \oplus (New? \mapsto \eta(Invoker\#))$$
$$\forall x, y \bullet (((x, y, t) \in r?) \Rightarrow ((x = \eta(Invoker\#)) \land (\forall z \bullet ((z \in y) \Rightarrow (x \leq z)))))$$
$$\forall x, y \bullet (((x, y, g) \in r?) \Rightarrow ((x = \eta(Invoker\#)) \land (\forall z \bullet ((z \in y) \Rightarrow (z \leq x)))))$$
$$\forall x, y \bullet (((x, y, r) \in r?) \Rightarrow ((x = \eta(Invoker\#)) \land (y = \phi)))$$
$$\forall x, y \bullet (((x, y, w) \in r?) \Rightarrow ((x = \eta(Invoker\#)) \land (y = \phi)))$$
$$AM' = AM \oplus ((Invoker\#, New?) \mapsto r?)$$

This schema states that a Subject can create an Entity that belongs to same information class and get any right to it. Pictorially this can be represented as shown below:

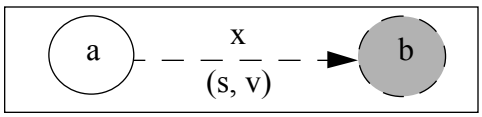


Figure 5.4 : The mcreate-rule of mtg-model

Here

$$s = \eta(a) \qquad and \qquad \left(v = \begin{array}{lll} \{ l \in C \mid (s \leq l) \} & if & x = t \\ \{ l \in C \mid (l \leq s) \} & if & x = g \end{array} \right)$$

5.4.2.3 The Initial State

Let the initial state, *Init* be as described below.

$$\begin{array}{|l} \textit{Init} \\ \hline \Delta mtg - model \\ \hline \forall x, y, a, b \bullet (((a, b, ' t') \in AM(x, y)) \\ \qquad \Rightarrow ((a = \eta(x)) \wedge (\eta(x) \geq \eta(y)) \wedge \\ \qquad\quad (\forall z \bullet ((z \in b) \Rightarrow (\eta(y) \leq z))))) \\ \forall x, y, a, b \bullet (((a, b, ' r') \in AM(x, y)) \\ \qquad \Rightarrow ((a = \eta(x) \wedge (\eta(x) \geq \eta(y)) \wedge \\ \qquad\quad (\forall z \bullet ((z \in b) \Rightarrow (\eta(y) \leq z))))) \\ \forall x, y, a, b \bullet (((a, b, ' g') \in AM(x, y)) \\ \qquad \Rightarrow ((a = \eta(y) \wedge (\eta(x) \leq \eta(y)) \wedge \\ \qquad\quad (\forall z \bullet ((z \in b) \Rightarrow (z \leq \eta(y)))))) \\ \forall x, y, a, b \bullet (((a, b, ' w') \in AM(x, y)) \\ \qquad \Rightarrow ((a = \eta(y)) \wedge (\eta(x) \leq \eta(y)) \wedge \\ \qquad\quad (\forall z \bullet ((z \in b) \Rightarrow (z \leq \eta(y)))))) \\ \hline \end{array}$$

This gives the set of conditions that characterize the set of initial states. It states that a take or a read arc can exist from an entity to an entity at the same or lower class and a grant or a write can exist from an entity to an entity at the same or higher level.

take or read arc:

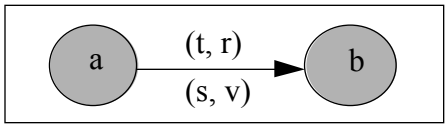


Figure 5.5 : The take or read arc of mtg-model to enforce MLS

Here

$$s = \eta(a) \qquad and \qquad (v = \{ l \in C | (\eta(b) \leq l) \})$$

Also, $\eta(a) \geq \eta(b)$.

grant or write arc:

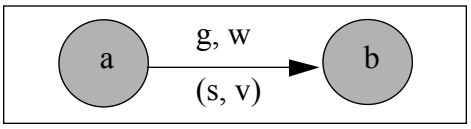


Figure 5.6 : The grant or write arc of mtg-model to enforce MLS

Here,

$$s = \eta(b) \qquad and \qquad (v = \{ l \in C | (l \leq \eta(b)) \})$$

Also, $\eta(b) \geq \eta(a)$.

5.4.3 The Verification

In the previous section we developed a formal specification of the system that enforces the MLS policy. In order to increase our confidence in the above specification, we need to *formally prove* that it correctly enforces the policy. In this section we develop such a proof.

Theorem: The system described as (mtg-model, Init, {mtake-rule, mtake-rule, mgrant-rule, mcreate-rule}) enforces the Multi-Level security policy i.e.,

$$(Init \wedge \mathbf{C}(mtake-rule, mgrant-rule, mcreate-rule))$$
$$\Rightarrow$$
$$\mathbf{A}\Box (\forall l1, l2 \bullet ((l1 \rightarrow l2) \Rightarrow (l1 \leq l2)))$$

Proof:

In order to prove the theorem, rule 3 of Section 5.2 can be used which is

$$\frac{\begin{array}{c}(Init \Rightarrow Inv) \\ ((Inv \wedge X_1) \Rightarrow \mathbf{A}\Box(Inv)) \wedge \ldots \wedge ((Inv \wedge X_n) \Rightarrow \mathbf{A}\Box(Inv)) \\ (Inv \Rightarrow P)\end{array}}{(Init \wedge \mathbf{C}(X_1, \ldots, X_n)) \Rightarrow \mathbf{A}\Box P}$$

Here select the condition *Inv* to be the statement shown below:

$$\forall x, y, a, b \bullet \left(\begin{array}{l} (((a, b, t) \in AM(x, y)) \Rightarrow ((\eta(x) \geq \eta(y)) \wedge (a = \eta(x)))) \wedge \\ (((a, b, r) \in AM(x, y)) \Rightarrow ((\eta(x) \geq \eta(y)) \wedge (a = \eta(x)))) \wedge \\ (((a, b, g) \in AM(x, y)) \Rightarrow ((\eta(x) \leq \eta(y)) \wedge (a = \eta(y)))) \wedge \\ (((a, b, w) \in AM(x, y)) \Rightarrow ((\eta(x) \leq \eta(y)) \wedge (a = \eta(y)))) \end{array} \right)$$

Case 1: Prove that $(Init \Rightarrow Inv)$

It is easy to see that the conditions in the body of the *Init* schema imply the conditions in *Inv*

Case 2: Prove that $((Inv \wedge mtake - rule) \Rightarrow \mathbf{A}\Box(Inv))$

By rule 1 this reduces to $((Inv \wedge mtake - rule) \Rightarrow Inv')$

Expanding *mtake-rule* gives the following expressions:

$$((s1, v1, 't') \in AM(Invoker\#, From?)) \tag{5.1}$$

$$(r? = (s, v, x)) \tag{5.2}$$

$$(s1 \in v2) \tag{5.3}$$

$$((s2, v2, x) \in AM(From?, To?)) \tag{5.4}$$

$$(s = max(s1, s2)) \tag{5.5}$$

$$(v \subseteq v2) \tag{5.6}$$

$$AM' = AM \oplus \left(\begin{array}{c} (Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\}) \end{array} \right) \tag{5.7}$$

Now consider *Inv'* that is as shown below

$$\forall x, y, a, b \bullet \left(\begin{array}{l} (((a, b, 't') \in AM'(x, y)) \Rightarrow ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x)))) \wedge \\ (((a, b, 'r') \in AM'(x, y)) \Rightarrow ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x)))) \wedge \\ (((a, b, 'g') \in AM'(x, y)) \Rightarrow ((\eta(x) \leq \eta(y)) \wedge (a = \eta(y)))) \wedge \\ (((a, b, 'w') \in AM'(x, y)) \Rightarrow ((\eta(x) \leq \eta(y)) \wedge (a = \eta(y)))) \end{array} \right)$$

We can prove each of the conjuncts individually.

case 21: Prove that

$$\forall x, y, a, b \bullet (((((a, b, 't') \in AM'(x, y)) \Rightarrow ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x))))))$$

Replacing the value of AM' from equation 5.7 transforms the above equation into:

$$\forall x, y, a, b \bullet \left(\begin{array}{c} \left((a, b, 't') \in \left(AM \oplus \left(\begin{array}{c} (Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\}) \end{array} \right) \right)(x, y) \right) \\ \Rightarrow \\ ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x))) \end{array} \right)$$

From the definition of function override operator the above expression can be transformed into:

$$\forall x, y, a, b \bullet \left(\begin{array}{c} \left(\begin{array}{c} (((a, b, 't') \in AM(x, y)) \wedge ((x \neq Invoker\#) \vee (y \neq To?))) \vee \\ \left((a, b, 't') \in \left(\begin{array}{c} (Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\}) \end{array} \right)(x, y) \right) \end{array} \right) \\ \Rightarrow \\ ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x))) \end{array} \right)$$

The above expression can again be split into two parts.

case 211:

$$\forall x, y, a, b \bullet \left(\begin{array}{c} (((a, b, 't') \in AM(x, y)) \wedge ((x \neq Invoker\#) \vee (y \neq To?))) \\ \Rightarrow \\ ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x))) \end{array} \right)$$

This is true since the invariant *Inv* implies it.

case 212:

$$\forall x, y, a, b \bullet \left(\begin{array}{c} (a, b, 't') \in \left(\begin{array}{c} (Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\}) \end{array} \right)(x, y) \\ \Rightarrow \\ ((\eta(y) \leq \eta(x)) \wedge (a = \eta(x))) \end{array} \right)$$

From the definition of mapping construction the above expression can be transformed into:

$$\forall a, b \bullet \left(\begin{array}{c} (a, b, 't') \in ((AM(Invoker\#, To?) \cup \{r?\})) \\ \Rightarrow \\ ((\eta(To?) \leq \eta(Invoker\#)) \wedge (a = \eta(Invoker\#))) \end{array} \right)$$

Again by algebraic simplification, the above statement can be transformed into:

$$\forall a, b \bullet \left(\begin{array}{c} ((a, b, 't') \in AM(Invoker\#, To?)) \vee ((a, b, t) = r?) \\ \Rightarrow \\ ((\eta(To?) \leq \eta(Invoker\#)) \wedge (a = \eta(Invoker\#))) \end{array} \right)$$

Again this can be split into two parts.

case 2121: Prove that

$$\forall a, b \bullet \left(\begin{array}{c} ((a, b, 't') \in AM(Invoker\#, To?)) \\ \Rightarrow \\ ((\eta(To?) \leq \eta(Invoker\#)) \wedge (a = \eta(Invoker\#))) \end{array} \right)$$

This follows from *Inv.*

case 2122: Prove that

$$\forall a, b \bullet \left(((a, b, 't') = r?) \Rightarrow \left(\begin{array}{c} (\eta(To?) \leq \eta(Invoker\#)) \wedge \\ (a = \eta(Invoker\#)) \end{array} \right) \right)$$

skolemizing the above expression with (a = s), (b = v) results in:

$$((s, v, 't') = r?) \Rightarrow ((\eta(To?) \leq \eta(Invoker\#)) \wedge (s = \eta(Invoker\#)))$$

From equation 5.4, with (x = t), we get:

$$((\eta(To?) \leq \eta(Invoker\#)) \wedge (s = \eta(Invoker\#)))$$

Now from 5.4 and *Inv* we can conclude that $(\eta(From?) \leq \eta(Invoker\#))$ and from 5.7 and *Inv* we can conclude that $(\eta(To?) \leq \eta(From?))$ which in turn from the transitivity of ≤ means that $(\eta(To?) \leq \eta(Invoker\#))$.

Also from 5.1 and *Inv*, $(s1 = \eta(Invoker\#))$ and also from 5.4 and *Inv*, $(s2 = \eta(From?))$ which means that $(s = max(s1, s2))$ and is equal to $\eta(Invoker\#)$ which completes proof of *case 122*.

This completes the proof of the first conjunct of *Inv'*.

All the other conjuncts from *Inv'* can be proved similarly.

Case 3: Prove that $((Inv \wedge mgrant - rule) \Rightarrow \mathbf{A}\Box(Inv))$

The proof is similar to the proof in case 2.

Case 4: Prove that $((Inv \wedge mcreate - rule) \Rightarrow \mathbf{A}\Box(Inv))$

Again the proof is similar to the proof in case 2.

Case 5: Prove that $(Inv \Rightarrow (\forall l1, l2 \bullet ((l1 \rightarrow l2) \Rightarrow (l1 \leq l2))))$

This follows directly from the definition of the interpretation function π and the definition of the information flow relation.

Hence we can conclude that $(\mathbf{A}\Box(\forall l1, l2 \bullet ((l1 \rightarrow l2) \Rightarrow (l1 \leq l2))))$.

In this section we showed that a formal description of an information system enforces the MLS policy. It should be noted that the proof process is fairly mechanical in nature.

5.5 Label Based Enforcement of Information Flow Policies

In previous sections we developed a framework for proving that a specification enforces a security policy. But this proof process does not give a constructive way of developing the specifications that enforce a given policy. In this section we show how information flow policies can be enforced by explicitly labeling the entities in the system and controlling the validity of operations that cause information flows based on these labels, among other things. We use the taxonomy of the information flow policies that was first investigated in [FOL89b] to develop these labeling mechanisms.

Information flow security policies are specified as restrictions on the information flows that can occur between different information classes. These policies can be enforced by a lattice model of information flow which is defined below.

Definition: A Lattice model of information flow [DEN76]

- consists of a set of labels with an ordering relation that forms a lattice (hence the name) and
- information is allowed to flow from an entity labeled A to an entity labeled B if the label A is less than label B in the ordering relation of the lattice.

All the entities in a system are assigned labels from the lattice such that if information is allowed to flow from entity a to entity b, then the label on the entity a is less than the label on entity b in the lattice ordering relation.

In some cases, like MLS policy, the information classes themselves form a lattice structure and therefore can be directly used to label the entities. But in many cases, such a

lattice structure has to be derived from the statements of the security policy. For example, consider a security policy defined as follows:

- The information classes are {A, B, C}.
- The policy statements are {**A** (¬ (C → A)) and **A** (¬ (C → B)) and **A** (¬ (A → B)) and **A** (¬ (B → A)). That is, information cannot flow from C to A, from C to B, from A to B and from B to A.

The above policy can be enforced by the lattice shown in the following figure where the ordering relation for the lattice is the subset relation.

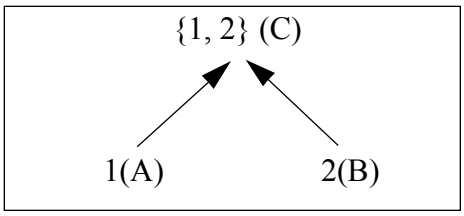


Figure 5.7 : Lattice to enforce an Information flow Policy

In a lattice model, the number of labels is finite and is decided at the system design time. But the assignment of labels to the entities can be [DEN83]

- static — which means that the labels associated with the entities do not change as a result of state changes in the system, or
- dynamic — which means that the labels can change as a result of state changes in the system.

It should be noted that the static labeling policies are more restrictive — they may have to reduce the functionality of the system in some cases to enforce the security policy whereas dynamic labeling mechanisms provide more flexibility for enforcing the policy.

5.5.1 Static Labeling Mechanism

In static labeling mechanism, the entities in the system are labeled by the information classes and the state changing operations use these labels for affecting changes to the states. An important thing to be noticed is that, in these models, the state changing operations do not modify the labels associated with the entities (hence the name). Since the state changing operations do not involve any modifications to the labels of entities, they are more efficient than the corresponding operations in a dynamic labeling mechanism.

A drawback of the static labeling mechanism is that the enforcement of the security policy might restrict the functionality of the system more than actually required. For example, consider the Chinese Wall policy that has a conflict of interest set {Bank1, Bank2} and another set {Consultant}. The information flow restrictions of this policy state that information can flow from Bank1 to the Consultant or from Bank2 to the Consultant but not both. A system that enforces this policy using a static labeling mechanism will have to make the choice a priori of the set of allowed information flows. This choice is made by associating same label to one of the banks and the consultant (to facilitate information flow between them) and a different unrelated label to the other bank (to preclude the information flow). Although such a labeling does enforce the security policy, it unnecessarily restricts the functionality of the system.

5.5.2 Dynamic Labeling Mechanisms

In dynamic labeling mechanisms, the security labels associated with the entities in the system can change as a result of state changing operations that take place in it. These mechanisms can enforce some security policies with more functionality than the static labeling mechanisms. In the following sections different types of information flow policies are identified and a dynamic labeling mechanism for their enforcement is presented.

5.5.2.1 Separation Policy

A security policy whose information flow requirements are of the form

$$\mathbf{A}\Box\ (\neg\,(A \rightarrow B)\,)$$

where A and B are information classes is called a separation policy.

The fact that information flow is transitive makes these kind of policies difficult to enforce. For example, consider a policy defined as follows:

- The information classes are {A, B, C}.
- The policy statement is **A** (¬ (A → C)). This states that information cannot flow from A to C.

A system that enforces the above policy, should ideally allow all the information flows which are not precluded by the policy. It means that the flows (A → B) and (B → C) should be allowed to occur from the initial state — but since information flow is transitive, the flow (B → C) should be disallowed once the flow (A → B) occurs.

A general method for enforcing such information flow policies by dynamic labeling can be defined as follows:

- Construct an access matrix AM: Class → **P** Class with the property

$$\forall A, B \bullet\ (\neg\,(A \rightarrow B)\ \Rightarrow\ (A \in AM\,(B)\,)\,)$$

- Every entity is assigned a label, referred to as Hwm (High-Water mark), of type (**P** Class) and it is initialized to be {A}, where A is the information class associated with the entity.
- Information is allowed to flow from an entity a of class A to another entity b of class B, if

$$(Hwm\,(a)\ \cap AM\,(B)\,)\ =\ \varnothing$$

- If the information is allowed to flow from entity a to entity b then Hwm of b is

updated with the Hwm of a i.e.,

$$Hwm\,(b) \;=\; (Hwm\,(a) \cup Hwm\,(b))$$

Example 5.1: Consider a messaging system consisting of a set of Entities that can communicate with each other by sending messages. The state description of such a system is shown below:

$Msg - Model$

$$Entities : \mathbb{P}\, NAME$$
$$Class : \mathbb{P}\, NAME$$
$$Hwm : NAME \pfun \mathbb{P}\, NAME$$
$$\eta : NAME \pfun NAME$$
$$MsgArea : NAME \pfun (NAME \times \mathrm{seq}\, CHAR)$$
$$AM : NAME \pfun \mathbb{P}\, NAME$$

$$Entities \cap Class = \phi$$
$$Entities \cap Hwm = \phi$$
$$Hwm \cap Class = \phi$$
$$\mathrm{dom}\, \eta = Entities$$
$$\mathrm{dom}\, AM = Entities$$
$$\mathrm{dom}\, Hwm = Entities$$
$$\mathrm{dom}\, MsgArea = Entities$$
$$\mathrm{ran}\, \eta = Class$$
$$\mathrm{ran}\, AM = \mathbb{P}\, Class$$
$$\mathrm{ran}\, Hwm = \mathbb{P}\, Class$$
$$\forall x, y \bullet (((x, y) \in \mathrm{ran}\, MsgArea) \Rightarrow (x \in Entities))$$

Here Entities, Class and η have the usual meaning. The Hwm at any instant records the informations flows that have occurred in the past and AM records the information about the security policy. MsgArea contains the message received by an entity at any instant.

The single operation in this system is *Msg-Send* where an entity can send a message to another entity. This operation can be described in Z as follows:

$$
\begin{array}{|l}
\hline
Msg - Send \\
\Delta Msg - Model \\
Invoker\#, To? : NAME \\
Msg? : \text{seq } CHAR \\
\hline
Invoker\# \in Entities \\
To? \in Entities \\
Hwm(Invoker\#) \cap AM(To?) = \phi \\
Hwm' = Hwm \oplus \\
\qquad (To? \mapsto (Hwm(Invoker\#) \cup Hwm(To?)) \\
MsgArea' = MsgArea \oplus (To? \mapsto (Invoker\#, Msg?)) \\
\hline
\end{array}
$$

The above specification, allows all the accesses allowed by the security policy — but it is expensive since every operation that causes an information flow involves updating the Hwm label and a comparison to decide the validity of the operation. Moreover, the cost of maintaining the Hwm labels of the entities is quite high — exponential in the total number of classes in the system.

Under certain conditions, the separation policies can be enforced by a simple static labeling mechanism. Given the set of policy statements where every statement is of the form $(\mathbf{A}\Box\,(\neg\,(A \rightarrow B))\,)$ build a flow graph FG = (V, X) where

- V is the set of vertices where each vertex represents an information class, and
- X is the set of arcs and a directed arc from a node representing class A to a node representing class B if there is no policy statement of the form $(\mathbf{A}\Box\,(\neg\,(A \rightarrow B))\,)$.

If this flow graph is transitive then the security policy can be simply enforced by labeling the entities and using these labels to validate the operations that cause information flows. It should be noted that the updating of Hwm's as a result of information flow operations is not required. The reason for this is that if information can flow from A to B,

then the transitive nature of the graph implies that if information cannot flow from A to X then information will not be able to flow from B to X. This makes updating the HWM of B with the HWM of A unnecessary.

A flow graph of a security policy, which is not transitive, can be made transitive by deleting some arcs from it. It should be noted that this deletion of arcs from the flow-graph does not violate the security policy in any way, but it reduces the functionality of the system. Therefore, to maintain the functionality as high as possible, the number of arcs deleted from the flow-graph to make it transitive should be the minimum. It means that we need to find a graph G' = (V, X') where X' $\subseteq$ X such that G' is transitive. This problem has been shown to be NP-Complete [GJ79]. But since the total number of information classes is fairly small this is not a great concern in the system design.

5.5.2.2 Intransitive Policies

An information flow policy whose policy statements are of the form

$$\Box \mathbf{A} ((A \rightarrow B) \Rightarrow \neg (\Box \mathbf{A} (B \rightarrow C)))$$

where A, B and C are the information classes is called an *intransitive policy*. In these policies, if information flows from A to B then from that instant information flow from B to C cannot occur. These policies are special forms of separation policies specified in the previous section. These are called Intransitive policies since they specifically disallow indirect flow of information through transitive nature of information flow. A dynamic labeling model similar to the one in the previous section can be used to enforce these policies.

- Construct an access matrix AM: Class $\rightarrow$ **P** (Class $\times$ Class) with the property that for any class B, ((A, C) $\in$ AM(B)) means that there is a policy axiom of the form

$$\Box \mathbf{A} ((A \rightarrow B) \Rightarrow \neg (\Box \mathbf{A} (B \rightarrow C)))$$

- Every entity is assigned a label, referred to as Hwm (High-water mark), of type (**P** Class) and it is initialized to be {A}, where A is the information class associated

with the entity.

- Information is allowed to flow from an entity a of class A to another entity b of class B, if

$$\neg (\exists X \bullet ((X \in Hwm(a)) \Rightarrow ((X, B) \in AM(A))))$$

- If the information is allowed to flow from entity a to entity b then Hwm of b is updated with the Hwm of a i.e.,

$$Hwm(b) = (Hwm(a) \cup Hwm(b))$$

Example 5.2: The state description of the messaging system described in the earlier example that enforces a given Intransitive information flow policy is as shown below:

$$
\begin{array}{|l}
\hline
Msg - Model \\
\hline
Entities : \mathbb{P}\, NAME \\
Class : \mathbb{P}\, NAME \\
Hwm : NAME \pfun \mathbb{P}\, NAME \\
\eta : NAME \pfun NAME \\
MsgArea : NAME \pfun (NAME \times \mathrm{seq}\, CHAR) \\
AM : NAME \pfun \mathbb{P}(NAME \times NAME) \\
\hline
Entities \cap Class = \phi \\
Entities \cap Hwm = \phi \\
Hwm \cap Class = \phi \\
\mathrm{dom}\, \eta = Entities \\
\mathrm{dom}\, AM = Entities \\
\mathrm{dom}\, Hwm = Entities \\
\mathrm{dom}\, MsgArea = Entities \\
\mathrm{ran}\, \eta = Class \\
\mathrm{ran}\, Hwm = \mathbb{P}\, Class \\
\forall x, y \bullet (((x, y) \in \mathrm{ran}\, MsgArea) \Rightarrow (x \in Entities)) \\
\forall x, y \bullet (((x, y) \in \mathrm{ran}\, AM) \\
\qquad \Rightarrow ((x \in Class) \land (y \in Class)) \\
\hline
\end{array}
$$

The explanation about the state variables are same as defined in example 5.1. Note that the definition of the variables AM and Hwm is different since the security policy that needs to be enforced is different.

The message send operation is as follows:

$$
\begin{array}{|l}
\hline
Msg - Send \\
\hline
\Delta Msg - Model \\
Invoker\#, To? : NAME \\
Msg? : \text{seq } CHAR \\
\hline
Invoker\# \in Entities \\
To? \in Entities \\
\forall x \bullet (x \in Hwm(Invoker\#)) \\
\quad \Rightarrow (x, To?) \notin AM(Invoker\#) \\
Hwm' = Hwm \oplus \\
\qquad (To? \mapsto (Hwm(Invoker\#) \cup Hwm(To?)) \\
MsgArea' = MsgArea \oplus (To? \mapsto (Invoker\#, Msg?)) \\
\hline
\end{array}
$$

The above specification checks the validity of operation by examining whether the information flow that can occur violates the security policy as recorded in the state variable AM. If the information flow is valid then it updates the Hwm variable of the appropriate entity to reflect that the information flow has occurred.

5.5.3 Aggregation Policies

An information flow policy whose policy statements are of the form

$$\Box \mathbf{A} ((A \rightarrow C) \Rightarrow \neg (\Box \mathbf{A} (B \rightarrow C)))$$

is called an *aggregation policy* since the policy statements disallow aggregation of different classes of information at certain classes. The above policy statement means that information from A and B cannot aggregate at C. For example, the Chinese wall policy is an aggregation policy. These kinds of policies can be enforced using a dynamic labeling model as shown below.

- Construct an access matrix AM: Class $\rightarrow$ **P** (Class $\times$ Class) with the property for any class C, ((A, B) $\in$ AM(C)) means that there is a policy axiom of the form
$$\Box \mathbf{A} ((A \rightarrow C) \Rightarrow \neg (\Box \mathbf{A} (B \rightarrow C)))$$
- Every entity is assigned a label, referred to as Hwm (High-water mark), of type (**P** Class) and it is initialized to be {A}, where A is the information class associated

with the entity.

- Information is allowed to flow from an entity a of class A to another entity b of class B, if
  $\neg (\exists X, Y \times (\, (X \in Hwm\,(a)\,) \wedge (\, Y \in Hwm\,(b)\,) \wedge (\, (X, Y) \in AM\,(B)\,)\,)\,)$

- If the information is allowed to flow from entity a to entity b then Hwm of b is updated with the Hwm of a i.e.,
  $$Hwm\,(b) \;=\; (Hwm\,(a) \cup Hwm\,(b)\,)$$

Example 5.3: The state description of the messaging system described in the earlier example that enforces a given aggregation policy is as shown below.

$Msg - Model$

$$Entities : \mathbb{P}\, NAME$$
$$Class : \mathbb{P}\, NAME$$
$$Hwm : NAME \pfun \mathbb{P}\, NAME$$
$$\eta : NAME \pfun NAME$$
$$MsgArea : NAME \pfun (NAME \times \mathrm{seq}\, CHAR)$$
$$AM : NAME \pfun \mathbb{P}(NAME \times NAME)$$

$$Entities \cap Class = \phi$$
$$Entities \cap Hwm = \phi$$
$$Hwm \cap Class = \phi$$
$$\mathrm{dom}\, \eta = Entities$$
$$\mathrm{dom}\, AM = Entities$$
$$\mathrm{dom}\, Hwm = Entities$$
$$\mathrm{dom}\, MsgArea = Entities$$
$$\mathrm{ran}\, \eta = Class$$
$$\mathrm{ran}\, Hwm = \mathbb{P}\, Class$$
$$\forall x, y \bullet (((x, y) \in \mathrm{ran}\, MsgArea) \Rightarrow (x \in Entities))$$
$$\forall x, y \bullet (((x, y) \in \mathrm{ran}\, AM) \Rightarrow ((x \in Class) \wedge (y \in Class))$$

The explanation about the state variables are same as defined in example 5.1. Note that the definition of the variables AM and Hwm is different since the security policy that needs to be enforced is different.

The message send operation is as follows:

$$
\begin{array}{|l}
\hline
Msg - Send \\
\hline
\Delta Msg - Model \\
Invoker\#, To? : NAME \\
Msg? : \text{seq } CHAR \\
\hline
Invoker\# \in Entities \\
To? \in Entities \\
Hwm(Invoker\#) \cap AM(To?) = \phi \\
Hwm' = Hwm \oplus \\
\qquad (To? \mapsto (Hwm(Invoker\#) \cup Hwm(To?)) \\
MsgArea' = MsgArea \oplus (To? \mapsto (Invoker\#, Msg?)) \\
\hline
\end{array}
$$

The above specification checks the validity of operation by examining whether the information flow that occurs violates the security policy as recorded in the state variable AM. If the information flow is valid then it updates the Hwm variable of the appropriate entity to reflect that this flow has occurred.

5.5.4 Dissemination Policies

An information flow policy whose policy statements are of the form

$$\Box \mathbf{A} ((A \rightarrow B) \Rightarrow \neg (\Box \mathbf{A} (A \rightarrow C)))$$

is called *a dissemination policy.* These policies are duals of aggregation policies and are used to restrict the dissemination of information and hence the name. These policies can be enforced using a dynamic labeling method as shown below.

- Construct an access matrix AM: Class $\rightarrow$ **P** (Class $\times$ Class) with the property for any class A, ((B, C) $\in$ AM(A)) means that there is a policy axiom of the form

  $$\Box \mathbf{A} ((A \rightarrow B) \Rightarrow \neg (\Box \mathbf{A} (A \rightarrow C)))$$

- There are two labels associated with each entity called ClsDst: Entity $\rightarrow$ **P** Class and EntDst: Entity $\rightarrow$ **P** Entity. The label ClsDst keeps track of the classes to which the information from an entity is distributed to and the label EntDst keeps track of the entities whose information is present at the current entity. An entity a

of class A is initialized such that EntDst(a) = {a} and ClsDst(a) = {A}.

- Information is allowed to flow from an entity a of class A to another entity b of class B, if

$$\neg\,(\exists X, Y \bullet ((X \in EntDst(a)) \land (Y \in ClsDst(X))) \land ((Y, B) \in AM(A)))$$

- If the information is allowed to flow between entity a of class A and entity b of class B then the following changes are made to the labels.

$$EntDst(b) = EntDst(b) \cup EntDst(a)$$
$$\forall x \bullet ((x \in EntDst(b)) \Rightarrow (ClsDst(x) = (ClsDst(x) \cup \{B\})))$$

Example 5.4: The state description of the messaging system described in the earlier example that enforces a given dissemination policy is as shown below.

$$\begin{array}{|l}
\hline
Msg - Model \\
\hline
Entities : \mathbb{P}\, NAME \\
Class : \mathbb{P}\, NAME \\
ClsDst : NAME \pfun \mathbb{P}\, NAME \\
EntDst : \mathbb{P}\, NAME \\
\eta : NAME \pfun NAME \\
MsgArea : NAME \pfun (NAME \times \mathrm{seq}\, CHAR) \\
AM : NAME \pfun \mathbb{P}(NAME \times NAME) \\
\hline
Entities \cap Class = \phi \\
\mathrm{dom}\, ClsDst = Entities \\
\mathrm{dom}\, EntDst = Entities \\
\mathrm{ran}\, ClsDst = \mathbb{P}\, Class \\
\mathrm{ran}\, EntDst = \mathbb{P}\, Entities \\
\mathrm{dom}\, \eta = Entities \\
\mathrm{dom}\, AM = Entities \\
\mathrm{dom}\, MsgArea = Entities \\
\mathrm{ran}\, \eta = Class \\
\forall\, x, y \bullet (((x, y) \in \mathrm{ran}\, MsgArea) \Rightarrow (x \in Entities)) \\
\forall\, x, y \bullet (((x, y) \in \mathrm{ran}\, AM) \Rightarrow ((x \in Class) \land (y \in Class)) \\
\hline
\end{array}$$

The explanation about the state variables are same as defined in example 5.1. Note that the definition of the variables AM and Hwm is different since the security policy that needs to be enforced is different.

The message send operation is as follows.

$$
\begin{array}{|l}
\hline
Msg - Send \\
\Delta Msg - Model \\
Invoker\#, To? : NAME \\
Msg? : \text{seq } CHAR \\
\hline
Invoker\# \in Entities \\
To? \in Entities \\
\neg\ (\exists\, x, y \bullet ((x \in EntDst(Invoker\#) \land (y \in ClsDst(x))) \\
\qquad \Rightarrow ((y, \eta(To?)) \in AM(\eta(Invoker\#))))) \\
EntDst' = EntDst \oplus \\
\quad ((\eta(To?)) \mapsto (EntDst(\eta(Invoker\#)) \cup EntDst(\eta(To?)))) \\
\forall\, x \bullet ((x \in EntDst(Invoker\#)) \\
\qquad \Rightarrow (ClassDst'(x) = (ClassDst(x) \cup \{\eta(To?)\}))) \\
\forall\, x \bullet ((x \in EntDst(To?)) \\
\qquad \Rightarrow (ClassDst'(x) = (ClassDst(x) \cup \{\eta(To?)\}))) \\
\forall\, x \bullet (((x \notin EntDst(Invoker\#)) \land (x \notin EntDst(To?))) \\
\qquad \Rightarrow (ClassDst'(x) = ClassDst(x))) \\
MsgArea' = MsgArea \oplus (To? \mapsto (Invoker\#, Msg?)) \\
\hline
\end{array}
$$

The above specification checks the validity of the message send operation by examining whether the information flow that occurs violates the security policy as recorded in the state variable AM. If this information flow is valid then it updates the Hwm variable of the appropriate entity to reflect that this flow has occurred.

5.6 Conclusions

In this chapter a framework for verifying the access control policies and information flow policies of systems described using the specification language Z has been developed. This framework has been demonstrated for access control policies by verifying the access control restrictions for can-share predicate of take-grant model. This framework was also used for verifying the MLS policy, which is an information flow policy, in a system similar to the take-grant model. We also identified some common kinds of information flow statements that can be constructed using temporal logic and gave a generic labeling mechanism for enforcing the policies that contain these statements.

Remember that the most beautiful things in the world are the most useless; peacocks and lilies for instance

- John Ruskin

Chapter 6

Functionality vs. Enforcement of Security Policy

Security policies are usually safety requirements; they specify that something bad doesn't happen. Therefore, by default, a system that does nothing satisfies such a policy. This is not a major problem since it is easy to detect that a system does not do anything useful, but a more subtle problem is to detect that the functionality of the system is reduced more than what is required for enforcing a policy. In this chapter we characterize the problem of maximizing the functionality of a system that enforces a security policy, and develop a framework for verifying that a given specification is no more restrictive than necessary.

6.1 Introduction

One of the major problems in the design and implementation of complex systems is communicating requirements between the user and implementor. The use of formal specification languages mitigate this problem to some extent, but due to the size and complexity of formal specifications there is always a possibility that some details might be overlooked. One way of addressing this problem is to develop the specification in stages,

where each stage concentrates on some particular aspect of the system. One such decomposition is between the functionality and security of a system, where a functional specification is modified to produce one that satisfies the security policy. One of the problems with such an approach is that the refinement may restrict the functionality more than what is necessary to enforce the security policy. The over-restrictive enforcement of security policies are all too common is real world systems. For instance, in most of the Unix systems, the suid (set user identification) operation that is used in many system break-in's, is completely disabled, instead of fixing the actual security holes.

In Section 6.2 we motivate the problem by developing two specifications from the functional specification of take-grant model, both of which enforce the MLS policy, but one is less restrictive than the other. Section 6.3 gives a formal definition of refinement as applied to the specification framework of chapter 4. Section 6.4 gives a framework for verifying that the refinement of a functional specification that enforces a security policy is not more restrictive than necessary and Section 6.5 gives our conclusions.

6.2 Motivation

Consider the task of enforcing MLS in a system whose functional description is given by tg-system defined in Section 4.3.2. In this system, protection state is represented as a graph where nodes represent the entities in the system and arcs represent the rights between entities. The protection state can be changed by applying the operations take-rule, grant-rule and create-rule that can potentially cause information flows. The task of enforcing MLS policy in such a system can be accomplished by suitably restricting the state changing operations so that information flows that violate MLS policy cannot take place.

One way of enforcing the MLS policy is to change the description of the system to Mtg-system specified as (Mtg-model, MInit, {Mtake-rule, Mgrant-rule, Mcreate-Rule}), where the schemas are defined as shown below.

The state description is:

$$
\begin{array}{|l}
\hline
Mtg - model \\
Entities : \mathbb{P}\, NAME \\
Class : \mathbb{P}\, NAME \\
\eta : NAME \nrightarrow NAME \\
AM : NAME \times NAME \nrightarrow \mathbb{P}\, RIGHT \\
\hline
Entities \cap Class = \phi \\
\mathrm{dom}\, \eta = Entities \\
\mathrm{ran}\, \eta = Class \\
\forall\, x, y : NAME \\
\quad \bullet\ ((x, y) \in \mathrm{dom}(AM) \Rightarrow ((x \in Entities) \wedge (y \in Entities)) \\
\hline
\end{array}
$$

This schema specifies that the state of the system consists of the state variables *Entities* that is the set of users in the system, *Class* that is the set of information classes, η assigns an information class to every entity and *AM* is the structure that contains the rights that an entity has to another entity.

The state changing rules are as shown below:

$$
\begin{array}{|l}
\hline
Mtake - rule \\
\Delta tg - model \\
Invoker\#, From?, To? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Entities \\
From? \in Entities \\
To? \in Entities \\
't' \in AM((Invoker\#, From?)) \\
r? \in AM((From?, To?)) \\
(((r? =' t') \vee (r? =' r')) \Rightarrow (\eta(Invoker\#) \geq \eta(To?))) \\
(((r? =' g') \vee (r? =' w')) \Rightarrow (\eta(Invoker\#) \leq \eta(To?))) \\
AM' = AM \oplus \\
\qquad ((Invoker\#, To?) \mapsto AM((Invoker\#, To?)) \cup \{r?\}) \\
\hline
\end{array}
$$

This is similar to the take-rule defined in chapter 4 except for two additional conditions which ensure that

- If the copied right is a *'t'* or a *'r'* between the entities *Invoker#* and *To?* then the Class of *Invoker#* is greater than or equal to the Class of *To?*.
- If the copied right is a *'g'* or a *'w'* between the entities *Invoker#* and *To?* then the Class of *Invoker#* is less than or equal to the Class of *To?*.

$$
\begin{array}{|l}
\hline
Mgrant - rule \\
\Delta tg - model \\
Invoker\#, To?, For? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Entities \\
For? \in Entities \\
To? \in Entities \\
'g' \in AM((Invoker\#, To?)) \\
(((r? =' t') \bigvee (r? =' r')) \Rightarrow (\eta(To?) \geq \eta(For?))) \\
(((r? =' g') \bigvee (r? =' w')) \Rightarrow (\eta(To?) \leq \eta(For?))) \\
r? \in AM((Invoker\#, For?)) \\
AM' = AM \oplus ((To?, For?) \mapsto (AM((To?, For?)) \cup \{r?\}) \\
\hline
\end{array}
$$

This is similar to the grant-rule defined in chapter 4 except for two additional conditions which ensure that

- If the copied right is a *'t'* or a *'r'* between the entities *To?* and *For?* then the Class of *To?* is greater than or equal to the Class of *For?*.
- If the copied right is a *'g'* or a *'w'* between the entities *To?* and *For?* then the Class of *To?* is less than or equal to the Class of *For?*.

$$
\begin{array}{|l}
\hline
Mcreate - rule \\
\Delta tg - model \\
Invoker\#, New? : NAME \\
r? : \mathbb{P}\, RIGHT \\
\hline
Invoker\# \in Entities \\
New? \notin Entities \\
Entities' = Entities \cup \{New?\} \\
\eta' = \eta \oplus (New? \mapsto \eta(Invoker\#)) \\
AM' = AM \oplus ((Invoker\#, New?) \mapsto r?) \\
\hline
\end{array}
$$

This schema is similar to the create-rule, with the new condition which specifies that the Class of the newly created entity is same as the Class of the entity that creates it.

The initial state is then described by the following schema.

$$
\begin{array}{|l}
\hline
MInit \\
\Delta tg - model \\
\hline
\forall\, x, y \bullet (('r' \in AM(x, y)) \Rightarrow (\eta(x) \geq \eta(y))) \\
\forall\, x, y \bullet (('w' \in AM(x, y)) \Rightarrow (\eta(y) \geq \eta(x))) \\
\forall\, x, y \bullet ('t' \notin AM(x, y)) \\
\forall\, x, y \bullet ('g' \notin AM(x, y)) \\
\hline
\end{array}
$$

This schema states that

- a *'r'* right can exist from an entity to another entity of a equal or greater class.
- a *'w'* right can exists from an entity to another entity of a equal or lesser class.
- there are no *'t'* or *'g'* rights between any of the entities.

MLS is enforced in the above system by requiring that there be no *read-up* and no *write-down* and that the take and grant rules cannot be applied by requiring that there be no

take or grant rights in the initial state of the system i.e., the above system satisfies the following statement

$$\mathbf{A}\Box\left(\forall x, y \times \left(\begin{array}{c}((\text{'}r\text{'} \in AM(x,y) \Rightarrow (\eta(x) \geq \eta(y)))) \wedge \\ ((\text{'}w\text{'} \in AM(x,y) \Rightarrow (\eta(x) \leq \eta(y)))) \wedge \\ (((\text{'}t\text{'} \in AM(x,y)) \vee (\text{'}g\text{'} \in AM(x,y)) \Rightarrow (\eta(x) = \eta(y))))\end{array}\right)\right)$$

This statement asserts that a *'r'* right can exist from an entity to an entity of same or greater class, a *'w'* right can exist from an entity to another entity of same or lesser class and a *'t'* or a *'g'* right can exist between entities belonging to the same class. The proof of this can be easily carried out in the framework developed in chapter 5. It can be seen that this condition implies the condition for MLS and therefore we can conclude that the above specification enforces MLS.

This enforcement of MLS by the Mtg-system specification is overly restrictive; it reduces the functionality of the system more than actually required. For example, in the above system a take right can never exist between entities that belong to different classes although such a state does not violate the policy.

The Mtg-system enforces MLS by requiring that the take and grant rights be completely absent between entities of different levels. Instead MLS can be enforced by carrying out additional checks in the take-rule and the grant-rule so that they do not lead to a state where MLS policy requirements are violated. This is enforced by the specification Rtg-system which is (Rtg-model, RInit, {Rtake-rule, Rgrant-rule, Rcreate-rule}) where the schemas are defined below. Here, it should be noted that the state description of the system Rtg-model and the operations Rtake-rule, Rgrant-rule are same as Mtg-model and Mtake-rule and Mgrant-rule.

The state description is

$$
\begin{array}{|l}
\hline
Rtg - model \\
Entities : \mathbb{P}\, NAME \\
Class : \mathbb{P}\, NAME \\
\eta : NAME \nrightarrow NAME \\
AM : NAME \times NAME \nrightarrow \mathbb{P}\, RIGHT \\
\hline
Entities \cap Class = \phi \\
\mathrm{dom}\, \eta = Entities \\
\mathrm{ran}\, \eta = Class \\
\forall\, x, y : NAME \\
\quad \bullet\ ((x, y) \in \mathrm{dom}(AM) \Rightarrow ((x \in Entities) \wedge (y \in Entities)) \\
\hline
\end{array}
$$

This schema is same as the schema for the Mtg-model specified earlier in this section.

The state changing commands are:

$$
\begin{array}{|l}
\hline
Rtake - rule \\
\Delta tg - model \\
Invoker\#, From?, To? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Entities \\
From? \in Entities \\
To? \in Entities \\
't' \in AM((Invoker\#, From?)) \\
r? \in AM((From?, To?)) \\
(((r? =' t') \vee (r? =' r')) \Rightarrow (\eta(Invoker\#) \geq \eta(To?))) \\
(((r? =' g') \vee (r? =' w')) \Rightarrow (\eta(Invoker\#) \leq \eta(To?))) \\
AM' = AM \oplus \\
\qquad ((Invoker\#, To?) \mapsto AM((Invoker\#, To?)) \cup \{r?\}) \\
\hline
\end{array}
$$

This schema is same as the schema for the Mtake-rule specified earlier in this section.

$$
\begin{array}{|l}
\hline
Rgrant - rule \\
\Delta tg - model \\
Invoker\#, To?, For? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Entities \\
For? \in Entities \\
To? \in Entities \\
(((r? =' t') \lor (r? =' r')) \Rightarrow (\eta(To?) \geq \eta(For?))) \\
(((r? =' g') \lor (r? =' w')) \Rightarrow (\eta(To?) \leq \eta(For?))) \\
'g' \in AM((Invoker\#, To?)) \\
r? \in AM((Invoker\#, For?)) \\
AM' = AM \oplus ((To?, For?) \mapsto (AM((To?, For?)) \cup \{r?\}) \\
\hline
\end{array}
$$

This schema is same as the schema for the Mgrant-rule specified earlier in this section.

$$
\begin{array}{|l}
\hline
Rcreate - rule \\
\Delta tg - model \\
Invoker\#, New? : NAME \\
r? : \mathbb{P}\, RIGHT \\
c? : NAME \\
\hline
Invoker\# \in Entities \\
New? \notin Entities \\
c? \in Class \\
Entities' = Entities \cup \{New?\} \\
(('t' \in r?) \lor ('r' \in r?)) \Rightarrow (c? \leq \eta(Invoker\#)) \\
(('g' \in r?) \lor ('w' \in r?)) \Rightarrow (c? \geq \eta(Invoker\#)) \\
\eta' = \eta \oplus (New? \mapsto c?)) \\
\forall\, x, y \bullet ((((x, y) \in \mathrm{dom}(AM)) \land \\
\qquad (x \neq Invoker\#) \land (y \neq New?)) \\
\qquad \Rightarrow (AM'((x, y)) = AM((x, y)))) \\
\forall\, x \bullet (((x \in Entities) \land \ (x \neq Invoker\#)) \\
\quad \Rightarrow (AM'((x, New?)) = \phi)) \\
AM'((Invoker\#, New?)) = r? \\
\hline
\end{array}
$$

Here, unlike the Mcreate-rule, an entity can create another entity belonging to any class and get:

- a *'r'* or a *'t'* right if the class of the new entity is less than or equal to its class and
- a *'g'* or a *'w'* right if the class of the new entity is greater than or equal to its class.

The initial state is

$$
\begin{array}{|l}
\hline
RInit \\
\Delta tg - model \\
\hline
\forall x, y \bullet (('r' \in AM(y,x)) \Rightarrow (\eta(x) \leq \eta(y))) \\
\forall x, y \bullet (('w' \in AM(x,y)) \Rightarrow (\eta(y) \leq \eta(x))) \\
\forall x, y \bullet (('t' \in AM(y,x)) \Rightarrow (\eta(x) \leq \eta(y))) \\
\forall x, y \bullet (('g' \in AM(x,y)) \Rightarrow (\eta(y) \leq \eta(x))) \\
\hline
\end{array}
$$

This schema states that:

- a *'r'* or a *'t'* right can exist from an entity to another entity of a equal or greater class.
- a *'w'* or a *'g'* right can exists from an entity to another entity of a equal or lesser class.

The state changing operations in the above specification enforce the condition that a *'t'* or a *'r'* right can exist from an entity at a higher level to an entity at a lower level and the grant or a write right can exist from an entity at a lower level to an entity at a higher level i.e., the specification enforces the following property.

$$
\mathbf{A}\Box\left(\forall x, y \times \left(\begin{array}{l} ((('t' \in AM(x,y)) \vee ('r' \in AM(x,y))) \Rightarrow (\eta(x) \geq \eta(y))) \wedge \\ ((('g' \in AM(x,y)) \vee ('w' \in AM(x,y))) \Rightarrow (\eta(x) \leq \eta(y))) \end{array} \right) \right)
$$

Thus information can flow from a lower-level entity to a higher-level entity and not vice-versa — which is the MLS policy. The proof that the above statement is true in the set

of traces generated by the Rtg-system can be carried out easily in the framework developed in Chapter 5.

It should be noted that both Mtg-system and the Rtg-system specifications, which enforce MLS policy, are refinements of the take-grant model. The difference between these two specifications is that Rtg-model is more expressive since, unlike Mtg-model, it allows the *'t'* and *'g'* rights to exist between entities belonging to different classes.

6.3 Refinement and Redundancy of System Specifications

In this section we characterize the process of developing a specification that enforces a security policy from the functional specification of the system. We also develop the notion of Nonredundant specifications, and show how a given specification can be proved Nonredundant.

6.3.1 Refinement of a Specification

The functional description of a system is modeled in Z as a state transition machine $S = (s, I, \{Op_1, Op_2, ..., Op_n\})$ where

- s is a schema that gives the state description,
- I is the schema that gives the initial state, and
- Op_i is a schema that describes a state transition operation.

The state description of the system contains information about the state variables, their types, and the conditions that are satisfied by these variables. The schema for the initial state describes the initial values of the state variables and the schemas of the operations describe the conditions under which an operation can take place and its affects on the state of the system. For example, in the take-grant model defined in chapter 4, the schema *tg-model* gives the state description, the schema *Init* gives the initial state and the schemas *take-rule*, *grant-rule* and *create-rule* give the operation descriptions.

As specified in chapter 4, an information system is considered as a generator of a set of traces. The traces generated by a system S whose specification is given by (s, I, {Op_1, Op_2, ..., Op_n}) is denoted as T_S and is expressed as:

$$I \wedge \mathbf{C}\,(Op_1, Op_2, \ldots, Op_n)$$

The specification of a system that enforces a security policy is developed in two stages. In the first stage, a functional specification is developed that just describes the functionality of the system without considering the policy that is to be enforced. An implication of this is — the set of traces generated by the functional specification do not satisfy the properties of the security policy. In the next stage, this functional specification is modified suitably so that the set of traces generated by the new specification possess the required properties. These modifications to the functional specification consist of:

- additions to the state description that incorporate the information relevant to the security policy, and
- changes to the initial state and the operation descriptions to preclude the traces that do not satisfy the policy.

This new specification is said to be a refinement of the functional specification. Formally refinement can be defined as shown below.

Definition: A system (Rs, RI, {ROp_1, ROp_2, ..., ROp_n}) is said to be a refinement of (s, I, {Op_1, Op_2, ..., Op_n}) if

- every state variable of s is in Rs,
- (RI $\Rightarrow$ I), and
- ($ROp_1 \Rightarrow Op_1$), ($ROp_2 \Rightarrow Op_2$) ..., ($ROp_n \Rightarrow Op_n$).

It should be noted that the set of traces, T_{RS} generated by a system RS which is a refinement of a system S is such that $T_{RS} \subseteq T_S$.

6.3.2 Nonredundant Specifications

In the previous section we stated that the refinement of a specification is developed by incorporating additional conditions into the specification of the operations that try to enforce the security policy. Since this checking of additional conditions can be expensive, they must be as weak as possible. In this section we develop the notion of Nonredundant specification which captures this aspect.

Definition: Let S be a system and let RS be its refinement. The specification RS is said to be Nonredundant if there is no system RRS, which is a refinement of RS, such that $(T_{RRS} = T_{RS})$.

Thus a refinement is Nonredundant if the set of traces that it generates cannot be produced by a weaker specification, i.e., a refinement of itself. Nonredundancy is a desirable property for a specification because it places least amount of restriction on the functionality of the system.

For example, consider the Mtg-model specified in Section 6.2 that is a refinement of the tg-model specified in Section 4.3.2. This specification is not Nonredundant since there is no need to check for the information class of the entities involved in take-rule and grant-rule. In this system, a *'t'* right or a *'g'* right can only exist between entities of the same class i.e., it can be proved that the following statement is true

$$\mathbf{A}\Box\ (\forall x, y \times (\ (\ ('t' \in AM(x,y))\ \vee\ ('g' \in AM(x,y))\ ) \Rightarrow (\eta(x) = \eta(y))\ ))$$

which makes the checking for the class of entities in the take-rule and the grant-rule redundant.

In fact the MMtg-system whose specification is (Mtg-model, MInit, {MMtake-rule, MMgrant-rule, Mcreate-rule}) where the schemas MMtake-rule and MMgrant-rule are given below enforce the MLS policy.

$MMtake - rule$

$\Delta tg - model$
$Invoker\#, From?, To? : NAME$
$r? : RIGHT$

$Invoker\# \in Entities$
$From? \in Entities$
$To? \in Entities$
$((r? =' r') \Rightarrow (\eta(Invoker\#) \geq \eta(To?)))$
$((r? =' w') \Rightarrow (\eta(Invoker\#) \leq \eta(To?)))$
$'t' \in AM((Invoker\#, From?))$
$r? \in AM((From?, To?))$
$AM' = AM \oplus$
$\quad ((Invoker\#, To?) \mapsto AM((Invoker\#, To?)) \cup \{r?\})$

This is similar to the Mtake-rule of Section 6.2 except that the class of the entities is not checked when the right to be copied is either a *'t'* or a *'g'*.

$MMgrant - rule$

$\Delta tg - model$
$Invoker\#, To?, For? : NAME$
$r? : RIGHT$

$Invoker\# \in Entities$
$For? \in Entities$
$To? \in Entities$
$((r? =' r') \Rightarrow (\eta(To?) \geq \eta(For?)))$
$((r? =' w') \Rightarrow (\eta(To?) \leq \eta(For?)))$
$'g' \in AM((Invoker\#, To?))$
$r? \in AM((Invoker\#, For?))$
$AM' = AM \oplus ((To?, For?) \mapsto (AM((To?, For?)) \cup \{r?\})$

This is similar to the Mgrant-rule of Section 6.2 except that the class of the entities is not checked when the right to be copied is either a *'t'* or a *'g'*.

In fact the Mtg-system and the MMtg-system produce same set of traces but MMtg-system is Nonredundant whereas Mtg-system is not Nonredundant.

6.3.3 Proving a Specification Nonredundant

In this section we will develop a procedural framework for verifying that a given specification is Nonredundant and show that the MMtg-system specification of the previous section is Nonredundant.

In order to prove that a specification (s, I, $\{Op_1, Op_2, ..., Op_n\}$) is Nonredundant, one must prove that all the pre-conditions associated with every operation Op_i are indeed used to decide the validity of that operation in some state that is reachable from the initial state. This can be done by showing that the following statement is true for every pre-condition C in all the schemas that describe the state changing operations.

$$Init \wedge \mathbf{C}(Op_1, \ldots, Op_n) \Rightarrow \mathbf{E} \Diamond (\neg C')$$

That is, a state where the condition C is false is reachable from the initial state by the application of the state transition operations of the system.

In order to show that the MMtg-system is Nonredundant, prove that the set of all the traces generated by it, which is described by the expression,

$$MInit \wedge \mathbf{C}(MMtake-rule, MMgrant-rule, Mcretae-rule)$$

possess the following properties. (These are obtained by considering each pre-condition of the operations of the MMtg-model and substituting them in the condition given at the beginning of this section)

$$\mathbf{E} \Diamond (Invoker\# \notin Entities') \tag{6.1}$$

$$\mathbf{E} \Diamond (From? \notin Entities') \tag{6.2}$$

$$\mathbf{E} \Diamond (To? \notin Entities') \tag{6.3}$$

$$\mathbf{E} \Diamond ('t' \notin AM'(Invoker\#, From?)) \tag{6.4}$$

$$\mathbf{E} \Diamond (r? \notin AM'(From?, To?)) \tag{6.5}$$

$$\mathbf{E} \Diamond (For? \notin Entities') \tag{6.6}$$

$$\mathbf{E} \Diamond ('g' \notin AM' (Invoker\#, To?)) \tag{6.7}$$

$$\mathbf{E} \Diamond (r? \notin AM' (Invoker?, For?)) \tag{6.8}$$

$$\mathbf{E} \Diamond (New? \in Entities') \tag{6.9}$$

It can be easily seen that all the above conditions are true in the initial state itself since the conditions in the schema given for the initial state do not contradict any of the statements. Hence it can be concluded that the specification of the MMtg-model is Nonredundant.

6.4 Enforcing Security Policies with Maximum Functionality

In the previous section, we showed that the specification of a system that satisfies a security policy needs to be the Nonredundant i.e., weakest possible one to generate the required set of traces. Unfortunately, a Nonredundant specification does not mean that the set of traces generated by it is the largest possible under the policy constraints. In this section we give a procedure for showing that the set of traces generated by a refinement specification is the largest allowed by the policy.

Ideally, a refinement specification should allow any finite trace of the functional specification that does not violate the given security policy. We can prove that a refinement satisfies this property by applying the induction principle on the length of the traces. The basis step is on a trace of length one i.e., any initial state of the specification that does not violate the security policy is also a valid initial state in the refinement specification. For the induction step, we prove that a trace t of length (n+1) can be generated by the refinement where

- the trace t satisfies the security policy and
- trace t is derived from a trace t' of length n by the application of a single transition operation where t' also satisfies the security policy that is generated by both the

functional and the refinement specifications.

In order to carry out the above proof we first need to characterize the set of all initial states that do not violate the security policy. The *normal form* of a temporal formula helps in characterizing these set of states.

Definition: A temporal formula X is said to be in *normal form* if

- X does not contain any temporal operators or
- X is of the form $(\mathbf{A}\Box Y)$ or $(\mathbf{E}\Box Y)$ or $(\mathbf{A}\Diamond Y)$ or $(\mathbf{E}\Diamond Y)$ and Y is in normal form or
- X is of the form $(Y \vee Z)$ or $(Y \wedge Z)$ and Y and Z are in normal form.

Definition: A temporal formula P can be reduced to its normal form by the following actions.

- If P is a formula that does not contain any temporal operators then it is already in normal form.
- If P is of the form $(\neg(X \wedge Y))$ then replace it with $(X' \vee Y')$ where X' is the normal form of $(\neg X)$ and Y' is the normal form of $(\neg Y)$.
- If P is of the form $(\neg(X \vee Y))$ then replace it with $(X' \wedge Y')$ where X' is the normal form of $(\neg X)$ and Y' is the normal form of $(\neg Y)$.
- If P is of the form $(\neg(\mathbf{A}\Box X))$ then replace it with $(\mathbf{E}\Diamond(X'))$ where X' is the normal form of $(\neg X)$.
- If P is of the form $(\neg(\mathbf{E}\Box X))$ then replace it with $(\mathbf{A}\Diamond(\neg X'))$ where X' is the normal form of $(\neg X)$.

- If P is of the form $(\neg(\mathbf{E}\Diamond X))$ then replace it with $(\mathbf{A}\Box(X'))$ where X' is the normal form of $(\neg X)$.
- If P is of the form $(\neg(\mathbf{A}\Diamond X))$ then replace it with $(\mathbf{E}\Box(X'))$ where X' is the normal form of $(\neg X)$.
- If P is of the form $(\neg(\overline{X}))$ then replace it with $(\overline{\Diamond}X')$ where X' is the normal form of $(\neg X)$.
- If P is of the form $(\neg(\overline{\Diamond}X))$ then replace it with $(\overline{X}')$ where X' is the normal form of $(\neg X)$.
- If P is of the form $(\mathbf{X}X)$ where **X** is one of the operators $\{\mathbf{A}\Diamond, \mathbf{A}, \mathbf{E}\Diamond, \mathbf{E}, \overline{\Diamond}, \overline{\ }\}$ then replace it with $(\mathbf{X}X')$ where X' is the normal form of X.

It can be seen that any temporal formula can be reduced to a normal form since any formula that is not in normal form can be reduced to normal form by the application of the above rules (note that the above list is complete in that it gives the actions to be performed for any form of temporal formula).

Definition: Given a security policy SP = (C, Op, P, A), the set of all initial states that do not violate the requirements of the security policy are characterized by the expression denoted by I^{SP} and it is obtained from the normal form of the policy statement by the following actions.

- replace any subformula of the form $(\mathbf{A}\Box X)$ or $(\mathbf{E}\Box X)$ or $(\overline{X})$ or $(\overline{\Diamond}X)$ with X and
- replace any subformula of the form $(\mathbf{A}\Diamond X)$ or $(\mathbf{E}\Diamond X)$ with *true*.

For example, in the case of MLS security policy and the mtg-model specified in Section 5.4.1, I^{MLS} is as shown below

$$\forall X, Y \times ((X \rightarrow Y) \Rightarrow (X \leq Y))$$

Definition: Given a security policy SP and a system S, I_S^{SP} denotes the set of states of S where the policy expression I^{SP} is true.

For example, in the case of MLS security policy and the tg-model specified in chapter 4, $I_{tg-model}^{MLS}$ is the expression

$$\forall X, Y, x, y \times \left(\left(\begin{array}{c} (('r' \in AM(y,x)) \vee ('t' \in AM(y,x)) \vee \\ ('g' \in AM(x,y)) \vee ('w' \in AM(x,y)) \end{array} \right) \wedge \begin{array}{c} \\ \\ \\ \\ \end{array} \right) \Rightarrow (X \leq Y) \left(\begin{array}{c} \\ \\ \\ \\ \end{array} \right)$$
$$((\eta(x) = X) \wedge (\eta(y) = Y))$$

For a security policy SP and a system S, the set of states I_S^{SP} characterizes all possible initial states of S which can potentially generate trace sequences that satisfy the policy statements of SP. Therefore, a refinement of S that needs to have maximum functionality while enforcing the policy SP needs to have its set of initial states described by the expression I_S^{SP}. Also, this refinement should be such that it can generate any trace of S that starts with a state described by I_S^{SP} that satisfies the security policy statements. Now, to prove that a refinement RS = (Rs, RI, {ROp_1, ROp_2, .., ROp_n}) of a system S = (s, I, {Op_1, Op_2, ..., Op_n}) that enforces the security policy SP, has maximum functionality, it is sufficient to prove that

- *Basis Step:* ($I_S^{SP} \Rightarrow$ RI) (note that the fact RS enforces SP means that RI $\Rightarrow I_S^{SP}$)
- *Induction Step:* For every operation ROp_i that is a refinement of the operation Op_i and for every pre-condition C_j of (ROp_i - Op_i) prove that there exists a policy statement from the set of policy statements (a_1, a_2, ..., a_m) such that the following

condition is satisfied.

$$(RInit \wedge \mathbf{D}(\mathbf{C}(Op_1, \ldots, Op_n), (\neg C_j \wedge (ROp_i - C_j)))) \Rightarrow \mathbf{A} \Diamond (\neg a)$$

That is, the refinement RS should be such that for every state reachable from the initial state where a pre-condition (present only in the refined operation) is false, execution of that operation without the pre-condition violates at least one statement of the security policy.

Theorem: Given the system RS = (Rtg-model, RInit, {Rtake-rule, Rgrant-rule, Rcreate-rule}) given in Section 6.2 that enforces Multi-Level Security Policy, which is a refinement of the system S = (tg-model, Init, {take-rule, grant-rule, create-rule}) defined in Section 4.3.2, prove that it has maximum functionality allowed by the security policy.

Proof:

Step 1:

The set of initial states $I_{tg-model}^{MLS}$ is described by the following expression

$$\forall X, Y, x, y \left(\left(\begin{pmatrix} ('r' \in AM(y,x)) \vee ('t' \in AM(y,x)) \vee \\ ('g' \in AM(x,y)) \vee ('w' \in AM(x,y)) \end{pmatrix} \wedge \right) \Rightarrow (X \leq Y) \right)$$
$$((\eta(x) = X) \wedge (\eta(y) = Y))$$

It can be easily seen that the $\left(I_{tg-model}^{MLS} \Rightarrow Init \right)$.

Step 2:

Prove that the following statement is true

$$(RInit \wedge \mathbf{D}(\mathbf{C}(Op_1, \ldots, Op_n), (\neg C_j \wedge (ROp_i - C_j)))) \Rightarrow \mathbf{A} \Diamond (\neg(a))$$

where

- ROp_i is an operation in the set {Rtake-rule, Rgrant-rule, Rcreate-rule}

- Op_i is an operation from the set {take-rule, grant-rule, create-rule} such that ROp_i is a refinement of Op_i,
- C_j is a pre-condition in (ROp_i - Op_i) and
- a is the MLS policy statement

Now consider the take-rule to be Op_i and the pre-condition C_j to be the condition from the body of the schema describing the Rtake-rule

$$(((r? = 't') \vee (r? = 'r')) \Rightarrow (\eta (Invoker\#) \geq \eta (To?)))$$

By Rule 10 of Section 5.2 which states that,

$$\frac{\begin{array}{c}(I \Rightarrow I_1) \\ ((I_1 \wedge X_1) \Rightarrow \mathbf{A} \Diamond (I_2)) \wedge \ldots \wedge ((I_n \wedge X_n) \Rightarrow \mathbf{A} \Diamond (I_{n+1})) \\ (I_{n+1} \Rightarrow P)\end{array}}{I \wedge \mathbf{D} (X_1, \ldots, X_n) \Rightarrow \mathbf{A} \Diamond P}$$

it is sufficient to prove the following statements in order to prove the theorem.

$$(RInit \Rightarrow I_1) \tag{6.1}$$

$$(I_1 \wedge \mathbf{C} (take-rule, grant-rule, create-rule)) \Rightarrow \mathbf{A} \Diamond (I_2) \tag{6.2}$$

$$\left(\neg \left(\left(\left(\begin{array}{c} (r? = 't') \vee \\ (r? = 'r') \end{array} \right) \Rightarrow (\eta (Invoker\#) \geq \eta (To?)) \right) \right) \div \begin{array}{c} \\ \\ \wedge (Rtake-rule-mod) \wedge I_2 \end{array} \right) \Rightarrow \mathbf{A} \Diamond (\neg a) \tag{6.3}$$

where *Rtake-rule-mod* is described by the schema similar to the schema for the *Rtake-rule* given in Section 6.2 except that the pre-condition that checks for the copying of the *'t'* and *'r'* rights is removed.

$$
\begin{array}{|l}
\hline
Rtake - rule - mod \\
\Delta tg - model \\
Invoker\#, From?, To? : NAME \\
r? : RIGHT \\
\hline
Invoker\# \in Entities \\
From? \in Entities \\
To? \in Entities \\
't' \in AM((Invoker\#, From?)) \\
r? \in AM((From?, To?)) \\
(((r? =' g') \bigvee (r? =' w')) \Rightarrow (\eta(Invoker\#) \leq \eta(To?))) \\
AM' = AM \oplus \\
\qquad ((Invoker\#, To?) \mapsto AM((Invoker\#, To?)) \cup \{r?\}) \\
\hline
\end{array}
$$

and some conditions I_1 and I_2 where I_1 is

$$
\begin{array}{c}
\forall x, y \times (\ ('r' \in AM(x, y)\ ) \Rightarrow (\eta(x) \geq \eta(y))\ ) \wedge \\
\forall x, y \times (\ ('t' \in AM(x, y)\ ) \Rightarrow (\eta(x) \geq \eta(y))\ ) \wedge \\
\forall x, y \times (\ ('g' \in AM(x, y)\ ) \Rightarrow (\eta(x) \leq \eta(y))\ ) \wedge \\
\forall x, y \times (\ ('w' \in AM(x, y)\ ) \Rightarrow (\eta(x) \leq \eta(y))\ )
\end{array}
$$

and I_2 to be *true*.

case 1: Prove that $(RInit \Rightarrow I_1)$.

It is easy to see from the definitions of *Init* and I_1 the above statement is true.

case 2: Prove that

$$(I_1 \wedge \mathbf{C}(Rtake - rule, Rgrant - rule, Rcreate - rule)\ ) \Rightarrow \mathbf{A} \Diamond\ (true)$$

This is again trivially true.

case 3: Prove that

$$\left(\begin{array}{c} \neg(\ (\ (\ (r? = 't') \vee (r? = 'r')\ ) \Rightarrow (\eta(Invoker\#) \geq \eta(To?))\ )\ ) \\ \wedge\ (Rtake - rule - mod) \wedge I_2 \end{array}\right) \div \Rightarrow \mathbf{A} \Diamond\ (\neg a)$$

From the statement of the MLS policy and since the antecedent does not contain any trace operators, the above statement can be simplified to

$$\left(\begin{array}{c}\neg(((r? = 't') \vee (r? = 'r')) \Rightarrow (\eta(Invoker\#) \geq \eta(To?)))) \wedge \\ (Rtake - rule - mod)\end{array}\right)$$
$$\Rightarrow$$
$$\exists x, y \times \left(\left(\begin{array}{c}('r' \in AM'(y,x)) \vee ('t' \in AM'(y,x)) \vee \\ ('g' \in AM'(x,y)) \vee ('w' \in AM'(x,y))\end{array}\right) \wedge (\eta(x) > \eta(y))\right)$$

Expanding the antecedent and simplifying the terms we get

$$\forall x, y \times (('r' \in AM(x,y)) \Rightarrow (\eta(x) \geq \eta(y))) \tag{6.4}$$
$$\forall x, y \times (('t' \in AM(x,y)) \Rightarrow (\eta(x) \geq \eta(y))) \tag{6.5}$$
$$\forall x, y \times (('g' \in AM(x,y)) \Rightarrow (\eta(x) \leq \eta(y))) \tag{6.6}$$
$$\forall x, y \times (('w' \in AM(x,y)) \Rightarrow (\eta(x) \leq \eta(y))) \tag{6.7}$$
$$((r? = 'g') \vee (r? = 'w')) \Rightarrow (\eta(Invoker\#) \leq \eta(To?)) \tag{6.8}$$
$$((r? = 't') \vee (r? = 'r')) \tag{6.9}$$
$$\neg(\eta(Invoker\#) \geq \eta(To?)) \tag{6.10}$$
$$Invoker\# \in Subjects \tag{6.11}$$
$$From? \in Entities \tag{6.12}$$
$$To? \in Entities \tag{6.13}$$
$$'t' \in AM(Invoker\#, From?) \tag{6.14}$$
$$r? \in AM(From?, To?) \tag{6.15}$$
$$AM' = AM \oplus \left(\begin{array}{c}(Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\})\end{array}\right) \tag{6.16}$$

Now Instantiating the consequent of the above implication with (x = To?) and (y = Invoker#) we get

$$\left(\begin{array}{c}('r' \in AM'(Invoker\#, To?)) \vee ('t' \in AM'(Invoker\#, To?)) \vee \\ ('g' \in AM'(To?, Invoker\#)) \vee ('w' \in AM'(To?, Invoker\#))\end{array}\right)$$
$$\wedge$$
$$(\eta(To?) > \eta(Invoker\#))$$

Dropping some terms in the antecedent of the above expression we get

$$('r' \in AM'(Invoker\#, To?)) \vee ('t' \in AM'(Invoker\#, To?))$$
$$\wedge$$
$$(\eta(To?) > \eta(Invoker\#))$$

Substituting for AM' from equation 6.32 we get

$$\left(\begin{array}{l}\left('r' \in \left(AM \oplus \left(\begin{array}{c}(Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\})\end{array}\right)\right)(Invoker\#, To?)\right) \vee \\ \left('t' \in \left(AM \oplus \left(\begin{array}{c}(Invoker\#, To?) \rightarrow \\ (AM(Invoker\#, To?) \cup \{r?\})\end{array}\right)\right)(Invoker\#, To?)\right)\end{array}\right)$$
$$\wedge$$
$$(\eta(To?) > \eta(Invoker\#))$$

From the definition of the override operator, the above statement can be transformed into:

$$('r' \in ((AM(Invoker\#, To?) \cup \{r?\}))) \vee ('t' \in (AM(Invoker\#, To?) \cup \{r?\}))$$
$$\wedge$$
$$(\eta(To?) > \eta(Invoker\#))$$

Now from the properties of sets and the definition of '>' operator, the above statement can be transformed into:

$$((r' = r?) \vee (t' = r?)) \wedge \neg(\eta(Invoker\#) \geq \eta(To?))$$

The above statement is true from equations 6.25 and 6.26.

A similar approach can be used to prove that the other conditions and other operations also satisfy the requirements specified in the framework for proving that a system enforces a security policy with maximum functionality.

In this section we characterized the notion of a specification enforcing a security policy with maximum functionality and developed a framework for verifying it. We also

demonstrated this framework by verifying that the a given refinement specification of the take-grant model enforces the MLS policy with maximum functionality.

6.5 Conclusions

In this chapter we showed that the functionality of a system can be reduced more than required while enforcing a security policy. We developed a framework to verify that the specification of a system that must enforce a security policy does so without reducing the functionality more than what is absolutely necessary using the verification framework Chapter 5.

Life is an art of drawing sufficient conclusions from insufficient premises

- Samuel Butler

Chapter 7

Conclusions

An important method of increasing the confidence in the security of a system is by verifying that it satisfies the required security properties. In this thesis we developed a comprehensive framework for specifying the security policies and protection features of information systems and used them to develop a framework to verify that the system specification enforces the policy correctly. This framework has the advantage that it is simple and fairly mechanical so that automatic theorem provers can be used in the verification process.

We developed a many-sorted predicate logic based language for specifying the information flow properties of security critical systems. This language can be used for specifying both the information flow predicates and their composition constructs. We developed a framework for proving the composition properties of different information flow predicates. The most important advantage of this formalism is that it can be automated easily. We showed that the composition properties can be proved using a theorem prover such as PVS.

We developed a temporal logic based language for specifying the security policies. We showed that this language can be used to specify different kinds of security policies in an unambiguous and succinct fashion. The formal semantics of this language are given with respect to a state machine that enables one to clearly interpret the security policy without any ambiguity.

We proposed that information systems can be described as state transition machines using the formal specification language Z. We also developed a trace specification language that can be used to specify the set of traces in a compact way about which security properties need to be proved.

We developed a framework for proving the enforcement of security policies of information systems. In this framework

- the information system is characterized by the set of traces generated by its specification that are compactly specified using the trace expression operators and
- the security policy is specified as set of temporal properties that must be satisfied by these traces.

Our framework for proving that a system enforces a given security policy consists of a set of rules that depend on the structure of the trace expressions and the temporal property in the policy. Our experience in proving the security properties is that the design of secure systems is an iterative process where verification brings out problems that are fixed in the next step of the design process. Also, we feel that the proof framework developed in this thesis can be automated and we intend to pursue this line of research in the future.

We observed that system specifications are developed in different stages where each stage tries to specify some aspect of the system. In the case of security policy enforcement, it was observed that the functional specification of the system is developed initially and

then it is refined by adding security relevant information to suitably restrict its behavior so that the security policy is enforced. One of the problems with this approach is that one has to be careful so as not to restrict the functionality of the system more than what is actually required. We developed a framework for proving that a specification of a system is not over-restrictive with respect to a given security policy using the proof rules developed for security property verification. This suggests that the framework that we proposed is general enough to handle an interesting class of verification problems. We intend to investigate the applicability of this framework to other kinds of system properties like safety and fault-tolerance.

Bibliography

[AL90] Abadi, M., Lamport, L., 'Composing Specifications', DEC-SRC Technical Report 66, 1990.

[AMM92] Amman, P. E., Sandhu, R. S., 'The Extended Schematic Protection Model', Journal of Computer Security, Vol 1, nos 3, 4, 1992.

[AP92] Abadi, M., Plotkin, G. D., 'A Logical View of Composition', Digital Technical Report 86, 1992.

[AS85] Alpern, B., Schneider, F., 'Defining Liveness', Information Processing Letters, October 1985.

[BAD89] Badger, L., 'A Model for Specifying Multi-Granularity Integrity Policies', Proceedings of the IEEE Symposium on Security and Privacy, July 1989.

[BAA90] Benson, G. S., Akyildiz, I. F., Appelbe, W. F., 'A Formal Protection Model of Security in Centralized, Parallel and Distributed Systems', ACM transactions on Computer Systems, August 1990.

[BEL88] Bell, D.E., 'Concerning Modeling of Computer Security', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[BIS84] Bishop, M., 'Practical take-grant systems: Do they exist?', Ph.D Dissertation, Purdue University, May 1984.

[BIS91] Bishop, M., 'Theft of Information in Take-Grant Protection Model', Dartmouth Technical Report 1991.

[BKY85] Boebert, W. E., Kain, R. Y., Young, W. D., Hanson, S. A., 'Secure ADA Target: Issues, System Design and Verification', Proceedings of the IEEE Symposium on Security and Privacy, 1985, pp 176-183.

[BL74] Bell, D. E., LaPadula, L. J., 'Secure Computer Systems', Tech Rep. ESD-TR-73-278, vols 1-3, MITRE, 1974.

[BN89] Brewer, D. F. C., Nash, M. J., 'The Chinese Wall Security Policy', Proceedings of the IEEE Symposium on Security and Privacy, July 1989.

[CFG87] Cummings, P. T., Fullam, D. A., Goldstein, M. J., Gosselin, M. J., Picciotto, J., Woodward, J. P. L., Wynn, J., 'Compartmented Mode Workstation: Results through prototyping', Proceedings of the IEEE Symposium of Security and Privacy, 1987.

[CHE81] Cheheyl, M., et. al., 'Verifying Security', Computer Surveys, September 1981.

[CUM91] Cumming, J. G., 'On Refinement of Non-Interference', Proceedings of the IEEE Symposium of Security and Privacy, 1991.

[CUP93] Cuppens, F., 'A Logical Analysis of Authorized and Prohibited Information Flows', Proceedings of the IEEE Symposium on Security and Privacy, 1993.

[CW87] Clark, D. D., Wilson, D. R., 'A Comparison of Commercial and Military Computer Security Policies', Proceedings of the IEEE Symposium on Security and Privacy, April 1987.

[DEN76] Denning, D., 'A Lattice Model of Secure Information Flow', Communications of the ACM, May 1976.

[DEN83] Denning, D., 'Cryptography and Data Security', Addison Wesley, 1983.

[DL88] Denning, D. R., Lunt, T. F., 'The SeaView Security Model', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[DOB89] Dobson, J.E., McDermid, J. A., 'A Framework for Expressing Models of Security Policy', Proceedings of the IEEE Symposium on Security and Privacy, July 1989.

[DOW85] Downs, D. D., et. al.,' Issues in Discretionary Access Control', Proceedings of the IEEE Symposium on Security and Privacy, April 1985.

[FAR90] Farmer, D., Spafford, E. H., 'The COPS Security Checker System', Proceedings of the summer 1990 USENIX Conference, June 1990.

[FIN90] Fine, T., 'Constructively Using Noninterference to Analyze Systems', Proceedings of the IEEE Conference on Security and Privacy, 1990.

[FOL89a] Foley, S. N., 'A Model for Secure Information Flow', Proceedings of the IEEE Conference on Security and Privacy, July 1989.

[FOL89b] Foley, S. N., 'A Taxonomy of Information Flow Policies', Proceedings of the IEEE Conference on Security and Privacy, July 1989.

[FOL90] Foley, S. N., 'Secure Information Flow Using Security Groups', Proceedings of the Computer Security Foundations Workshop III, June 1990.

[GJ79] Garey, M. R., Johnson, D. S., 'Computers and Intractability: A Guide to the Theory of NP-Completeness', W H Freeman, 1979.

[GM82] Goguen, J. A., Meseguer, J., 'Security Policies and Security Models', Proceedings of the IEEE Symposium on Security and Privacy, April 1982.

[GM84] Goguen, J. A., Meseguer, J., 'Unwinding and Inference Control', Proceedings of the IEEE Symposium on Security and Privacy, April 1984.

[GM87] Glasgow, J.I., MacEwen, G.H., 'The Development and Proof of a Formal Specification for a Multilevel Secure System', ACM Transactions on Computer Systems, May 1987.

[GM88] Glasgow, J. I., MacEwen, G. H., 'Reasoning About Knowledge in Multilevel Secure Distributed Systems', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[GM90] Glasgow, J., MacEwen, G., 'A Logic for Reasoning About Security', Proceedings of the Computer Security Foundations Workshop III, June 1990.

[GMP93] Glasgow, J., MacEwen, G., 'A Logic for Reasoning About Security', ACM Transactions in Computer Systems, 1993.

[HAR76] Harrison, M., et al, 'Protection in Operating Systems', Communications of the ACM, August 1976.

[HAR85] Harrison, M., 'Theoretical Issues Concerning Protection in Operating Systems', Advances in Computers, v24, 1985.

[JAC88] Jacob, J., 'Security Specifications', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[JL78] Jones, A. and Lipton, R., 'The Enforcement of Security Policies for Computation', Journal of Computer and Systems Sciences, August 1978.

[JON78] Jones, A., 'Protection Mechanism Models: Their Usefulness', Foundations of Secure Computation, Academic Press, 1978.

[JT88] Johnson, D.M., Thayer, F.J., 'Stating Security Requirements with Tolerable Sets', ACM Transactions on Computer Systems, August 1988.

[KL85] Kain, R. Y., Landwehr, C. E., 'On Access Checking in Capability based systems', Proceedings of the Symposium on Security and Privacy, 1985.

[LAM74] Lampson, B., 'Protection', Proceedings of the 5th Symposium on Operating Systems, January 1974.

[LAM93] Lamport, L., ‘Temporal Logic of Actions’, DEC-SRC Research Report 79, 1993.

[LAN81] Landwehr, C., 'Formal Models for Computer Security', Computing Surveys,

September 1981.

[LEE88] Lee, T. M. P., 'Using Mandatory Integrity to Enforce Commercial Security', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[LIP82] Lipner, S. B., 'Non-Discretionary Controls for Commercial Applications', Proceedings of the IEEE Symposium on Security and Privacy, 1982.

[LS77] Lipton, R and Snyder, L., 'A Linear Time Algorithm for Deciding Subject Security', Journal of the ACM, July 1977.

[MCC87] McCullough, D., 'Specifications for Multi-level Security and Hook-Up Property', Proceedings of the IEEE Symposium of Security and Privacy, 1987.

[MCC88] McCullough, D., 'Noninterference and the Composability of Security Properties', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[MCL84] McLean, J., et. al, 'A Formal Statement of the MMS Security Model', Proceedings of the IEEE Symposium on Security and Privacy, April 1984.

[MCL87] McLean, J., 'Reasoning about Security Models', Proceedings of the IEEE Symposium of Security and Privacy, 1987.

[MCL88] McLean, J., 'The Algebra of Security', Proceedings of the IEEE Symposium of Security and Privacy, 1988.

[MCL90] McLean, J., 'Security Models and Information Flow', Proceedings of the IEEE Conference on Security and Privacy, 1990.

[MCL94] McLean, J, 'A general Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions', Proceedings of the IEEE Symposium on Research in Security and Privacy, 1994.

[NEE5] Neely, R. B., Freeman, J. W., 'Structuring Systems for Formal Verification', Proceedings of the IEEE Symposium on Security and Privacy, April 1985.

[ORA85] DoD 5200.28-STD, 'Department of Defense Trusted Computer System Evaluation Criteria',1985

[POP78] Popek, G.J., Farber, D. 'A Model for Verification of Data Security in Operating Systems', Communications of the ACM, September 1978.

[PFL89] Pfleeger, C., 'Security in Computing', Prentice Hall, 1989.

[PN77] Pnueli, A., 'The Temporal Logic of Programs', Proceedings of the Eighteenth Symposium on Foundations of Computer Science, November 1977.

[PW94] Peri, R. V., Wulf, W. A., 'Formal Specification of Information Flow Security Policies and Their Enforcement in Security Critical Systems', Proceed-

ings of the Computer Security Foundations Workshop VII, Franconia, 1994.

[SAN88] Sandhu, R. S., 'The Schematic Protection Model: Its definition and Analysis for Acyclic Attenuating Schemes', JACM, April 1988.

[SAN89] Sandhu, R. S., 'Transformation of Access Rights', Proceedings of the IEEE Symposium on Security and Privacy, July 1989.

[SAN92] Sandhu, R. S., 'Typed Access matrix Mode', Proceedings of the IEEE Symposium on Security and Privacy, May 1992.

[SNY81] Snyder, L., 'Formal Models of Capability-Based Protection Systems', IEEE Transactions on Computers, March 1981.

[SPI87] Spivey, J. M., 'The Z Notation - A Reference Manual', Prentice Hall International Series in Computer Science, 1987.

[SUT86] Sutherland, D., 'A Model of Information', 9th National Security Conference, 1986.

[VIN88] Vinter, S. T., 'Extended Discretionary Access Controls', Proceedings of the IEEE Symposium on Security and Privacy, May 1988.

[WAL80] Walker, B.J., et al., 'Specification and Verification of the UCLA Unix Security Kernel', Communications of the ACM, February 1980.

[WD87] Williams, J. C., Dinolt, G. W., 'A Graph-Theoretic Formulation of Multilevel Secure Distributed Systems: An Overview', Proceedings of the IEEE Symposium on Security and Privacy, April 1987.

[WJ90] Wittbold, J. T., Johnson, D. M., 'Information Flow in Non-Deterministic Systems', Proceedings of the IEEE Conference on Security and Privacy, 1990.

[WL89] Whitehurst R. A., Lunt T. F., 'The SeaView Verification', Proceedings of the Computer Security Foundations Workshop II, June 1989.

[WN86] Wing, J. M., Nixon, M. R., 'Extending INA JO with Temporal Logic', Proceedings of the IEEE Symposium on Security and Privacy, April 1986.

[WOL87] Wolper, P., 'On the Relation of Programs and Computations to Models of Temporal Logic', Proceedings of Temporal Logic in specification, LNCS 398, April 1987.

[WUL74] Wulf, W., et al, 'Hydra - The Kernel of a Multiprocessor Operating System,' Communications of the ACM, June 1974.

[WUL80] Wulf, W., et. al., 'Hydra/C.mmp: An Experimental Computer System'. McGraw-Hill Publishing Co., 1980.

Appendix A

PVS and Information Flow Predicates

In this appendix we develop the proof that generalized Non-Interference is preserved under cascade operation using the PVS theorem prover by applying the framework that was developed in chapter 2.

A.1 Generalized NonInterference and Cascade operation in PVS

In this section we give a specification of the cascade operation and the definition of Generalized Non-Interference in the form suitable for PVS system.

```
secprop: THEORY
BEGIN

    TRACE:      TYPE
    VALSEQ:     TYPE
    HIN:        [TRACE -> VALSEQ]
    HOUT:       [TRACE -> VALSEQ]
    LIN:        [TRACE -> VALSEQ]
    LOUT:       [TRACE -> VALSEQ]
    PL:         [VALSEQ, VALSEQ -> VALSEQ]
    MUL:        [VALSEQ, VALSEQ -> VALSEQ]

    S1:         [TRACE -> boolean]
    S2:         [TRACE -> boolean]

% S is the composition of S1 and S2 under some composition construct

    S:          [TRACE ->  boolean]

% Statement that S1 satisfies Generalized Non-Interference
```

```
S1ax:
        AXIOM
            FORALL (x, y: TRACE):
                ((S1(x) AND S1(y))
                    IMPLIES
                        ((EXISTS (p: TRACE):
                            ((S1(p) AND (HIN(p) = HIN(y)) AND (LIN(p) = LIN(x))
                            AND (LOUT(p) = LOUT(x)))))
                        AND
                        ((EXISTS (q: TRACE):
                            ((S1(q) AND (HIN(q) = HIN(x)) AND (LIN(q) = LIN(y))
                            AND (LOUT(q) = LOUT(y)))))))

% Statement that S2 satisfies Generalized Non-Interference

S2ax:
        AXIOM
            FORALL (x, y: TRACE):
                ((S2(x) AND S2(y))
                    IMPLIES
                        ((EXISTS (p: TRACE):
                            ((S2(p) AND (HIN(p) = HIN(y))
                            AND (LIN(p) = LIN(x)) AND (LOUT(p) = LOUT(x))))))
                        AND
                        ((EXISTS (q: TRACE):
                            ((S2(q) AND (HIN(q) = HIN(x))
                            AND (LIN(q) = LIN(y)) AND (LOUT(q) = LOUT(y)))))))

S1S2Prop:
        AXIOM
            FORALL (x, y: TRACE):
                ((S1(x) AND S2(y))
                    IMPLIES
                        (EXISTS (z : TRACE):
                            (S2(z) AND (HIN(z) = HOUT(x))
                            AND (LIN(z) = LOUT(x)) AND (LOUT(z) = LOUT(y)))))

% Definitions of cascade operation

Sax:
        AXIOM
            FORALL (x: TRACE):
                (S(x)
                    IMPLIES
                        (EXISTS (y, z: TRACE):
                            (S1(y) AND (LIN(y) = LIN(x)) AND (HIN(y) = HIN(x)) AND S2(z)
                            AND (LOUT(z) = LOUT(x)) AND (HOUT(z) = HOUT(x))
                            AND (HIN(z) = HOUT(y)) AND (LIN(z) = LOUT(y)))))

S1S2ax:
        AXIOM
            FORALL (x, y: TRACE):
                ((S1(x) AND S2(y) AND (LIN(y) = LOUT(x)) AND (HIN(y) = HOUT(x)))
                    IMPLIES
                        (EXISTS (z: TRACE):
                            (S(z)
                            AND (LIN(z) = LIN(x)) AND (HIN(z) = HIN(x))
                            AND (LOUT(z) = LOUT(y)) AND (HOUT(z) = HOUT(y)))))

% The theorem that the system S satisfies Generalized Non-Interference
% This is the theorem that needs to be proved

Sprop:
        CONJECTURE
```

```
                FORALL (x, y: TRACE):
                    ((S(x) AND S(y))
                        IMPLIES
                            ((EXISTS (p: TRACE):
                                ((S(p) AND (HIN(p) = HIN(y))
                                AND (LIN(p) = LIN(x)) AND (LOUT(p) = LOUT(x))))))
                            AND
                            ((EXISTS (q: TRACE):
                                ((S(q) AND (HIN(q) = HIN(x))
                                AND (LIN(q) = LIN(y)) AND (LOUT(q) = LOUT(y)))))))

        END secprop
```

A.2 Proof of the composition property in PVS

This is the set of PVS commands that are used to prove the proposition Sprop given in the above specification.

```
(|secprop|
 (|Sprop| "" (SKOLEM!)
  (("" (FLATTEN)
    (("" (LEMMA "Sax" ("x" "x!1"))
      (("" (SPLIT)
        (("1" (SKOLEM!)
          (("1" (FLATTEN)
            (("1" (LEMMA "Sax" ("x" "y!1"))
              (("1" (SPLIT)
                (("1" (SKOLEM!)
                  (("1" (FLATTEN)
                    (("1" (LEMMA "S1ax" ("x" "y!2" "y" "y!3"))
                      (("1" (SPLIT)
                        (("1" (FLATTEN)
                          (("1" (SKOLEM!)
                            (("1" (SKOLEM!)
                              (("1" (FLATTEN)
                                (("1" (LEMMA "S1S2Prop" ("x" "p!1" "y" "z!1"))
                                  (("1" (SPLIT)
                                    (("1" (SKOLEM!)
                                      (("1" (FLATTEN)
                                        (("1" (LEMMA "S1S2Prop" ("x" "q!1" "y" "z!2"))
                                          (("1" (SPLIT)
                                            (("1" (SKOLEM!)
                                              (("1" (FLATTEN)
                                                (("1" (LEMMA "S1S2ax" ("x" "p!1" "y" "z!3"))
                                                  (("1" (SPLIT)
                                                    (("1" (SKOLEM!)
                                                      (("1" (FLATTEN)
                                                        (("1" (LEMMA "S1S2ax" ("x" "q!1" "y" "z!4"))
                                                          (("1" (SPLIT)
                                                            (("1" (SKOLEM!)
                                                              (("1" (FLATTEN)
                                                                (("1" (SPLIT)
                                                                  (("1" (QUANT * ("z!5"))
                                                                    (("1" (SPLIT)
                                                                      (("1" (PROPAX) NIL)
                                                                       ("2" (REPLACE -8 (1))
                                                                        (("2" (REPLACE -20 (1))
                                                                          (("2" (PROPAX) NIL)))))
                                                                       ("3" (REPLACE -7 (1))
```

```
                                        (("3" (REPLACE -21 (1))
                                          (("3" (PROPAX) NIL)))))
                                       ("4" (REPLACE -9 (1))
                                        (("4" (REPLACE -18 (1))
                                        (("4" (PROPAX) NIL)))))))))
                                     ("2" (QUANT * ("z!6"))
                                      (("2" (SPLIT)
                                        (("1" (PROPAX) NIL)
                                         ("2" (REPLACE -3 (1))
                                          (("2" (REPLACE -24 (1))
                                          (("2" (PROPAX) NIL)))))
                                         ("3" (REPLACE -2 (1))
                                          (("3" (REPLACE -25 (1))
                                            (("3" (PROPAX) NIL)))))
                                         ("4" (REPLACE -4 (1))
                                          (("4" (REPLACE -14 (1))
                                            (("4" (PROPAX) NIL)))))))))))))))
                                  ("2" (PROPAX) NIL)
                                  ("3" (PROPAX) NIL)
                                  ("4" (PROPAX) NIL)
                                  ("5" (PROPAX) NIL)))))))))
                             ("2" (PROPAX) NIL)
                             ("3" (PROPAX) NIL)
                             ("4" (PROPAX) NIL)
                             ("5" (PROPAX) NIL)))))))))
                        ("2" (PROPAX) NIL)
                        ("3" (PROPAX) NIL)))))))))
                   ("2" (PROPAX) NIL)
                   ("3" (PROPAX) NIL)))))))))))))))
              ("2" (PROPAX) NIL) ("3" (PROPAX) NIL)))))))))
         ("2" (PROPAX) NIL)))))))))
("2" (PROPAX) NIL)))))))))))
```

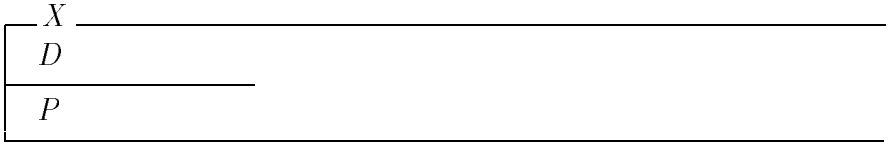
Appendix B

An Introduction to Z

Z, pronounced as zed, is a specification language based on first order predicate logic and set theory. We chose Z to specify the protection features of information systems because of its growing popularity and wide spread acceptance in the formal methods community. In this Appendix we give a brief overview of the features of Z that are used in this dissertation. More details can be found in [DIL90].

B.1 System Specification in Z

Systems that are modeled as state transition machines are specified in Z by means of a state description and a description of the transition relations. The basic building block of a Z specification is a *schema* that is a collection of variable declarations and some predicates involving these variables. A schema can be named so that it can be referenced later. The schemas are represented as two dimensional graphical constructs in Z as shown below:



Here X is the name of the schema, D is the set of variable declarations and P, usually referred to as the body of the schema, is the set of conditions that are required to be satisfied. Z defines some operations on schemas for modularizing the specifications. These operations are defined later.

The other important construct in Z is an axiomatic definition that is used to define functions. The form of an axiomatic definition is as shown below.

$$\begin{array}{|l} < Name >:< Signature > \\ \hline < Body > \\ \hline \end{array}$$

Here <Name> is the name of the function, <Signature> specifies the types of inputs and the outputs of the function and <Body> is the definition of the function.

B.2 State Description

States in Z are defined by means of state variables where every variable is associated with a type. A state variable can be introduced into the specification by a statement of the form $(var : Type)$. Types are modeled as sets in Z which include some pre-defined sets and facilities to build new sets from the existing ones.

B.2.1 Basic Sets

A basic set is introduced by the declaration [NAME] where NAME is a type. Operations on these basic types can be introduced by means of axiomatic definitions. Z includes some predefined sets like Integers with some predefined operations like addition and subtraction.

B.2.2 Set Constructors

Z provides constructors that can be used to construct sets having special properties. These new sets can be used to model arbitrary structures that occur in real world systems.

B.2.2.1 Enumeration

Sets can be created by enumerating all the elements in it. For example,

$$\{Mon, Tue, Wed, Thurs, Fri, Sat, Sun\}$$

is the set of weekdays.

B.2.2.2 Set Comprehension

Sets can be created from already existing sets by set comprehension. For example, the set of all integers, which are prefect squares, can be created from the set of Integers as shown below:

$$\{n : \mathbf{N} \bullet n \times n\}$$

B.2.2.3 Power Set

The set of all subsets of a given set is called the power set. Given a set X, its power set is represented as $(\mathbf{P}\ X)$. For example, $(\mathbf{P}\ \{1, 2\})$ represents the set.

$$\{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$$

where $\varnothing$ is the empty set.

B.2.2.4 Cross-Product

Given two sets X and Y, their cross-product represented as $(X \times Y)$ is a set of ordered pairs where each ordered pair is such that the first element is from the set X and the second element is from the set Y. For example, $(\{1, 2\} \times \{a, b\})$ represents the set

$$\{(1, a), (1, b), (2, a), (2, b)\}$$

An ordered pair (a, b) is also written as $(a \rightarrow b)$.

B.2.2.5 Relations

Given two sets X and Y, a relation from X to Y represented as $(X \leftrightarrow Y)$ is the set $(\mathbf{P}\ (X \times Y))$.

B.2.2.6 Functions

Given two sets X and Y, a function from X to Y represented as $(X \rightarrow Y)$ is a subset of $(\mathbf{P}\ (X \times Y))$ that satisfies the following property

$$\{ (F : (X \leftrightarrow Y)) \mid \forall x : X; \forall y : Y \bullet (((x, y) \in F) \wedge ((x, z) \in F)) \Rightarrow (y = z) \}$$

The above statement means that F maps an element of X to a unique element of Y.

B.2.2.7 Sequences

Sequences are written enclosed in angle brackets with their elements separated by commas as in $<x_1, x_2, ..., x_n>$. An empty sequence is written as < >.

Given a set X, all the finite sequences of elements drawn from X is denoted as $(seq\ X)$. For example, $(seq\ \{1, 2\})$ represents a set whose elements include <1>, <2>, <1, 2> etc.

B.2.3 Operations

Z provides a number of operations on sets. These operations can be classified depending on the type of the sets that they operate on. These are:

- Set Operations,
- Relation Operations, and
- Sequence Operations.

Each of the above operations are described in more detail in the following sections.

B.2.3.8 Set Operations

These are the common operations like equality (=), membership ($\in$), subset ($\subseteq$), union ($\cup$), intersection ($\cap$), set difference (-) whose semantics are well known.

B.2.3.9 Relation Operations

These operations are specific to sets that are relations. They are:

- Composition — Given two relations $(F : (X \leftrightarrow Y))$ and $(G : (Y \leftrightarrow Z))$, their composition is denoted as $(F ; G)$ and it is defined as:

  $$((x, z) \in (F ; G)) \Leftrightarrow (\exists y : Y \bullet ((x, y) \in F) \wedge ((y, z) \in G)))$$

- Domain — Given a relation $(F : (X \leftrightarrow Y))$, the domain of F is a set denoted as $(dom\ F)$ that satisfies the following property:

  $$(x \in dom\ F) \Leftrightarrow \exists y : \mathrm{Y} \bullet ((x, y) \in F)$$

- Range — Given a relation $(F : (X \leftrightarrow Y))$, the range of F is a set denoted as $(ran\ F)$ that satisfies the following property:

  $$(y \in ran\ F) \Leftrightarrow \exists x : \mathrm{X} \bullet ((x, y) \in F)$$

- Override — Given the functions $(F, G : (X \leftrightarrow Y))$, then $(F \oplus G)$ is denotes the overriding of F with G which is defined as below:

  $$(F \oplus G)\ x = \begin{matrix} g\ x & if\ \ (x \in dom\ g) \\ f\ \mathrm{x} & Otherwise \end{matrix}$$

- Domain Restriction Operators — Given a relation $(F : (X \leftrightarrow Y))$ and $(U : \mathbf{P}X)$,

  - the domain restriction of F is a relation of type $(X \leftrightarrow Y)$ which is defined as below:

    $$((a, b) \in (U \lhd F)) \Leftrightarrow (((a, b) \in F) \wedge (a \in U))$$

  - the domain co-restriction of F is a relation of type $(X \leftrightarrow Y)$ which is defined as below:

    $$((a, b) \in (U \lhd\!\!\!- F)) \Leftrightarrow (((a, b) \in F) \wedge (a \notin U))$$

- Range Restriction Operators — Given a relation $(F : (X \leftrightarrow Y))$ and $(U : \mathbf{P}\, Y)$,
  - the range restriction of F is a relation of type $(X \leftrightarrow Y)$ which is defined as below:

$$((a, b) \in (F \rhd U)) \Leftrightarrow (((a, b) \in F) \land (b \in U))$$

  - the range co-restriction of F is a relation of type $(X \leftrightarrow Y)$ which is defined as below:

$$((a, b) \in (F \rhd U)) \Leftrightarrow (((a, b) \in F) \land (b \notin U))$$

B.2.3.10 Sequence Operations

The operators for concatenation (^), head (*head*), tail (*tail*), last (*last*), front (*front*) are defined on sequences whose semantics are well known.

B.3 Constraints on States

The body of a schema contains a set of conditions that are required to be satisfied by the system. These conditions are statements of the propositional calculus or predicate calculus where the propositions or the predicates are defined in terms of the state variables described in the declaration part of the schema.

B.3.4 Propositional Calculus

Z uses propositional calculus for describing the information systems. A system is modeled by selecting a set of primitive propositions that can be either true or false and using the logical operators to build more complicated propositions that describe the system. These logical operators are negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), Implication ($\Rightarrow$) and Equivalence ($\equiv$). The semantics of these logical operators is well known.

The propositions can be simplified by a set of rules that can be used to reason about the properties of systems. The properties of a system can be proved by applying the simplification rules to the description of the system.

B.3.5 Predicate Calculus

Predicate calculus with its quantifiers, in Z, facilitates the specification of the properties of classes of objects that is tedious in propositional calculus. Predicates are boolean valued expressions that evaluate to either true or false and compound expressions are built using the propositional operators mentioned earlier. The two kinds of quantifiers are:

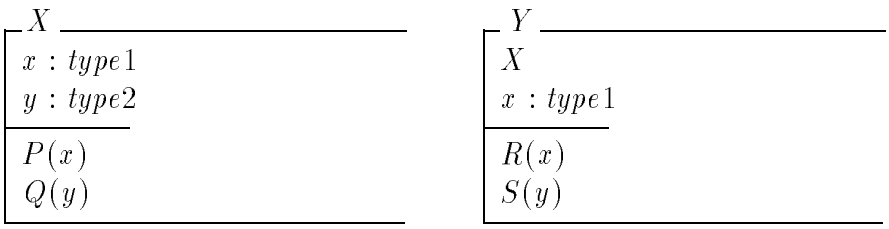
- existential quantifier (∃) which specifies that a property is true for at least one member of a class, and
- universal quantifier (∀) which specifies that a property is true for every member of the class.

B.4 The Specification Methodology

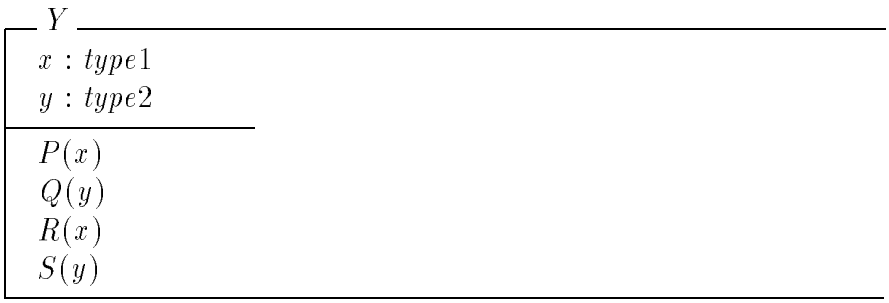
An information system, in Z, is specified as a state machine consisting of a set of states and a set of operations. The states are described by means of a schema that describes the set of state variables and any conditions that are universally true in the system. The operations are also described by schemas that specify the conditions under which an operation can take place called the pre-conditions and their affect on the state of the system called the post-conditions. In a schema, all un-primed variables refer to the state of the system before the operation and all primed variables refer to the state of the system after the operation. Therefore an expression that does not contain any primed variables is a pre-condition and an expression which has at least one primed variable is a post-condition.

In order to facilitate modular specifications, Z provides some operations on schemas. These schema operations are:

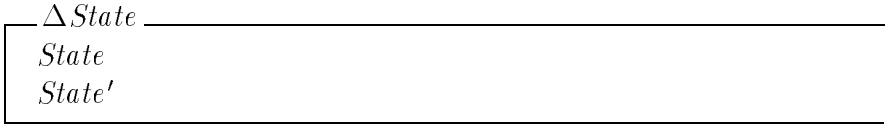
- Schema inclusion — A schema X can be included in the declaration part of another schema Y. The effect of this is that all the declarations of X become part of declarations of Y and the predicates of Y is a conjunction of the predicates of X and Y. If a variable is declared in both X and Y then it should be of the same type. For example, consider the following two schemas:



The schema Y is equivalent to:



- The Δ and Ξ conventions - The ΔState is the schema obtained by combining the before and after specifications of state i.e., it is equivalent to:



The ΞState is equivalent to ΔState along with the conditions that each primed state variable is equal to its un-primed counterpart — it means that the state does not change. This is used in specifications of operations that do not change the state.

For more details on Z see [DIL90].

Appendix C

Protection features of Unix File System in Z

In this Appendix we give a formal specification of the Unix file system protection mechanisms in Z.

C.1 Type Definitions

In this specifications we use the following types.

- NAME which is the set of all strings of characters
- PRIVLVLS is {priv, unpriv}
- INTEGER is the set of all positive integers
- RIGHT is the set {‘r’, ‘w’, ‘x’}
- CHAR is the set of all characters
- FILTYPE is {File, Dir} where File denotes that the object is a file and Dir denotes that the object is a directory.

- PATH is a set of sequence of names where each sequence denotes a path name.

C.2 The State Description

The state description of the unix file system consists of a set of state variables whose values give the information about the files and directories. These are

- Entities - The names of the users in the system.
- PrivStatus - Specifies whether a user a privileged or not.
- Pwd - It is the current directory of each user.
- InodeList - It is the list of Inodes [BAC86] which contain all the information about files.
- OwnInode - Gives the name of the owner of each Inode.
- ORtInode - Gives the rights of owner to the object pointed to by this Inode.
- GRtInode - Gives the rights of a user who belongs to the same group as the owner of the Inode.
- WRtInode - Gives the rights of user who is not the owner or does not belong to the group of the owner of the Inode.
- DCInode - If the object represented by an Inode is a directory, this gives the contents of that directory which is a list of names and Inode numbers.
- DatCInode - If the object represented by an Inode is a file, this gives the contents of that file which is a sequence of characters.
- TypInode - This gives the type of an Inode which can be either a file or a directory.
- NoLinks - This gives the number of references to an Inode in all the directories.
- Root - This is the Inode number of the root directory.
- GList - This is the list of all groups.

- EntGrps - This gives list of all groups that an entity belongs to.

The specification in Z of this state is as shown below.

$$
\begin{array}{|l}
\hline
Unix - File - System \\
\hline
Entities : \mathbb{P}\, NAME \\
GList : \mathbb{P}\, NAME \\
EntGrps : NAME \pfun \mathbb{P}\, NAME \\
PrivStatus : NAME \pfun PRIVLVLS \\
Pwd : NAME \pfun INTEGER \\
InodeList : \mathbb{P}\, INTEGER \\
OwnInode : INTEGER \pfun NAME \\
ORtInode : INTEGER \pfun \mathbb{P}\, RIGHT \\
GRtInode : INTEGER \pfun \mathbb{P}\, RIGHT \\
WRtInode : INTEGER \pfun \mathbb{P}\, RIGHT \\
DCInode : INTEGER \pfun \mathbb{P}(NAME \times INTEGER) \\
DatCInode : INTEGER \pfun \mathrm{seq}\, CHAR \\
TypInode : INTEGER \pfun FILTYPE \\
NoLinks : INTEGER \pfun INTEGER \\
Root : INTEGER \\
\hline
Entities \cap GList = \phi \\
InodeList = \mathrm{dom}\, ORtInode = \mathrm{dom}\, GRtInode \\
\qquad = \mathrm{dom}\, WRtInode = \mathrm{dom}\, TypInode \\
\qquad = \mathrm{dom}\, NoLinks = \mathrm{dom}\, OwnInode \\
\qquad = \mathrm{dom}\, DCInode \ \cup \ \mathrm{dom}\, DatCInode \\
\mathrm{dom}\, DCInode \cap \mathrm{dom}\, datCInode = \phi \\
\mathrm{ran}\, OwnInode \subseteq Entities \\
Root \in InodeList \\
\forall x : NAME \mid (x \in \mathrm{ran}\, EntGrps) \\
\quad \bullet\ (x \subseteq GList)) \\
\mathrm{dom}\, Pwd = \mathrm{dom}\, PrivStatus = Entities \\
\forall x : NAME;\ y : INTEGER \mid ((x, y) \in (\mathrm{ran}\, DCInode)) \\
\quad \bullet\ (y \in InodeList) \\
TypInode(Root) = Dir \\
\mathrm{dom}\, EntGrps = Entities \\
\forall x : \mathbb{P}\, NAME \mid (x \in \mathrm{ran}\, EntGrps) \\
\quad \bullet\ (x \subseteq GList) \\
\hline
\end{array}
$$

Here the body of the above schema gives the constraints on the various state variables in the system which are self explanatory.

C.3 Definition of Functions

Here we give the definitions of some functions that are used in the specification of operations which can change the protection state of the system.

The PIList function takes a path name, an entity name and a starting Inode number and gives a sequence of Inode numbers corresponding to each directory in the given path name. This sequence is obtained by traversing the path name in order and looking up the next name of the path in the Inode contents of the current directory. This can be specified as shown below:

$$
\begin{array}{|l}
PIList : PATH \times NAME \times INTEGER \rightarrow \mathrm{seq}\, INTEGER \\
\hline
\forall\, path : PATH;\ user : NAME;\ start : INTEGER; \\
\qquad\qquad SR : \mathrm{seq}\, INTEGER \\
\quad \mid PIList(path, user, start) = SR \\
\quad \bullet\ (\#path = \#SR) \land \\
\qquad (first(SR) = start) \land \\
\qquad (\forall\, x : INTEGER \mid (x \in \mathrm{ran}\, SR) \\
\qquad\quad \bullet\ (x \in InodeList) \land \\
\qquad\qquad ((\, TypInode(x) = Dir) \lor (x = last(SR)))) \land \\
\qquad (\forall\, i : 2..\#SR \\
\qquad\quad \bullet\ ((path(i), SR(i)) \in DirConInode(SR(i-1))))
\end{array}
$$

The ChkPerms function takes a path name, an entity name and a starting Inode number and tells whether the user has appropriate permissions to traverse the path or not.

This is decided by checking whether the user has read permissions to all the directories specified in the given path. The specification of this function in Z is as shown below.

$$
\begin{array}{|l}
ChkPerms : PATH \times NAME \times INTEGER \rightarrow BOOLEAN \\
\hline
\forall\, path : PATH;\ user : NAME;\ start : INTEGER; \\
\quad SR : \mathrm{seq}\, INTEGER \\
\quad \mid ((ChkPerms(path, user, start) = TRUE) \land \\
\quad\ (SR = PIList(path, user, start))) \\
\quad \bullet ((\#path = \#SR) \land \\
\quad\ (first(SR) = start) \land \\
\quad\ (\forall\, x : INTEGER \mid (x \in \mathrm{ran}\, SR) \\
\quad\quad \bullet (x \in InodeList) \land \\
\quad\quad\ ((TypInode(x) = Dir) \lor (x = last(SR)))) \\
\quad\ (\forall\, i : 2..\#SR \\
\quad\quad \bullet (((path(i), SR(i)) \in DirConInode(SR(i-1))) \land \\
\quad\quad\quad (((OwnInode(SR(i-1)) = user) \land \\
\quad\quad\quad\quad ('r' \in ORtInode(SR(i-1)))) \lor \\
\quad\quad\quad ((EntGrps(OwnInode(SR(i-1))) \cap \\
\quad\quad\quad\quad EntGrps(user) \neq \phi) \land \\
\quad\quad\quad\quad ('r' \in GRtInode(SR(i-1)))) \lor \\
\quad\quad\quad ('r' \in WRtInode(SR(i-1))))))) \\
\end{array}
$$

C.4 The State Changing Operations

A number of operations exist in Unix file system that either check the protection state of the system or change the protection state of the system. Here we give a few operations of each kind.

A common operation in Unix is the file copy. This operation takes as input two paths that are

- the source path including the name of the file to be copied and
- the destination path, which is a directory, where the copy has to be created.

It also takes as input a name that is the name of the copy. This operation checks the rights of the user for the source path and the destination path and makes a new copy of the file by creating a new Inode whose data contents are copied from the Inode of the source

file. The permissions of the new file are set so that the user who copies it will be the owner and gets the read, write and execute accesses to it. The formal specification of this operation is shown below:

$$
\begin{array}{|l}
\hline
\textit{CopyFile} \\
\Delta Unix - File - System \\
Invoker\# : NAME \\
src?, Dst? : PATH \\
NewName? : NAME \\
\hline
Invoker\# \in Entities \\
(first(src?) =' /') \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, ROOT) = TRUE \\
(not(first(src?) =' /')) \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, Pwd(Invoker\#)) = TRUE \\
TypInode(last(PIList(src?))) = File \\
(first(Dst?) =' /') \Rightarrow \\
\quad ChkPerms(Dst?, Invoker\#, ROOT) = TRUE \\
(not(first(Dst?) =' /')) \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, Pwd(Invoker\#)) = TRUE \\
TypInode(last(PIList(src?))) = Dir \\
\forall x : NAME;\ y : INTEGER \\
\quad \mid ((x, y) \in DirConInode(last(Dst?))) \\
\quad \bullet (x \neq NewName?) \\
\exists NewInode : INTEGER \\
\quad \bullet (NewInode \notin InodeList) \wedge \\
\quad (DirConInode' = DirConInode \oplus \\
\qquad (last(PIList(Dst?)), DirConInode(PIList(Dst?)) \cup \\
\qquad\quad \{(NewName?, NewInode)\})) \wedge \\
\quad (InodeList' = InodeList \cup \{NewInode\}) \wedge \\
\quad (TypInode' = TypInode \cup \{(NewInode, File)\}) \wedge \\
\quad (DatConInode' = DatConInode \cup \\
\qquad \{(NewInode, DatConInode(last(PIList(Src?))))\}) \wedge \\
\quad (NoLinks' = NoLinks \oplus \{(NewInode, 1)\}) \\
\quad (WRtInode' = WRtInode \cup \{(NewInode, \{'r'\})\}) \wedge \\
\quad (GRtInode' = GRtInode \cup \{(NewInode, \{'r'\})\}) \wedge \\
\quad (ORtInode' = ORtInode \cup \{(NewInode, \{'r', 'w', 'x'\})\}) \\
\hline
\end{array}
$$

Another operation in unix is the *link* operation that creates an entry in a directory for a given Inode instead of actually copying the contents of the Inode. The effect of this

operation is similar to the copy file operation but it differs in some important details. The formal specification of this operation is shown below:

$$
\begin{array}{|l}
\hline
\textit{MakeHardLink} \\
\Delta Unix - File - System \\
Invoker\# : NAME \\
src?, Dst? : PATH \\
NewName? : NAME \\
\hline
Invoker\# \in Entities \\
(first(src?) =' /') \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, ROOT) = TRUE \\
(not(first(src?) =' /')) \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, Pwd(Invoker\#)) = TRUE \\
(first(Dst?) =' /') \Rightarrow \\
\quad ChkPerms(Dst?, Invoker\#, ROOT) = TRUE \\
(not(first(Dst?) =' /')) \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, Pwd(Invoker\#)) = TRUE \\
TypInode(last(PIList(src?))) = Dir \\
\forall x : NAME;\ y : INTEGER \\
\quad \mid ((x, y) \in DirConInode(last(PIList(Dst?))) \\
\quad \bullet (x \neq NewName?) \\
DirConInode' = DirConInode \oplus \\
\quad (last(PIList(Dst?)), DirConInode(last(PIList(Dst?))) \cup \\
\qquad \{(NewName?, last(PIList(Src?))\})) \\
\hline
\end{array}
$$

An operation that changes the current working directory can be specified as shown below:

$$
\begin{array}{|l}
\hline
\textit{ChangeDir} \\
\Delta Unix - File - System \\
ndir? : PATH \\
Invoker\# : NAME \\
\hline
Invoker\# \in Entities \\
(first(ndir?) =' /') \Rightarrow \\
\quad (ChkPerms(ndir?, Invoker\#, Root) = TRUE \wedge \\
\quad Pwd' = last(PIList(ndir?, Invoker\#, Root))) \\
(not(first(ndir?) =' /')) \Rightarrow \\
\quad (ChkPerms(ndir?, Invoker\#, Pwd(Invoker\#)) = TRUE \wedge \\
\quad Pwd' = last(PIList(ndir?, Invoker\#, Pwd(Invoker\#)))) \\
\hline
\end{array}
$$

The operation that changes the protection state of the system is the *chmod* command. This changes the rights associated with an object (either file or a directory) for the owner, group and world under some conditions. This operation is specified as shown below:

$$
\begin{array}{|l}
\hline
ChangePermissions \\
\Delta Unix - File - System \\
ORts?, GRts?, WRts? : RIGHT \\
src? : PATH \\
Invoker\# : NAME \\
\hline
Invoker\# \in Entities \\
OwnInode(last(PIList(src?)) = Invoker\# \\
(first(src?) =' /') \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, ROOT) = TRUE \\
(not(first(src?) =' /')) \Rightarrow \\
\quad ChkPerms(src?, Invoker\#, Pwd(Invoker\#)) = TRUE \\
ORtInode' = ORtInode \oplus (last(PIList(src?)), ORts?) \\
GRtInode' = GRtInode \oplus (last(PIList(src?)), GRts?) \\
WRtInode' = WRtInode \oplus (last(PIList(src?)), WRts?) \\
\hline
\end{array}
$$

C.5 Conclusions

In this appendix we developed the formal specification of the protection mechanisms of Unix file system in Z. This demonstrates that Z can be used to develop the specification of protection mechanisms of large systems in a concise and unambiguous manner. These specifications can be used both for understanding the features of the system and also for proving the security properties of such systems.