Scenario-Based Access Control

by

G.S. Knight

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
January, 2000

Abstract

This work describes an access-control modeling scheme for secure systems that is based on the object interaction specifications used in contemporary object-oriented analysis and design methods. The scheme is primarily intended to model integrity and legitimate-use in commercial systems. The primary concern of these systems is to prevent fraud and errors. Access controls are usually based on hierarchical delegation of authority and separation of duty. Security policies and control mechanisms can be based on the tasks and business activities that are performed by the system. Object-oriented analysis and design techniques are commonly used to model systems using abstractions and interactions closely related to the actual tasks and business activities of the problem domain. This makes these techniques an attractive basis for access-control modeling. The proposed model makes extensive use of data that is already collected by commercial object-oriented analysis and design tools. The motivation is the productivity gains that may be realized through reducing the effort required in the maintenance of separate security and design models and in ensuring there is consistency between security models and other system models. In object-oriented modeling the description of a problem and its solution are in terms of interacting objects. Object-oriented models specify the kinds of objects that can exist in a system and the kinds of interactions that they can take part in. The models describe the possible interactions in terms of object scenarios. Each scenario has a limited number of ways in which it can be combined with other scenarios. This can be the basis for defining a security policy. The proposed scenario-based access-control model extends current object-oriented models to bring more rigour to the relationship between scenarios. The limited ways in which objects interact in these scenarios provide the basis of a technique for safety analysis. In the security models produced, the set of access authorizations held by system entities is inherently non-monotonic over system execution. A decidable safety analysis method is provided for instances of non-monotonic scenario-based security models. It is expected that for a broad class of useful systems the analysis is tractable.

ACKNOWLEDGEMENTS

I thank God for the succour and sustenance without which I would not have completed.

I thank Deanna my wife for her support and her reviews of my work, and I appreciate the additional load all this placed on her; her efforts were essential in allowing me to complete.

I thank Glenn MacEwen my supervisor and Terry Shepard of my committee for their careful reviews that unquestionably improved the quality of the dissertation.

Table of Contents

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

List of Figures

LIST OF FIGURES—*Continued*

List of Tables

List of Abbreviations

ACL	access control list
BIS	Battlefield Information System
BLP	the Bell and LaPadula model
BTRM	binary variant of the TRM model
CDI	constrained data item
DAC	discretionary access control
ESPM	Extended Schematic Protection Model
HRU	Harrison, Ruzzo and Ullman's protection matrix model
I&A	identification and authentication
ITU	International Telecommunications Union
MAC	mandatory access control
MLS	multi-level secure
MSC	message sequence chart
MSD	message sequence diagram
MTAM	monotonic variant of the TAM model
NMT	Non-monotonic Transform model
OO	object-oriented
OOA	object-oriented analysis
OOD	object-oriented design
ORCON	the originator controlled security policy
RBAC	role-based access control
RBAC96	a family of role-based models
$RBAC_i$	a model in RBAC96, where $i = 0..3$
SBAC	scenario-based access control

SDL	Specification Description Language
SPM	Schematic Protection Model
TAM	Typed Access Matrix model
TBA	task-based authorization
TBAC	Task-Based Authorization Control family of models
TBAC_i	a model in TBAC, where $i = 0..3$
TP	transformation procedure
TRM	Transformation Model
UML	Unified Modeling Language
UTRM	unary variant of the TRM model

Chapter 1

Introduction and Motivation

1.1 Introduction

One can view a commercial organization as a system that is required to maintain a certain state (or standard) of integrity. Organizational procedures and internal controls then have to ensure that the tasks carried out in the organization preserve such a state of integrity [San96]. To maintain such an integrity state there must be some assurance that users can only use their access to an information system (and through it access to corporate data/information assets) for legitimate purposes. Users should be limited to data accesses required to perform tasks for which they have authorization. The tasks should be statically defined and form the basis of a mandatory security policy for the system. Specific tasks should be authorized for an individual based on the duties of that person in fulfilling the corporation's business objectives. The authorized tasks should be sufficient for individuals to accomplish their duties but should not provide superfluous access to data. *I.e.*, the performance of a task must be a legitimate use of the information system.

The first section of this chapter defines the goals of this dissertation with respect to providing modeling support for the capture of the legitimate use properties of a system. The next section provides motivation for these goals and for scenario-based access control modeling. The last section provides an outline of the research presented in the remaining chapters of the thesis.

1.2 Goals

The development of scenario-based access control is driven by two main goals. The first goal is to provide a scheme that will provide efficient safety analysis for systems modeling legitimate use policies. This implies efficient analysis of non-monotonic systems. This follows because legitimate use policies that employ just-in-time availability of access control permissions are inherently non-monotonic. The reasons for this will become apparent later in the dissertation. The second goal is to provide a modeling scheme that complements contemporary software engineering modeling techniques. The objective is to leverage the information that is already being captured by such techniques and to provide security modeling as an extension to existing software engineering methods. This eliminates duplication of effort in security modeling and may serve to encourage the use of security modeling.

The scope of this research is directed toward achieving these goals and is reported on in the dissertation. This includes the definition of a modeling scheme for capturing legitimate-use policies based on just-in-time availability of access-control permissions. The primary abstraction in the modeling scheme is the scenario. A scenario is a formal description of a set of actions permitted by a group of objects. The modeling scheme is called *scenario-based access control* (SBAC). The research also includes a safety analysis scheme for scenario-based models. A security model design capture and analysis tool has been implemented based on the SBAC modeling and safety analysis schemes. The SBAC tool is used to develop example models that are representative of some interesting classes of information system applications. The analysis of these systems is tractable and the tool is used to gather some empirical data about the efficiency of the analysis algorithm. The results do not provide a proof of tractability for SBAC modeling in general, but it is expected that there is a large class of systems for which analysis based on the unfolding algorithm is tractable.

1.3 Motivation

The modeling scheme is intended to specify commercial and application-based security policies. The control of the access to information is usually divided into principal areas of concern: secrecy, integrity, availability, and legitimate-use. The modeling technique presented here is particularly appropriate for specifying integrity and legitimate-use policies. Clark and Wilson in [CW87], Moffet and Sloman in [MS88], and more recently Thomas and Sandhu's task-based access control in [TS94] have asserted that commercial security concerns should mirror an organization's internal control systems and work flows. These controls are usually based on hierarchical delegation of authority and separation of duty. The primary concern of commercial systems is usually to prevent fraud and errors. The problem domain statements of requirement for an organization's information systems tend to be task based. Therefore, role and task-based access control seems to be a promising direction of research and is motivation for modeling legitimate-use policies. "The fact that authorization is transient and dependent on organizational circumstances" [TS94] means that such policies tend to be non-monotonic in the generation of access rights as system execution evolves. That is, the entities in a system can both gain and lose access rights as the evolution of the system progresses.

There are modeling schemes in the literature that can be used to express systems with monotonic security policies that also have efficient safety analysis [San88, San89, San92]. There are also modeling schemes in the literature that can be used to express systems with non-monotonic security policies [HRU76, SS92, San92, SG94]. However, there has been less success in providing an efficient safety analysis for modeling schemes used to express systems with non-monotonic security policies. The ability to provide efficient safety analysis for such systems would significantly expand the classes of system for which access-control modeling and safety analysis can provide

assurance for security critical design.

Object-oriented analysis and design (OOA/OOD) techniques are commonly used to model systems using abstractions and interactions closely related to the actual problem domain. This makes these techniques an attractive basis for access-control modeling. The system specification can be captured in terms of high-level abstract tasks that meet an organization's business objective. This provides intuitive semantic content that takes advantage of natural human cognitive skills [Boo94]. Another motivation for the use of object-oriented (OO) modeling techniques is that they scale well over varying levels of abstraction. The same basic philosophy of object-oriented decomposition can be applied during system analysis, design, and implementation. Although OO models can be used at various levels of abstraction this does not imply model correspondence between the levels. Model correspondence between the levels is not straightforward and is itself a separate area of research. Correspondence notwithstanding, the ability to use the same basic techniques to model systems at various levels of abstraction is an attractive property. Current security models tend to work at a relatively low level of abstraction [TS94]. Security modeling based on OO techniques should allow modeling at higher levels of abstraction.

Security modeling using OO techniques also has the advantage of allowing security modeling to complement the system requirements and design modeling techniques used in contemporary software engineering practice. It is expected that much of the data required for security modeling is routinely captured by OOA/OOD methods. The data captured by contemporary modeling tools can be augmented to provide a security model compatible with current analysis and design methods. This should provide the basis for an access-control modeling and analysis capability that is compatible with the way system architects and designers do their work. This should result in productivity gains, as it reduces the effort required in the maintenance of separate security models and in ensuring there is consistency between security models

and other system specifications. A security-modeling tool that works with familiar system design tools also increases the likelihood that such modeling will be done.

This dissertation explores development of an access-control modeling scheme based on contemporary OO analysis and design methods to capture and provide efficient analysis for non-monotonic task-based models for secure systems.

1.4 Thesis Outline

Chapter 2 provides a literature review and the contextual information necessary to provide background for the area of research, to provide foundations and inspirations for the work, and to provide contrasting examples for comparison. Chapter 3 describes the modeling of scenarios of interacting objects. The models provide rigor to the relationships between types of scenarios and for the relationships between scenario instances. Chapter 4 defines the concepts of security policy and system. This chapter presents a scheme for safety analysis of scenario-based access-control models. Chapter 5 presents a series of worked examples. The examples presented are representative of some interesting classes of system for which analysis is tractable. Finally Chapter 6 provides some comparisons between SBAC and other security modeling schemes found in the literature. This chapter also provides a discussion that summarizes the contributions of the research and conclusions.

Chapter 2
Literature Review

2.1 Introduction

This chapter provides a review of existing literature which is relevant to the presentation of material in later chapters. A review of previous research in the modeling of secure systems serves to provide for the reader an understanding of the relevant issues surrounding access-control modeling schemes. The review outlines how previous authors have contributed to the topic area. The various modeling schemes illustrate the need to balance the ability of a modeling scheme to specify a wide variety of useful systems with the ability to perform an analysis of the security properties of those systems. The primary type of analysis considered is safety analysis, which is described later in the chapter. The review material also provides a brief description of some concepts associated with software engineering and object-oriented analysis and design. The scenario-based access-control modeling scheme proposed in this dissertation is built on and inspired by ideas arising from previous literature presented in this chapter.

The first section of the chapter presents a review of computer security issues and modeling schemes. The next section provides a review of object-oriented analysis and design issues. The end of that section provides an example of a system taken from the literature. The example has been reworked in a presentation format convenient for expressing scenario-based access control. The example will be used to provide an intuitive counterpoint to the formal presentation of the components of the modeling scheme in Chapter 3.

2.2 Review of Computer Security Issues

Security models can be used to transform security requirements into technical specifications and as a means to provide acceptance criteria for evaluating a system or system component. "Without such models, system developers are forced to apply ad hoc security related techniques throughout the design and implementation of the system. This approach inevitably leads to exploitable flaws, and makes the security assessments necessary for certification virtually impossible." [And72] This section will examine fundamental aspects of computer system security and present the relevant research issues and seminal models used to address the control of access to information and computing resources.

In general, secure systems will control, through use of specific security features, access to data such that only properly authorized individuals, or processes operating on their behalf, will have access to read, write, create, or delete data elements [Dep85]. Through the control of access to data the system endeavors to protect and preserve the information represented by the data. The set of rules and procedures governing this use of information is called the security policy. A system can be said to control the access of subjects (individuals, or processes operating on their behalf) to objects (information being held on the system, *e.g.* files). The control of access to information is usually divided into three areas of concern: *secrecy*, *integrity*, and *availability* [Gas88]. To this traditional set of areas of concern can be added *legitimate-use*.

Secrecy (confidentiality) concerns restricting the flow of information such that it does not become available or is not disclosed to some set of subjects. For example, the flow of information can be restricted such that a subject without an appropriate security classification, or need-to-know, is not permitted to read from a specific object (or have the information contained in that object written to it). Integrity issues concern the control of information such that information objects are not exposed to

accidental or malicious alteration. For example, objects can be controlled to prevent unauthorized subjects from modifying (writing, deleting) a specific object. Availability of information is a property of the flow of information that requires information contained in an object to be accessible to a subject, and requires that the flow does in fact occur when needed. A failure of this property, for example *denial of service*, would occur when a subject acts in a way that prevents or delays the valid flow of information between other subjects and objects on the system. Legitimate-use pertains to the prevention of the unauthorized use of system resources. For example, a subject may have access to an object but only within the context of some defined and authorized business task or workflow. Even though a subject has access to an object for a specific task, the access permission should not be able to be used by that subject for other purposes.

References to "information contained within an object" in the descriptions of the properties above can also be viewed as characterizing the services provided by an object. For instance, access to information can also be construed as access to a service (program, device, etc.). Given these definitions of security and the control of the flow of information, it should be understood that there are differing interpretations, and some concepts are less well-understood than others. The historic focus of the research community has been on confidentiality. This is because much of the research and system procurement activity was driven by the government/military. Integrity and availability were only generally defined in the literature and then usually in relation to a specific secure system implementation and its associated security policy. Recently there has been more emphasis placed on integrity as the interests of the research community have shifted to commercial information systems. This trend also drives the interest in legitimate-use issues. The modeling scheme proposed in this dissertation provides direct support for considering legitimate-use issues.

This section is organized in the following way. The first subsections present fun-

damental aspects of computer system security as a foundation for the discussions in following subsections. The next set of subsections present the safety problem as defined in the context of computer security, and seminal models are introduced to explore tractable analysis of the migration of access rights. Later subsections discuss the flow of information, access-control models conceived to restrict information flow in secure systems, and role-based access-control models.

2.2.1 Security Policy

The rules and procedures of a security policy are designed to meet the confidentiality, integrity, availability and legitimate-use requirements for the specific circumstances of users of the system. This can include regulating the processing, storage, distribution, and presentation of information [Com88]. The following paragraphs describe some common policies that will be referenced later in this dissertation.

Military/Government Security Policy. The military/government security policy [Gas88] is primarily concerned with confidentiality. The policy defines an ordered set of security *levels* (*e.g.* Unclassified $<$ Confidential $<$ Secret $<$ Top Secret) and a set of *categories* (*e.g.* Atomic, NATO, Alpha). A user has a *clearance* for a certain level and is also cleared for some number (possibly zero) of categories. Information has a *classification* which is composed of a security level and some designated number (possibly zero) of categories. A user is never allowed access to any information for which he is not cleared. Clearance implies the information's classification level $\leq$ (is dominated by) the user's clearance level, and the information's set of classification categories $\subseteq$ user's set of clearance categories.

Need-to-know. This policy is also associated with the military/government and can operate in parallel to, or separately from, the clearance-based policy described above. The policy specifies that a user is not allowed access, regardless of his classi-

fication, to any information unless he has a legitimate need to use the information. When operating in parallel with the military/government security policy, need-to-know access can only be extended to a subject which is already authorized access under the clearance-based policy.

ORCON. The originator controlled policy is again associated with the military/government. This policy is usually implemented in parallel with the standard clearance-based military/government security policy. As described in [San92] the creator of a document retains control over granting access to the information in the document. For example, if the creator of a document grants another user the authorization to read the document, that user cannot propagate the information in the document to a third user. It is prohibited for propagation to occur either directly by granting the third user the right to read the document, or indirectly by granting the third user the right to read a copy of the information in the document.

Separation of Roles. Security in the commercial environment often requires that more than one person be involved with some operation in order to reduce the probability of the occurrence of fraud. Such separation might involve a division of responsibilities. In some cases, the users perform separate functions. For example, the system administrator's role may be separated from the system security officer's role so there is no one person who can subvert the system. In other cases, the users cooperate in performing some task. In a business application a clerk and a manager may be required to cooperate to draft a cheque. This concept of a two-person rule can be generalized into n-person rules for synergistic authorization.

Chinese Wall. The Chinese wall policy has its origins in the business community [BN89]. In the financial services sector, a consultant must not divulge information pertaining to a client to a competitor. Thus, if a consultant is advising one company in a business sector (*e.g.* banking) he is not permitted to become privy to any knowledge about any other company in that sector, or to impart his insider knowl-

edge to any other company in that sector. The consultant's firm may represent a number of companies in a number of business sectors. The analyst may work with companies in sectors which are not in competition with each other (*e.g.* oil companies, insurance brokers, etc.), but only one in each sector. The choice of which companies the consultant will become involved with is unconstrained provided these rules are followed.

2.2.2 The Reference Monitor Concept

In a secure system there are a number of different system services that work together to provide security. Access controls are one of these services but others include *identification*, *authentication*, and *audit*. Identification and authentication (I&A) are closely related. The identification service ensures that a user is uniquely identified to the system. This unique identity can be used later to make decisions about what that user should be allowed to do. The authentication service is used to establish the validity of the claimed identity of a user. A user claiming a unique identity must prove that he is who he says he is by providing information known only to that user. Such information can be about something he knows, something he has, or something he is [Com88]. Examples of these types of proofs might be a password, a smart card token, and a biometric fingerprint scan respectively. Audit services are used to track events in the system and record what user identity initiated or is otherwise responsible for their occurrence. The logs produced are used to provide accountability of the users for their actions and to provide *a posteriori* evidence relating to breaches in system security.

Access control is closely related to the concept of the *reference monitor*. With the advent of multiuser systems, the reference monitor was introduced to control the sharing of resources. The reference monitor validates all references made by a program

in execution against those authorized for the subject by the system security policy [And72]. The references might be to programs, data, devices, etc. The effectiveness of the reference monitor depends on a trustworthy I&A mechanism to positively identify a user with a unique identity upon which access-control authorizations will be based. The actual mechanism (Reference Validation Mechanism) that implements a reference monitor may be a combination of hardware, software, and firmware and must have the following fundamental properties:

1. the Reference Validation Mechanism must be tamper proof,
2. the Reference Validation Mechanism must always be invoked, and
3. the Reference Validation Mechanism must be small enough to be subjected to analysis and tests to ensure that it is correct.

The requirement to subject the reference validation mechanism to analysis can be facilitated by the use of modeling techniques. In this way modeling can be used to support a reference validation mechanism. Models can be used to provide validation of security policy, to provide assurance of correctness, as a language of specification, and as a basis for refinement leading to design and implementation. The following sections will outline some of the major security modeling techniques used to express the control of access to information and computing resources.

2.2.3 The Access Matrix

One of the most basic abstractions used in dealing with access control is the access matrix [Lam71]. The model is defined as a state-machine. The state of the system is represented by an access matrix and a set of commands which operate on the matrix define the state transitions [Den82]. A state of the system is defined by a triple (S, O, A) where: S is a set of subjects, O a set of objects to be protected, and A is

a matrix with rows corresponding to subjects and columns corresponding to objects. A cell of the matrix $A[s, o]$ is the set of rights of subject s for object o. Depending on the model being used, subjects may also be objects, $S \subseteq O$, to support modeling of subject-to-subject communication. The rights specify the type of access allowed by a subject for a specific object, and can include the familiar *read*, *write*, *execute* and other privileges/properties such as *own*, *copy*, etc. as defined by the model.

The access matrix tends to be sparsely populated as each subject usually only has access to a restricted number of objects. Therefore, systems modeled in this way are seldom actually implemented as a matrix [SS94a]. A means of reducing the resource requirement is to associate a list of the filled cells in a column with its related object, or conversely a list of the filled cells in a row with its related subject. The first scheme is said to use *access-control lists* (ACLs), the second scheme is said to use lists of *capabilities*.

In ACL-based systems, each object in the system has an ACL and each entry in an ACL identifies a specific subject in the system and describes the type of access allowed by that subject. The list only contains entries for those subjects with some type of access. When access for an object is requested, the list is checked to verify that the requesting subject has the appropriate access right. The basic ACL scheme can be modified to include group names, or to limit the specification of subjects to simple classes such as owner, group, or world. The latter case allows the representation of ACLs as a regular concise set of bits, such as the familiar UNIX protection bits. ACLs have the disadvantage that if there is a need to find all the objects to which a subject has access, it is necessary to examine the ACL for every object in the system.

The converse approach is to associate a set of capabilities with each subject, each of which identifies a specific object in the system and describes the type of access which is permitted for that object. The subject can be thought to have a ticket or capability to access an object in a certain way. The disadvantage here is that to

find all the subjects who may have access to a specific object, all the subjects in the system must be checked for the capability.

A third way to handle access matrix information is to maintain a table (*authorization relation*) which contains a record for each permitted access. *i.e.* for each right present in each filled cell of the matrix. ACL-style or capability-style results can be obtained by sorting the table by object or subject as appropriate. This scheme is popular for relational database management systems which provide built-in support for record sorting and selection. An approach presented in [BSS+95] proposes a high-level language to capture the semantics of the access authorizations stored in tables without using the actual tables. The reference monitor will interpret these language-defined security attributes as access requests are made.

So far the model we have presented deals with the representation of a single state. This is suitable to describe a snapshot in time but not a dynamic system where access rights, or the number of subjects and objects, can vary with time. Some mechanism must be provided to allow state transitions. A set of commands can be provided in the model to allow operations on the access matrix. The commands may add (or remove) rows or columns and thereby subjects or objects to the system. They also control the entry (and deletion) of access rights into matrix cells. Such commands can be conditioned on other matrix entries. For example, it is common that if a subject owns an object (*i.e.* has the ownership right for that object), it is allowed to grant other rights for that object either to itself or to other subjects. *E.g.* an owner of a file, at his discretion, might grant the right to read that file to another user. An access-control policy which allows such commands at the discretion of a user is a form of a *Discretionary Access Control* (DAC) policy.

The primary purpose of multiuser systems is the sharing of resources. Objects contained in the system are valuable resources to be shared and manipulated (by authorized users). To unnecessarily limit sharing would be to defeat the purpose of

such systems. On the surface it might appear that an owner-based DAC policy is sufficient for most security concerns and would provide for adequate resource sharing. If the owner only passes rights to authorized trustworthy users what could be the harm? The harm lies in the possibility that although the user may be trustworthy, the processes (subjects) executing on his behalf may not be. This is the classic downfall of DAC. The problem is that the user is not generally aware of all of the behaviour of a process. A user may unwittingly execute a process which has been designed or altered to perform some malicious action in addition to the action desired by the user. Such a program is called a *Trojan horse*. A Trojan horse may be planted, given to, or copied by an unaware user. A Trojan horse embedded within a subject runs with all the rights of that subject. Once running, the Trojan horse may exploit the access-control mechanism to cause a transfer of information or a transfer of access rights in violation of the security policy. We will return to the problem of unauthorized transfer of information later. There are also problems associated with revocation of rights under DAC [San96]. The issue arises when considering the case where an access right is passed from the original owner to another user and then passed on again to further users. There is more than one interpretation for what it means to revoke such a right. That is, it is not clear whether the revocation of the right by the owner should pass beyond the user to which the right was originally given, and also revoke that right from all the other users to which it had subsequently been given by that user (a cascading revoke).

2.2.4 The Safety Problem

In the next few subsections we will look at the unauthorized transfer of rights. In a protection scheme such as the matrix model described in this subsection, an untrusted process may directly, or though a series of operations on the access matrix pass a right

to some other subject. In accomplishing this it may collaborate with other trusted and untrusted processes. This migration of rights may not be consistent with the system security policy. Given an initial system state we would like to characterize the *protection states* that are reachable in the system [San92]. This is the *safety problem*.

2.2.5 The HRU Model

Harrison, Ruzzo and Ullman in [HRU76] offer a formalized general model of protection systems based on the concept of an access matrix. In their work they address the issue of the migration of access rights by unreliable subjects. The safety problem in this context is to determine whether in a given situation a right can be passed to a subject that did not already have it. The HRU model has states, as described above, (S, O, A), with set S of current subjects, O of current objects ($S \subseteq O$), and matrix A. The model also includes:

R a finite set of generic rights (*e.g.* read, write, ownership, etc.)

C a finite set of commands of the form:

command $\alpha(X_1, X_2, \ldots, X_k)$
  if r_1 **in** (X_{s1}, X_{o1}) **and**
    r_2 **in** (X_{s2}, X_{o2}) **and**
    ...
    r_m **in** (X_{sm}, X_{om})
  then
    $op_1; op_2; \ldots; op_n$
end

or if m is zero,

command $\alpha(X_1, X_2, \ldots, X_k)$

$op_1; op_2; \ldots ; op_n$

end

Where α is a name, $X_1, \ldots, X_k$ are formal parameters, $r, r_1, \ldots, r_m$ are generic rights, and $s_1, \ldots, s_m, o_1, \ldots, o_m$ are integers between 1 and k. Each op_i is one of the following primitive operations which define a transition from some state (S, O, A) to another state (S', O', A'). When actual parameters (*e.g.* s, o) have been substituted for formal parameters (X_{si}, X_{oj}) they have the described effect on the access matrix.

enter r **into** (s, o)	$S' = S, O' = O, A'[s_i, o_j] = A[s_i, o_j]$ for all $(s_i, o_j) \neq (s, o)$, and $A'[s, o] = A[s, o] \cup r$, where $r \in R, s \in S, o \in O$
delete r **from** (s, o)	$S' = S, O' = O, A'[s_i, o_j] = A[s_i, o_j]$ for all $(s_i, o_j) \neq (s, o)$, and $A'[s, o] = A[s, o] - r$, where $r \in R, s \in S, o \in O$
create subject s'	$S' = S \cup s', O' = O \cup s'$, for all $(s, o) \in S \times O, A'[s, o] = A[s, o]$, for all $o \in O', A'[s', o] = \{\}$ and $A'[s', s'] = \{\}$, where $s' \notin O$
create object o'	$S' = S, O' = O \cup o'$, for all $(s, o) \in S \times O, A'[s, o] = A[s, o]$, and for all $s \in S', A'[s, o'] = \{\}$, where $o' \notin O$
destroy subject s'	$S' = S - s', O' = O - s'$, and for all $(s, o) \in S' \times O', A'[s, o] = A[s, o]$, where $s' \in S$
destroy object o'	$S' = S, O' = O - o'$, and for all $(s, o) \in S' \times O', A'[s, o] = A[s, o]$, where $o' \in O - S$

Each command will execute a sequence of primitive operations (the body of α) on the access matrix, conditioned on the presence of certain access rights in certain cells of the access matrix (the conditions of α). All conditions must be valid to invoke the

body. For example, the ownership-based DAC scheme described above might have a command $CONFER_r$:

command $CONFER_r(owner, friend, file)$
  if *own* **in** $(owner, file)$
  then enter r **into** $(friend, file)$
end

for which if the subject *owner* has the *own* right for object $file$, will grant the r right (read) to the subject $friend$.

Now consider the safety problem: to determine whether, in a given situation, a right r for an object can be passed to a subject that did not already have it. *I.e.,* r is said to *leak* to a subject that did not already have it. Subjects in the system which are known to be trustworthy and have the ability to grant r should not be considered as they make the system trivially unsafe. For example the owner of the object may be able to grant r to another subject. If the owner is trustworthy, he should not be considered in the safety analysis. What really needs to be known is this: if a subject is about to give away a right, could that action lead to further leakage of the right to untrusted subjects. The problem is therefore considered with trustworthy subjects remaining passive.

There may be a complex chain of operations involving a number of subjects and objects which may lead to such a leak. We say $Q \vdash_\alpha Q'$ if there exists a command α and actual parameters $a_1, \ldots, a_k$ for a protection system in state Q such that Q yields Q' under $\alpha(a_1, \ldots, a_k)$. $Q \vdash^* Q'$ indicates that there is a sequence of commands $\alpha, \beta, \ldots \omega$ such that $Q = Q_0 \vdash_\alpha Q_1 \vdash_\beta \ldots \vdash_\omega Q_n = Q'$.

A command $\alpha(X_1, \ldots, X_k)$ leaks a generic right r from state $Q = (S, O, P)$ if α, when run on Q, can execute a primitive operation which enters r into a cell of A which did not previously contain r. Given a particular protection system, an initial

configuration Q_0 is unsafe for r if there is a state Q and a command α such that $Q_0 \vdash^* Q$, and α leaks r from Q. State Q_0 is safe for r if it is not unsafe for r. By reduction to the Halting Problem it is proved in [HRU76] that it is undecidable whether a given configuration of a given protection system is safe for a given generic right. Undecidable in this case means that for any algorithm for deciding the safety of arbitrary protection systems, either some unsafe system is found safe or it cannot be established that a system is safe when in fact it is.

Given that the safety problem is in general undecidable, useful progress in the control of the migration of rights can only be made by:

1. dealing with more restricted systems for which specific tractable solutions are possible, or

2. building incomplete but sound modeling systems which will not misidentify a system as being safe, even though they cannot identify all safe systems.

Thus, the theme of research on the safety problem is to provide access-control models expressive enough to specify useful systems which will also allow tractable safety analysis of the system being modeled. For example, given that a computer system has limited resources, it might be natural to limit the number of subjects and objects that can be created to some finite number, or to prohibit the use of create operations altogether. The latter restriction, as noted in [HRU76], does yield decidable results, although the solution is PSPACE-complete and is therefore likely to be computationally intractable. Another example of a restriction is to limit the number of operations in HRU commands to a single operation [HRU76]. This also yields decidable results. A decision procedure for mono-operational HRU is NP-complete although a polynomial algorithm can be devised for a given protection problem.

2.2.6 Monotonic Protection Systems

Harrison and Ruzzo in [HR78] explore a class of systems which, as before, have primitive create and enter operations but do not have delete or destroy operations. Such a system is monotonic, in that it only increases in size and in entries in the access matrix. As it turns out, there is no real gain in the decidability of the safety of such systems. They prove that even if the number of conditions allowed in the operations of the system is reduced to a maximum of two, the solution to the safety problem is still undecidable. However, for the class of monotonic systems which are restricted to one condition (*i.e.* they may only check for one right in one cell of the access matrix), safety is decidable. The decision procedure in their proof has an NP complexity, but the authors propose a solution in linear time in the size of the access matrix which may yield a tractable solution for some cases.

Mono-conditional HRU and mono-operational HRU are of limited utility. In general, more than one condition and operation is needed in commands in order to express useful policies. For example, a mono-conditional system would not let a parent subject grant a right for an object it owns to a child subject. This would require testing for both an ownership right for the object and for an appropriate parent/child right. Mono-operational systems can be even more restrictive as it is not possible to both create an object and grant any right(s) associated with that object. A different approach is to restrict the transfer of rights based on a subject's possession of special rights which allow the propagation of rights. The following model is both bi-conditional and multi-operational (in contrast to the decidable cases of HRU above) but has an efficient safety analysis.

2.2.7 The Take-Grant Model

Jones, Lipton, and Snyder have proposed the Take-Grant model which is summarized in [Den82]. The authors prefer a graphical representation of system state with subjects and objects as nodes and a directed edge (x, y) representing the set of access rights x has for y. In this model, subjects are not objects. However, subjects can have access rights for each other and self referentially. There are two special rights *take* (t) and *grant* (g). $t \in (x, y)$ allows x to take any of y's rights, and if $g \in (x, y)$, x can share any of its rights with y. t and g themselves can be propagated in this way. Any transfer of a right in a system is constrained by the possession of t and g. Where r is a right, s is a subject, x and y are nodes, the primitive operations allowed in a system are :

s **take** r **for** y **from** x	for $t \in (s, x), r \in (x, y)$ adds r to (s, y)
s **grant** r **for** y **to** x	for $g \in (s, x), r \in (s, y)$ adds r to (x, y)
s **create** p **for new [subject | object]** x	for $p \subseteq$ graph adds new node x, where $(s, x) = p$
s **remove** r **for** x	removes r from (s, x)

Note that such systems are not strictly monotonic in that although a system can only increase in number of nodes, rights can be removed from the system. This model is expressive enough to solve certain protection problems. Subjects and objects in the model are naturally interpreted as possessing a set of rights for other subjects or objects. Since the rights are interpreted as being associated with the accessing entity rather than the accessed entity, this is a useful paradigm for use with capability-based systems. A shortcoming of the expressiveness of the model is that either all rights may be granted to another subject or none of them may be granted. This restriction is limiting in application to actual protection systems. The general safety question of

whether any other subject can obtain a right r given some initial state is solvable in $O(n^3)$ for an initial state with n nodes. Although this is a cubic complexity, only the initial size of the system affects complexity. The more specific question of whether a right r for a specific object can be transferred to some specific subject is solvable in linear time in the size of the initial state. All commands that would be possible in the general case (an HRU style matrix) are not possible here due to the take-grant restrictions on the propagation of access rights. This sacrifice in expressiveness by the modeling scheme yields a tractable analysis.

The concept of restricting the migration of rights to some set of links, identified by an assortment of control rights, has been investigated by other researchers. In the attempt to expand the useful class of systems which can be modeled, various refinements and modified schemes have been proposed, such as Minsky's Send-Receive Transport model [Min84]. Sandhu proposed in his Schematic Protection Model a unified way of dealing with such links. This has lead to a series of stronger protection models. The succession of these models will be discussed in the next subsections.

2.2.8 The Schematic Protection Model (SPM)

Previous subsections have described the inherent tension between the generality of a protection model and tractable analysis of the safety problem. The Schematic Protection Model [San88] proposes a different set of restrictions on the model to expand the class of systems which can be defined. SPM borrows from the Take-Grant model the concept of using tickets (capabilities) to control the dynamic migration of access rights. To this, the model adds static restrictions based on the protection type of subjects and objects. The use of typing to control security policy is the most important aspect of this model. In more general models, the unrestricted ability of a system to create subjects greatly increases the complexity of analysis. The SPM

model, therefore, places type-based restrictions on subject creation.

Again in this model subjects are not objects. Objects do not possess access rights; however, passive entities that possess rights, such as file system directories, can be modeled as a kind of subject. Subjects and objects are jointly referred to as entities. Each subject and object is created with a specific unchanging protection type which is a member of the set TS or TO respectively (let $T = TS \cup TO$). This strong typing is used to specify many of the major features of the system security policy. Decisions on entity creation and access right migration are based on type. Protection types are defined by the security administrator and cannot be changed during the operation of the system. They are therefore static. The ability of a specific subject to access some specific entity during system operation is represented by its possession of a ticket for that entity. For example, a subject A may have a ticket allowing it right r for some entity X. Such a ticket would be in the domain of A, $dom(A)$. The ticket specifies an entity and an access right and can be denoted as X/r. Tickets can be specified with or without a copy flag. The existence of the copy flag indicates that the ticket is copyable. The absence of the copy flag indicates that the ticket cannot be copied. A ticket with a copy flag can be denoted with a 'c' (*e.g.* X/rc). Each ticket has a type which is an element of $T \times R$. That is, the ticket type is specified by the type of the entity and the right denoted by the ticket.

The only operations which can operate on the protection state of a system are: *copy*, *demand*, and *create*. The *copy* operation requires three elements to be successful. For example, to copy a ticket X/r from subject B to subject A there must first be a ticket X/rc in the domain of subject B. Next there must be a *link* from B to A. The existence of a link is predicated on the existence of *control rights*, such as take/grant, in the domains of A and/or B. The model allows the flexibility to specify a number of different kinds of link based on different control rights and where they must exist. The following two examples define links for the Take-Grant model

described above and for the Send-Receive model [Min84]:

$$link(X,Y) \equiv Y/g \in dom(X) \vee X/t \in dom(Y)$$
$$link(X,Y) \equiv Y/s \in dom(X) \wedge X/r \in dom(Y)$$

The third element which needs to be satisfied is that the type of ticket being copied must be allowed by a filter defined for the link being used. Each kind of link has a *filter function*: $TS \times TS \rightarrow 2^{T \times R}$. *I.e.*, from $TS \times TS$ to the power set of ticket types. That is, for the type of subjects participating in a copy on some kind of link, only certain types of tickets can be transferred. Filter functions depend only on the static typing scheme and restrict the discretionary behaviour of subjects.

The *demand* operation allows a subject to be granted a right by asking for that right. To be successful, the subject's access to that right must be authorized by the *demand function*: $TS \rightarrow 2^{T \times R}$. The demand function maps a subject type to a set of ticket types. For a subject to gain a right by demand, the right must be one of those associated with that type of subject in that set. This implicit distribution of tickets by type is very useful for the distribution of control tickets used for setting up standard links.

The *create* operation is the only other operation allowed and is the most interesting. The *create* operation allows a subject to create a new subject or object in the system. A relation *can-create* (*cc*) and entity *creation-rules* (*cr*) control the creation of entities in the system. The *cc* relation is a subset of $TS \times T$. For example, subject A of type a can create entity B of type b iff $cc(a, b)$. To make efficient analysis possible the creation of entities can be further restricted to ensure that the graph resulting from the *cc* relation is acyclic. That is, there can be no cycles in the graph except self-loops, where a subject may be allowed to create other subject of its own type. The acyclic nature of the creation graph produces a hierarchical structure, which proves to be a useful property for the analysis of safety. There is a create rule for

every pair in *cc*. For the creation of an entity B of type b by a subject A of type a, the rule $cr(a, b)$ would specify what tickets for B are placed in $dom(A)$ and what tickets for A are placed in $dom(B)$. The model places restrictions on what rights may be specified. The rules must be *attenuating* for loops of the form $cr(a, a)$. This means that for loops, a subject which has been created must not have more rights that the subject that created it. For example, when subject A is creating subject A', $dom(A') \subseteq dom(A)$; also if a ticket for A' is placed in $dom(A)$ the corresponding ticket for A must be placed in $dom(A)$. The second condition ensures that the creator subject can not set up links for created subject that it cannot set up for itself. The attenuation property also proves useful in analysis. Together the acyclic and attenuating restrictions on subject creation ensure that the number of subjects that can be created in the system can be bounded for the purposes of the analysis.

In safety analysis we are concerned with the migration of rights as the system transitions from one state to another. The migration of rights by way of *demand* and *create* operations is dependent on the static typing scheme only and not on the distribution of tickets associated with any state. Of more interest to safety analysis is the migration of access rights associated with the *copy* operation. This depends on the initial set of entities and distribution of tickets, the initial state, and the evolution of state thereafter. Link predicates and filter functions control ticket copy. In a worst case scenario, all subjects cooperate in the migration of rights where possible.

A ticket can *flow* from one subject to another if there is *path* (a set of links) which connects the two subjects and the ticket is authorized by the filter function associated with each link on the path. In any state then, the capacity of all paths for some ticket is defined by the transitive closure of the links which are authorized for that ticket existing in that state.

Safety analysis for SPM is based on the concept of a *maximal state*. If there is some maximal state beyond which any state transitions provide no new migration of

tickets, then the analysis of this state provides a solution to the safety problem. As it turns out, such a maximal state does exist. Actually, there are many such states which are isomorphic for the purposes of the following analysis. One such maximal state can be produced by allowing each subject in the initial state to create a child subject for each pair in *cc* associated with it. These children are then recursively allowed to create their own children as authorized by *cc*. The acyclic restriction on *cc* ensures that this process terminates, provided loops of the type $cc(a, a)$ are only allowed a depth of one recursion. The intuition here is that the subjects in this creation hierarchy can act as the surrogates for any other subject of the same type that might be created by the parent. They will hold any right that might have been granted to another sibling in a different sequence of execution. Once the initial subjects have been fully *unfolded* by creation of their child surrogates, allowed *demand* operations can be executed and flows of the resulting system examined. The resulting flows will be the only flows possible for the initial set of entities and distribution of tickets under the specific typing scheme. Time complexity of this analysis is polynomial in the number of subjects in the initial state. Depending on the complexity of the *cc* graph, a worst case could yield complexity exponential in the number of subject types, TS. This is likely to be tractable in most cases, since *cc* is likely to be sparse. [VC94] demonstrates a similar result for an extension to SPM which allows ticket authorization to be qualified by an uninterpreted set of conditions characterized by the system environment.

Note that this model is monotonic. Entities can be created but not destroyed and subjects can gain tickets via the defined operations but they cannot be revoked. To allow this restriction on models to be more applicable to practical systems, Sandhu proposes the *restoration principle*. The restoration principle is applied in a safety analysis based on a worst case scenario. As long as a ticket which has been revoked can be restored or an entity which has been deleted can be replaced by an equivalent

entity then such an operation will not effect the outcome of the analysis. Therefore, any policy which allows revocation within the limits of the restoration principle can be overlooked in safety analysis.

The advantage of this model is the number of useful models/policies that can be represented in the decidable cases. This model has expressive power approaching that of more general models such as HRU and also provides a tractable solution to safety. The model subsumes Take-Grant type models. The introduction of typing allows the specification of static security policy schemes within a mechanism flexible enough to be broadly applicable. The SPM model can support policies which employ amplification (used by some models in implementing abstract data types), copy flags, n-person rules, and some forms of separation of duties [San88].

Further progress in this work follows two distinct but related tracks. The first direction reduces the work to a more basic model based on the fundamental notion of the transformation of access rights as the basis of access control. A model called *Transform*, and its derivatives are proposed in [San89]. The second direction recaptures the basic intuitive appeal of the HRU access matrix but with the advantages of strong typing to restrict the evolution of a system and provide a tractable solution to safety. This work introduces the *Typed Access Matrix* (TAM) model [San92].

2.2.9 Transform

[San89] proposes the principle of transformation of access rights as a unified way of specifying various access-control mechanisms. The aim of the model is to provide an abstraction of the basic behaviour of access-control mechanisms. The principle of transformation is that the propagation of access rights for an object by a subject should depend only on the subject's existing rights for the object. This contrasts with more general models like HRU and SPM where the propagation of rights can

also depend on the rights associated with other subjects in the system or involved in the transfer. Access control-mechanisms based on such a model lend themselves to implementation using ACLs. Sandhu claims that a protection model which has general applicability should be able to instantiate Transform. Transform has a simpler construction than SPM but maintains the protection type innovations of that model. In common with SPM, Transform has a set of rights, R, and disjoint sets of subject types, TS, and object types, TO. There is also a function $cc : TS \rightarrow 2^{TO}$, and create rules $cr : TS \times TO \rightarrow 2^R$ which specify the rights the creator obtains for the object created. Note that in this model subjects cannot be created and can only be introduced as part of the initial state. To provide for migration of access rights, instead of the operations *demand* and *copy*, Transform defines transformation functions *itrans* and *grant*. *itrans* is the internal, or self, transformation function, $TS \times TO \times 2^R \rightarrow 2^R$. The function allows a subject that possesses certain access rights for an object to obtain for itself additional access rights for that object. For example, a subject that possesses write access for an object may be able to transform that right into an append right for the object. *grant* is the external transformation function, $TS \times TS \times TO \times 2^R \rightarrow 2^R$. The function allows one subject that possesses certain access rights for an object to grant to another subject some specific access rights. For example, a subject with the *own* right for an object might grant the *read* right to another subject. A *grant* from a subject to itself may not be allowed for some systems. In fact, the security policy in some cases depends on the fact that self grant (as opposed to *itrans*) is not allowed.

The transformation of an access right can be either *attenuating* or *amplifying*. A transformation is attenuating if it does not grant a right that the granting subject does not itself possess. This agrees with the notion of attenuation discussed in the context of SPM. A transformation that is not attenuating is amplifying. It is readily apparent that *itrans* must be amplifying to be useful. Amplifying transformations

are very powerful. In fact they may be too powerful and lead to difficulties in safety analysis [Min78]. In [San89] it is shown that Transform with only attenuating *grants* is as powerful as general Transform. It is also shown that amplifying *itrans* can actually be implemented using only attenuating mechanisms as long as the required rights exist somewhere in the system.

Transform can be instantiated by HRU or SPM. In the case of HRU, transform cannot be instantiated within those cases known to be decidable as it is necessary to have multiple terms in the conditions of the model's commands. In the case of SPM, Transform can be instantiated within the efficiently decidable acyclic attenuating cases. The Transform model described so far is monotonic, in that access rights may be added to the system but there is no provision for removing a right. There are, however, some important security policies which require non-monotonicity.

2.2.10 Non-Monotonic Transform (NMT)

A policy might require non-monotonicity in that some rights for an object may be transfer only. For example, normally there is only one owner for an object. If transfer of ownership is to be allowed then the *own* right must be granted to the new owner and deleted from the old owner. The restoration principle does not apply to such revocations as the original owner cannot be regranted the *own* right as long as the new owner also retains ownership. Some separation-of-roles-based policies are also inherently non-monotonic. These policies require that a subject is allowed to access and modify an object during some step in a transaction; however, once the subject has played its role in the transaction, further access is denied.

The NMT model [SS92] defines the sets R, TS, and TO familiar from SPM and Transform. Although the original paper employs a procedural notation for the commands used to change the protection state (similar to those of HRU) the model can

be defined using the same set theoretic notation employed for SPM and Transform above. *cc* and *cr* are defined as for Transform. *itrans* is also defined as before except that, in addition to specifying a set of new rights to be added to the domain of the transforming subject, a set of rights to be deleted from that subject's domain is also specified. Similarly with *grant*, in addition to the set of new rights specified for the grantee subject, a set of rights is specified for deletion from the domain of the grantor subject. In addition to these transformations, new commands are defined for deletion of access rights. For each of the commands, the subject doing the revoking must possess the *own* right for the object. The commands allow the owner of an object to revoke from another subject any specified right or all rights to the object. Some government evaluation criteria [Dep85] specify a requirement that a subject may be denied any kind of access to an object. To facilitate this total denial of access requirement, there is also a version of the *revoke* command which will grant to any specified subject the null right, $\perp$, for an object. This special right implies that the object is totally inaccessible to the subject possessing that right.

The safety analysis for NMT in [SS92] shows that it can be subsumed by HRU with no subject creation. Such models still allow the unlimited creation of objects and Lipton and Snyder [LS78] have shown that such cases are decidable. The decision procedure of [LS78] for the general case has exponential complexity; however, the authors of [SS92] comment that the simplicity and strong typing of NMT may lead to a more efficient safety analysis. A more recent paper [AS94] provides a better safety analysis for one-representative cases of NMT. One-representative cases mean that only a single subject of any type need be considered in the analysis. These cases, in spite of the restrictions, have significant expressive power. The worst case safety analysis is still exponential in the number of subject types and rights. These values are constant; so, by itself this represents an improvement. As well, the analysis of actual model instances is often much easier than the worst case.

Transform and NMT have grown out of the concept of strong typing introduced by SPM by using the principle of transformation to provide a simpler, more abstract and expressive model. In related work, the strong typing of SPM is integrated with the intuitively straightforward matrix-based HRU model to produce a typed access matrix.

2.2.11 Typed Access Matrix (TAM)

Sandhu's TAM model [San92] grows out of two observations. The first is that the addition of strong typing should add strength to matrix-based protection models (specifically HRU) and provide a basis for decidable safety analysis. The second is that the ability to perform multiple parent entity creation makes HRU more expressive than SPM. One of the reasons HRU is more general and expressive than SPM is its capacity to have more than one parent entity (subject or object) responsible for the creation of a new entity. For example command $\alpha(S_1, S_2, O_1, O_2)$ might allow subject S_1 to create a new object O_2 such that subject S_2 is the new object's owner and O_1 is a file containing the initial data to be contained by the new object. S_1, S_2, and O_1 are all parents to the new object O_2. Multi-parent creation allows the specification of policies not possible in single parent creation models. For example, the ORCON policy seems to require multi-parent creation. The requirement to provide multi-parent creation led to the development of an Extended Schematic Protection Model (ESPM) [AS90]. With multi-parent creation, the expressive power of ESPM is formally equivalent to monotonic HRU, however, ESPM has the same positive safety results as SPM. TAM is a matrix-based model incorporating strong typing but retaining the expressiveness of ESPM.

The TAM modeling scheme combines these properties. TAM is constructed using a set of objects, OBJ, a set of subjects, SUB, ($SUB \subseteq OBJ$), and an access matrix.

This is the same as HRU. In addition, TAM also includes sets of types, TS and TO, ($TS \subseteq TO$) as defined for SPM. A state in TAM, (SUB, OBJ, t, AM), is the same as HRU with the addition of a *type function* $t : OBJ \rightarrow TO$, which defines the type of every object (including subjects) in the system state. TAM commands are similar in construction to HRU commands with the added requirement that every formal parameter specifies the type of object that must be specified as an actual parameter. Type mismatches between formal and actual parameters are not allowed. The primitive operations which form the body of the commands defined for a model instance are the same as before: *enter right*, *delete right*, *create subject*, *destroy subject*, *create object*, and *destroy object*. The *create subject* and *create object* operations now also specify the type of the entity to be created.

An interesting derivative of this modeling scheme is monotonic TAM (MTAM) which is the same as TAM but with the *delete*, *destroy subject*, and *destroy object* primitive operations omitted. MTAM has the properties of SPM's strong typing and the expressiveness of monotonic HRU. As well, strong typing allows another dimension of expressiveness that is not present in monotonic HRU. Monotonic HRU can be thought of as a special case of MTAM that has only two types: subject and object. Unfortunately, and not surprisingly, the safety of MTAM is no more decidable than for monotonic HRU. The nature of the MTAM model, however, allows the definition of useful restrictions that lead to decidable safety analysis with little sacrifice of expressive power. MTAM can be restricted to a ternary version, where all commands are limited to three parameters. As it turns out, ternary MTAM is equivalent in expressive power to MTAM. Analysis of safety is tractable for ternary MTAM provided creation of entities is acyclic. As with SPM, an MTAM scheme is acyclic if its creation graph is acyclic. Safety analysis is again based on a worst case and is very similar to the safety analysis of SPM. The commands defined for an instance of the model can be placed in a canonical form. The canonical form

separates commands that contain subject/object creation operations into two new, related commands, which have the same effect on protection state. The creation operations are placed into unconditional commands. The remaining, non-creating commands may be conditional. There is an unfolding of the initial state by applying the creation commands to the initial state entities where possible. This unfolded state contains a surrogate for any entity that might be created in the system. All non-creation commands are then performed until the state no longer changes. The resulting state is a maximal state and any system evolution from the initial state can be mapped onto this maximal state. This safety analysis for acyclic ternary MTAM is polynomial in time in the size of the initial access matrix. This surprising result is due to the restrictions imposed by strong typing and the local authorization of commands resulting from the restriction to three command parameters. The limit to three parameters means that the conditions authorizing a command are limited to examining only a small portion of the access matrix. Ternary commands lead to no loss in expressiveness as multi-parent creation is still possible, i.e. two parents and a child object can be specified. Unary and binary MTAM, although they yield a tractable safety analysis, do not allow multi-parent creation and are, therefore, weaker than ternary MTAM.

The safety analysis for acyclic MTAM without the ternary restriction does not have the same polynomial time result for complexity. Consider that, monoconditional monotonic HRU with no creation has NP-complete safety [HRU76, San92] and it can be seen that acyclic MTAM can subsume monoconditional monotonic HRU with no creation. Therefore, safety analysis for acyclic MTAM can be no better than NP-complete. A summary of these results indicates that safety analysis for acyclic MTAM is of complexity no better than NP-complete. Safety analysis for acyclic ternary MTAM is of polynomial complexity. Ternary MTAM is equivalent in expressive power to MTAM. This is not a claim that $NP = P$ because in these results nothing has

been asserted about what can be expressed in the acyclic cases of MTAM and ternary MTAM. The two models differ in the instances that can be modeled using acyclic creation.

2.2.12 Transformation Model (TRM)

The next model to be examined, TRM, fuses the desirable qualities of the two previous directions of development, which started with SPM and resulted in NMT and TAM. TRM generalizes NMT by adapting the principle of transformation to the more expressive structure of TAM. TRM also focuses on the capability to express policies that require non-monotonic changes to system state.

The construction of the TRM model [SG94] is much like TAM. The main difference is the application of the principle of transformation. Where TAM, like HRU and SPM, can base a change in access rights for an object on the current rights of a number of subjects or objects, TRM is restricted to examining only the rights for the object in question. The new model, like NMT, is non-monotonic and does not allow subject creation. So far TRM, as described, is a special case of TAM. However, TRM also allows for the testing for absence of rights, which is not allowed in the standard version of TAM.

The TRM model defines the sets R, TS, and TO ($TS \cap TO = \{\}$) as for SPM and NMT. States and the access matrix are as they are for TAM. A finite set of commands is specified for the creation and destruction of objects and the transformation of access rights. A command has one of the following formats:

command $create(S_1 : s_1, O : o)$
  create object O
  enter own **in** (S_1, O)
end

or

command $destroy(S_1 : s_1, O : o)$

if $own \in (S_1, O)$ **then**

destroy object O

end

or

command $\alpha(S_1 : s_1, S_2 : s_2, \ldots, S_k : s_k, O : o)$

if *predicate* **then**

$op_1; op_2; \ldots; op_n$

end.

$S_1, \ldots, S_k$ are the formal parameters corresponding to the subjects involved in the transformation. O is the object for which access rights are to be transformed. $s_1, \ldots, s_k$ and o are the types of the respective formal parameters which must match the types of the actual parameters used in invoking the command. The *predicate* is the condition of the command and is a propositional expression containing terms of the form: $r \in (S, O)$ or $r \notin (S, O)$, where S and O are formal parameters of the command. Each op_i is a primitive operation: enter r into (S, O), or delete r from (S, O). The following example of a command might form part of a transaction-based policy where a clerk can obtain the right to issue a cheque only if he does not have the right to prepare or approve the cheque.

command $issueCheque(S_1 : clerk, O : payCheque)$

if $prepare \notin (S_1, O) \wedge approve \notin (S_1, O)$ **then**

enter $issue$ **into** (S_1, O)

end

Two versions of TRM are specified in [SG94] which restrict the number of matrix cells that can be examined by a command. Unary TRM (UTRM) and binary TRM

(BTRM) only allow the predicate of a command to test one or two cells respectively. The predicate may still be composed of a number of terms checking for the presence or absence of rights in the cell(s), but the number of cells checked is restricted. For example, NMT is a restricted version of UTRM, as it checks only one cell and modifies, at most, two. It can be shown that BTRM can express any policy expressible in TRM. In fact, ternary BTRM (BTRM restricted to commands with three parameters) is strong enough to model any system which can be modeled in TRM. Although [SG94] claims that BTRM allows the expression of some policies that cannot be conveniently expressed by UTRM (and therefore NMT) a later paper [SS94b] proves BTRM and UTRM equivalent in expressive power, and therefore to TRM.

TRM is a matrix-based model with no subject creation. TRM restricted to check only for the presence of access rights can be subsumed by HRU with no subject creation. As stated before such cases are decidable with a decision procedure for the general case having exponential complexity. We can also note that the model is restricted by strong typing, is trivially acyclic, and we only really need to consider the UTRM cases. These factors may lead to a more efficient analysis. On the other hand, the fact that the model is non-monotonic and has the added intricacy of checking for the absence of access rights, as well as their presence, will likely add to the complexity of safety analysis. Presently, TRM has no efficient non-monotonic safety results. Tractable safety analysis for useful non-monotonic systems has been a difficult problem.

2.2.13 The Information Flow Problem

The motivation for looking at the issue of migration of access rights and safety was that although processes/subjects in the system are executing on behalf of users we trust, we don't necessarily trust the processes themselves. Unless a program is known

to be trustworthy, it may harbour a Trojan horse. A Trojan horse acting with the trusted user's privileges might propagate access rights in some insecure way. The safety analysis will tell us if the model instance we are examining will allow propagation of rights inconsistent with the security policy. If a system is 'safe' it may still not be secure. For example, when confidentiality is an issue, the overriding concern is the possible flow of information to a user who should not have it. If the system is to protect the information in a file, it does not really matter that an unauthorized user cannot obtain rights for the file if he can obtain the information in the file. Suppose that a Trojan horse, instead of trying to grant access rights for a file to an unauthorized user, copies the content of the file to another file which can be accessed by the unauthorized user. The access rights associated with the original file do not apply to the new file. In fact, the unauthorized user may own the new file. The propagation of information cannot adequately be controllled by a system where access decisions are solely at the discretion of the user (DAC).

One approach to the information flow problem is to apply a mandatory access control (MAC) scheme. Under a MAC scheme some access decisions are built into the system and cannot be overridden by the user even if it is the user's desire to do so. The strong typing schemes we have seen in the previous models are an example of MAC. The definition of protection types and the restrictions on entity creation and access right propagation based on type are built into the system. The user is constrained by the static policy decisions embodied in this typing scheme. The use of MAC in system modeling predates the introduction of protection types, and the formalization of the access matrix and the safety problem in [HRU76]. Initially, MAC controls were applied as an analogue to military/government security policy as an attempt to overcome the weaknesses associated with DAC.

Systems which support the military/government security policy are usually called multi-level secure (MLS) systems. MLS systems are probably the best known MAC-

based systems. Objects in such systems have a label corresponding to their classification permanently bound to them. Subjects have a clearance level associated with them. The control of information flow is provided by reference monitor based access control. The reference monitor of such systems has a MAC component which ensures that a subject is never allowed access to any information for which he is not cleared. Usually, the system will also have a DAC component which authorizes need-to-know accesses based on the discretion of the owner of the information. The next few subsections discuss modeling schemes designed to address the information flow problem. Some aproaches apply formal modeling to MAC while others try to prove information flow properties for a system. Information flow approaches have been applied in both the military/government sector and in the commercial sector.

2.2.14 Bell and LaPadula Model (BLP)

The first formalization of an MLS system was by Bell and LaPadula [BL73a, BL73b, Bel74]. The model explored the access-control properties required of a reference monitor to enforce military/government security policy. The model, BLP, is defined below:

- S a set of subjects
- O a set of objects
- C an ordered set of classifications
- K a set of categories
- L a set of security levels with a partial order relation $\leq$, where $L \subset C \times 2^K$
- A a set of rights { r, e, w, a}, where
  - r is read (read-only)
  - e is execute (no read, no write)

- $\underline{w}$ is write (read and write)
- $\underline{a}$ is append (write-only)

An element of L is a security level which is made up of a classification component and a category component, *e.g.* (Top Secret, NATO, ALPHA). The pairs of L form a partial order given the access rules for military/government security policy described previously. In the models previously described, the set of access rights was defined for a model instance, depending on the implementation being modeled. *I.e.*, rights are specifically defined in the context of an application being modeled. Here the set of rights, A, is defined for the model itself and is not changed for model instances. In this model system states are ordered triples from a set $V = (B \times M \times F)$, where

- B the set of possible sets of current accesses $2^{S \times O \times A}$, where $b \in B$ defines a current set of accesses
- M a set of access matrices, a matrix $AM \in M$ defines the current set of access rights each subject holds for each object
- F a set of security level vectors $F \subseteq L^S \times L^O \times L^S$, where $f \in F$ is a triple (f_S, f_O, f_C) where,
  - f_S subject security level function (clearance)
  - f_O object security level function (classification)
  - f_C current security level function
- R a set of possible requests to change the security state of the system
- D a set of possible responses to a request indicating an access decision result (*i.e. yes*, *no*, *error* (ambiguous request), ? (request not recognized))
- T the set of positive integers, $t \in T$ is a time index for request, decision, and state sequences:
  - X R^T, request sequences, where $x_i \in X$
  - Y D^T, decision sequences, where $y_i \in Y$

$Z \qquad V^T$, state sequences, where $z_i \in Z$

A tuple of a current access set b, $b \in B$, defines an access some subject is making to some object in the present state. In any particular state, a subject rarely has current access for all the objects which it may be authorized to access, only for those objects specific to the subject's current processing. The security level functions which compose F map subjects and objects to security levels. f_C designates a subject's current security level, such that for a subject s, $f_C(s) \leq f_S(s)$. A subject may currently not be accessing any objects at the upper limit of its security clearance level. f_C thus provides a security level based on a subject's current accesses. Elements of R can, for example, be requests to get or to release current access to an object, requests to give an access right to another subject (or rescind the right), create or destroy objects, etc. Requests can be used to attempt to modify the security state of the system. Depending on the current system state the request will yield a response and possibly a modified system state. An *action* of the system (r, d, v_2, v_1) describes a request r yielding a decision d and moving the system from state v_1 to v_2. $W \subset (R \times D \times V \times V)$ is a relation defining the possible actions of a system. Actions are the primitives for inductively defining an *appearance* of the system (a sequence of actions) and a system (a set of possible system appearances). A system is defined, $\sum(R, D, W, z_0) \subset X \times Y \times Z$, where an appearance of the system $(x, y, z) \in \sum(R, D, W, z_0)$ iff $(x_t, y_t, z_t, z_{t-1}) \in W$, for all $t \in T$, and z_0 is a specified initial state.

What remains is to define the characteristics of the system which must be maintained to ensure security. The three aspects of security which are considered are: the *simple security property* (ss-property), the **-property* (star-property), and the *discretionary security property* (ds-property). A state satisfies the ss-property if for every current access, $(S, O, \underline{x}) \in B$, which allows a subject to read data in an object (*i.e.* $\underline{x} = \underline{r}$ or $\underline{w}$), $f_O(O) \leq f_S(S)$. This means that for a subject to access an object such

that it is able to read data from the object, then the clearance of the subject must *dominate* the classification of the object.

The ss-property may seem to be enough. On the surface, this is a direct implementation of military/government security policy. This might be enough if all processes run by a user are as trustworthy as the user himself. As we have seen, real processes may not be and could write data to an object at a security level lower than the user. It is to counter this threat that the *-property is introduced. A state satisfies the *-property if for every current access, $b = (S, O, \underline{x})$, $b \in B$, which allows a subject to write data into an object (*i.e.* $\underline{x} = \underline{a}$ or $\underline{w}$), and for every current access $b' = (S', O', \underline{x}')$, $b' \in B$ which allows a subject to read data in an object (*i.e.* $\underline{x}' = \underline{r}$ or $\underline{w}$), $f_O(O') \leq f_O(O)$. This means that if a subject has simultaneous access to more than one object, the classification of all of the objects it can read data from must be dominated by the classification of all the objects it can write to. Note that $\underline{w}$ implies both read and write. Therefore, all objects to which a subject has $\underline{w}$ access must be at the same level, and the current security level of the subject, $f_C(S)$, must be at the level of those objects. Most subjects in a system are bound by the *-property. Those that are not are called trusted subjects. Usually such subjects must be guaranteed trustworthy (i.e. no errors, no Trojan horses, etc.) by some verification technique.

A state satisfies the ds-property if every current access, $b \in B$, is permitted by the current access matrix $AM \in M$ as we have seen for matrix-based models such as HRU.

A state $v \in V$ is a secure state iff v satisfies the ss-property, the *-property (trusted subjects excepted), and the ds-property. A state sequence $z \in Z$ is a secure sequence iff z_t is secure for each $t \in T$. An appearance $(x, y, z) \in \sum(R, D, W, z_0)$ is a secure appearance iff z is a secure sequence and a system $\sum(R, D, W, z_0)$ is a secure system iff every appearance $(x, y, z) \in \sum(R, D, W, z_0)$ is secure. A valuable property of secure systems in the BLP model is that they can be proved secure inductively. Preservation

of security from one state to the next guarantees total system security [BL75]. An action of a system, $(r, d, v_{i+1}, v_i) \in W$, transitions the system from one state to the next. An action $(r, d, (b_{i+1}, M_{i+1}, f_{i+1}), (b_i, M_i, f_i))$ is security preserving iff it adds no new elements to b_i that would violate the ss-property, the *-property (trusted subjects excepted), or the ds-property, and removes any elements of b_i that, following the state change, would violate these properties [McL87]. If the system begins in a secure state z_0 and all actions, $(r, d, v_{i+1}, v_i) \in W$, of the system are security preserving, then the system $\sum(R, D, W, z_0)$ is secure. This is the *Basic Security Theorem*. Construction of a secure system proceeds by defining operations, or rules, for changing the system protection state. The rules are proven to be security preserving with respect to any action they define. The system can then be proven secure inductively, given that it starts in a secure state. Bell and LaPadula provide an interpretation of the model for the Multics security kernel in [BL75].

BLP is probably the most widely known model for computer security. In 1985 the US government published the Trusted Computer System Evaluation Criteria (TCSEC) [Dep85], or *Orange Book* as it is commonly known, as a standard for the procurement of government information systems. Although the standard is supposed to accommodate a variety of models, its structure enshrines the concept of security through ACL-based DAC, and security-label-based MAC. Security labels are data classifications bound to and stored with the data elements/objects and are used as a basis for reference monitor access-control decisions. Although the Orange Book criteria for labelled MAC do not specify a modeling standard, it is heavily influenced by BLP. Later European standards and the Canadian Trusted Computer Product Evaluation Criteria [Com88] are broader in their consideration of integrity and availability issues but their confidentiality specifications are similar to, and compatible with, the TCSEC. These standards have significantly influenced the direction of secure system development. There are currently a number of secure products which

have been evaluated against the criteria; however, diminishing government/military budgets are beginning to move the focus of developers toward more generic solutions which combine government and commercial requirements [Ada95].

2.2.15 The Lattice Model

Denning, in [Den76], models security levels as a lattice structure consisting of a partial order of security levels and least upper and greatest lower bound operators. Most security models based on security classifications use a similar lattice structure to define the security level relation, including later descriptions of BLP-style MAC models. As we have seen, BLP models a run-time mechanism which enforces flow restrictions on the dissemination of information by use of a reference monitor. The lattice model is also applicable to other run-time models and also to compile-time certification mechanisms. The latter are useful in that they can provide assurance that a process is trusted at the component level (*i.e.* the process as a system component is trusted to behave in compliance with some security specification). Such processes are excluded from run-time access-control checks on the assumption that they can be trusted not to disseminate information in a manner inconsistent with the system security policy. In a real system, such processes are often necessary. For example, a useful system probably has mechanisms for trusted downgrade of data, multi-level mail handling, multi-level networking, etc. Trusted processes are required to handle data at more than one level and may be required to write to low-level objects without allowing the inadvertent flow of high-level information into those objects.

2.2.16 Lattice-based Integrity

Biba in [Bib77] proposes a model for integrity in information systems which is essentially the dual of the BLP model. The military/government security policy, which

drives BLP, is based on the control of flow of information for confidentiality reasons. Information is allowed to flow from lower secrecy levels to higher secrecy levels. Biba's model is based on the observation that information should not flow from low integrity objects to high integrity objects. To allow information to do so would compromise confidence in the high integrity object. A simple integrity property and integrity *-property are defined which are duals to the respective BLP properties. The simple integrity property allows a subject only to read objects at an integrity level which dominates the subject. The integrity *-property only allows a subject to write to objects which are dominated by the integrity level of each of the objects for which the subject has read access. The integrity lattice has been proposed as a MAC paradigm for commercial security on its own and also in conjunction with a BLP-style lattice for secrecy [Lip82]. Operating systems having MAC lattices for both secrecy and integrity have been produced commercially. In [San93], it has been shown that such a composite scheme can be modeled as the product of lattices, (the BLP lattice and an inversion of the Biba lattice) which is itself a lattice. [San93] also proposes a lattice solution to Chinese Wall security policies.

2.2.17 Information Flow Analysis

The lattice-based security models we have just been examining are useful in providing access control for the objects identified by the model. A significant weakness of these systems is that it is difficult to have a model granularity detailed enough to identify (and thereby control) all objects in the system and still provide efficient analysis. Processes are not restricted to using the legitimate communication channels provided for interprocess communication (*e.g.* files, messages, etc.). Legitimate channels can be identified as objects, and usually are controlled in accordance with the system security policy. In an actual implemented system, many observations a process may make do

not lend themselves to such control. The inclusion of every implementation specific entity in the system which can hold a bit of data will also unreasonably complicate the model. Any action by a process that is observable by another process is a possible communication channel. The flow of information by other than a normal channel is by a *covert channel*. A *covert storage channel* is any communication resulting from the ability of one process to observe another process modifying the state of the system. The information observed may be object attributes, object existence, or the state of shared resources. The observation may be direct, *e.g.* the appearance of a new file name in a directory (whether or not the observer has read access to the new file or not). The observation may also be indirect. For example, the fact that the use of a peripheral is denied because it is already in use by another process, provides a bit of information about that process. A *covert timing channel* results from communication by means of observing the effect another process may have on system performance, measured against some timing base such as a real-time clock [Gas88]. A Trojan horse can modulate a covert channel to leak information out of the host process.

A means to identify possible covert channels is *information-flow analysis* based on *information flow models*. Information flow models also stand by themselves as modeling techniques for secure systems in general, and some believe that the correct explication of security should be formulated in terms of information flow [McL90]. To this end, information flow models can be used to provide system specifications. Some models provide methods for the refinement of specifications to provide system design and implementation. Information flow analysis and models are based on detailed formal specification, rather than on an abstract state machine [Gas88]. This is because the variables that participate in covert channels are not necessarily represented in an abstract model. The basic form of such models is of state or trace-based specifications which specify what a subject can observe of the system. For example, a model might specify that information cannot flow to one user from a second user if the purging

of the second user's input from the system has no effect on the outputs the first can observe of the system, i.e. the second user does not interfere with the first. Another model might specify that no observations one user can make of the system reflect the actions of a second user of the system, and thus represent a flow of information from that user to the first, *i.e.* the first user is unable to deduce anything about the inputs of the second user. In theory, as a system evolves to implementation, proof that the specifications still hold for each new more detailed level of abstraction provides assurance that the security policy is being met.

A profile of research in information flow models for covert channel analysis and system specification includes the following work [Den76, FLR77, Den82, GM82, GM84, McC87, McC88, Jac88, McL90, GMP92, BC92, BCC94, BY95, Ros95]. As this dissertation is primarily concerned with access control, these models will not be explored in detail here.

2.2.18 Role-based Security (RBAC)

Role-based security models have currently become the object of more interest as the focus of security research moves more from government/military environments to the commercial environment. It has been recognized above that DAC-based security may be adequate for cooperative environments but is too weak in environments subject to malicious attack. Rigid classification-label based MAC environments as defined by Orange Book criteria and implemented in a number of operational systems are based on government/military policy for confidentiality of information. These mechanisms do not lend themselves well to commercial security requirements [MS88, SS94a]. Clark and Wilson in [CW87], Moffet and Sloman in [MS88], and Smith in [Smi93] have asserted that commercial security concerns should mirror an organization's internal control systems (as Orange Book criteria mirror internal government/military con-

fidentiality controls). Commercial control systems are usually based on hierarchical delegation of authority and separation of duty. The primary concern of commercial systems is usually to minimize fraud and errors. The policy must ensure that no user can create or modify data in such a way that assets or accounting records can be lost or corrupted [CW87]. These are primarily integrity issues.

Role-based access controls (RBAC) address commercial security requirements by focusing on how users interact with data. A role is a semantic construct around which access control policy is formulated [San98]. In an RBAC policy, users are assigned to roles and permissions/rights are assigned to roles. Role authorizations are granted to users, or groups of users, based on what activities they are allowed to perform on system data. This differs from government/military DAC and MAC controls which allow, or disallow, access without regard to the use that the subject is going to make of the data. Usually under RBAC a user is authorized to take on different roles at different times during his interaction with the system (a discretionary property). While in a specific role, the user is restricted to the data accesses and activities authorized for that role. A role should provide just enough permission for the user to perform the tasks associated with the role. This is the principle of *least privilege* (a mandatory policy). This differs from classic DAC 'user groups'. Such groups are primarily sets of users. A system usually allows discretionary assignment of rights to a group. Roles explicitly define a set of rights available to the role. Therefore the type of data available to a role is fixed in the policy scheme by the system administrator and is non-discretionary. As well, in the type of rights allocated by RBAC there is usually a greater degree of data abstraction. The rights defined for a system using RBAC imply more complex interfaces than the standard read, write, and execute of DAC user groups. The authorizations typically allow a subject to perform a specific action on a specific type of data items [SCFY94]. Thus, a *clerk* role may be authorized to *post* an entry to a *bank account* while a *secretary* role may be authorized to *edit* a

letter.

Clark and Wilson in [CW87] define a model which identifies certain objects as Constrained Data Items (CDIs). Access to these objects is provided solely through Transformation Procedures (TPs). A TP is a kind of well-formed transaction which will move a CDI from one valid state to another. TPs are defined to operate on a specific set of CDI types. Users are restricted to a certain set of TPs. The definition of, and access to, TPs is defined by the security administrator and is a static scheme. Although the paper does not explicitly define its model in terms of abstract data types and role-based access control, these concepts seem implicitly to be a natural context. The various typed access-control models we have examined provide support for such a model.

There have also been authors who have proposed models which support RBAC based on Orange-Book-style DAC and MAC (*e.g.* [Lee88]). These models tend to be awkward and [SCFY94] notes that awkward models can lead to awkward implementations and a model better suited to the implementation of RBAC policies can lead to easier implementation.

There are a number of RBAC models proposed to handle disparate user environments and policies. Many of these models can be unified under the RBAC96 family of access-control models proposed by [San98]. A specific role-based policy can then be matched to RBAC mechanisms defined by one of the models in the family. Four models are defined in RBAC96: $RBAC_0$, $RBAC_1$, $RBAC_2$, and $RBAC_3$.

$RBAC_0$ is the base model and includes a minimal set of features to allow a system to support RBAC. Included in these features is support for the concept of sessions. A user may have multiple sessions running simultaneously (*e.g.* in different windows). Each session may be assigned a different combination of the user's authorized roles.

$RBAC_1$ and $RBAC_2$ add to the features of $RBAC_0$. $RBAC_1$ adds support for hierarchical roles. The intuition behind role hierarchies is that roles form a partial

order. Roles higher up the hierarchy inherit all the permissions for authorized roles below them in the hierarchy.

$RBAC_2$ adds constraints, which impose restrictions on acceptable configurations of the RBAC models. The features introduced in $RBAC_1$ and $RBAC_2$ are independent, *i.e.* the features can be added orthogonally to each other. $RBAC_3$ consolidates the features of $RBAC_1$ and $RBAC_2$. The $RBAC_1$ model has the following components:

U, R, P, and S	sets of users, roles, permissions and sessions respectively
$PA \subseteq P \times R$	a many-to-many permission to role assignment relation
$UA \subseteq U \times R$	a many-to-many user to role assignment relation
$user : S \rightarrow U$	a function mapping each session s_i to the single user $user(s_i)$ (constant for the session's lifetime)
$RH \subseteq R \times R$	is a partial order on R called the role hierarchy or role dominance relation, also written as $\geq$ in infix notation
$roles : S \rightarrow 2^R$	a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r \mid (\exists r' \geq r)[(user(s_i), r') \in UA]\}$ (which can change with time) and session s_i has the permissions $\cup_{r \in roles(s_i)} \{p \mid (\exists r'' \leq r)[(p, r'') \in PA]\}$

Constraints are added to the basic model in $RBAC_2$ and $RBAC_3$ to enforce higher level organizational policy. For example two roles may be declared as being mutually disjoint, *i.e.* the same user can not be assigned both roles. This particular constraint would define a separation of duties. Constraints can apply to all aspects of the RBAC model.

2.2.19 Task-based Security (TBAC)

Recent work [TS94, TS97] has proposed a shift in the focus of security models toward the representation of *authorizations* as a higher level of abstraction for the security requirements of an application or business enterprise. As it has been presented so far, the usual approach to access-control modeling is subject-object based. The models defined which subjects had access to which objects, and what kind of access they had. For the most part, the intent of the access by the subject, or in what context the access is being made, has not been considered as part of the model. Once access is available the subject seems free to use that access for any purpose. Under these kinds of policy models it is difficult to model legitimate-use security properties.

Role-based access control provides an initial step toward the ability to capture legitimate-use policies. RBAC by its nature encourages the definition of fine-grained rights/permissions. This support for data abstraction lends itself to task-oriented permissions such as the *post* and *edit* rights examples in the RBAC subsection, Section 2.2.18. RBAC also supports least privilege through the use of sessions. By controlling which roles are active for the user sessions the user is provided with just those permissions required to accomplish the work needing to be done.

But the specification mechanisms available in RBAC are not able to model the order in which the permissions are to be used or how many times they should be permitted to be used. There is still the notion that once a permission becomes available to a subject, the subject can use that permission for any purpose, and as often as it desires. To restrict the use of a permission to legitimate purposes it is desirable to be able to specify when in the execution of some business task a permission should become available, what that permission can be used for, and how many times it can be used. *i.e.* provide a context for the legitimate use of that permission. The permissions are provided on a just-in-time basis as required by the

task at hand.

The work on Task-based Authorization (TBA) and Task-based Authorization Control (TBAC) presented in [TS94, TS97] proposes a framework for *active* security models and enforcement from the perspective of activities and tasks. Permissions are constantly monitored, and activated and deactivated in accordance with emerging context associated with the progress of the tasks being performed.

As noted in the introduction to the dissertation, internal controls of an enterprise, and therefore the information system supporting that enterprise, are normally designed to ensure that the tasks carried out in the enterprise preserve a certain standard of integrity. In the classic paper-based systems an *authorization* is required to proceed with a task. An authorization is often captured as a signature on an archival document and represents permission to proceed with the task as well as the acceptance of some liability by the authorizer for the execution of the task. The task may also involve the requirement for separate authorizations for its subtasks. An authorization results in the enabling of one or more activities and related permissions. Authorization management is central to TBAC models. The fundamental abstraction is an *authorization-step*. An authorization-step represents a primitive authorization processing step and is the analog of a single act of granting a signature in a paper-based system.

Part of the motivation for this work is to provide modeling techniques for more abstract representations of security requirements. These higher-level models are appropriate for capturing an organization's policy requirements that pertain to security, and the interfaces between the organization and the computer system.

In classical subject-object access control models the information associated with a permission can be thought of as an element of a cross product, $P \subseteq S \times O \times A$. S is the set of subjects, O is the set of objects, and A is the set of actions or access rights. Under TBAC, access control also involves task-based contextual information.

Two more sets are introduced. AS is the set of authorization-steps. U is a set of usage and validity counts. The members of U control usage, validity, and expiration characteristics that may be tracked at runtime. *e.g.* how many times a permission is used. A permission under TBAC now becomes an element of $P \subseteq S \times O \times A \times U \times AS$. For example, a permission specifies a certain kind of access to a specific object by a specific subject, as was the case for matrix-based schemes (the A, O, and S components). The permission also specifies that an access can only be made in the context of a specific authorization-step and perhaps that the access can only be made n times (the AS and U components).

Permissions are associated with exactly one instance of an authorization-step. An instance of an authorization-step is associated with exactly one instance of a task. Each authorization-step maintains a protection state that is the set of permissions currently valid for the authorization-step. The members of the set will change with time as the authorization-step is processed. For each kind of authorization-step there is defined a *trustee-set*. The *trustee-set* represents the individuals/entities that are permitted to invoke and grant an authorization-step. For every instance of an authorization-step there is a single trustee from this set that invokes and grants the authorization-step.

A family of models for TBAC is proposed which is similar in spirit to the family of models proposed by RBAC96. Four models are defined: TBAC_0, TBAC_1, TBAC_2, and TBAC_3. TBAC_0 is the base model and includes a minimal set of features to allow a system to support TBAC. The TBAC_1 model adds support for composite authorizations. The TBAC_2 model adds support for constraints which impose restrictions on acceptable configurations of the TBAC models. As with RBAC96 the features introduced in TBAC_1 and TBAC_2 are independent, *i.e.* the features can be added orthogonally to each other. TBAC_3 consolidates the features of TBAC_1 and TBAC_2.

The TBAC_0 model defines the components that make up every authorization-

step, the life-cycle of an authorization-step, and dependencies that are used to model authorization policies.

Every authorization-step has to specify the following attributes:

Step-name	This is the name of the authorization-step.
Processing-state	The current processing state indicates how far the authorization-step has progressed in its life-cycle (discussed below).
Protection-state	The protection-state defines all potential active permissions that can be checked-in by the authorization-step. The current value of the protection-state, at any given time, gives a snapshot of the active permissions at the time. Associated with every permission is a validity-and-usage specification. The validity-and-usage specification specifies the validity and usage aspects of the permissions associated with an authorization-step. It will thus specify how the usage of the permissions will relate to the authorization remaining valid (or becoming invalid).
Trustee-set	This contains relevant information about the set of trustees that can potentially grant/invoke the authorization-step, such as their user identities and roles.
Executor-trustee	This records the member of the trustee-set that eventually grants the authorization-step.
Task-handle	This stores relevant information such as the task and the event identifiers of the task from which the

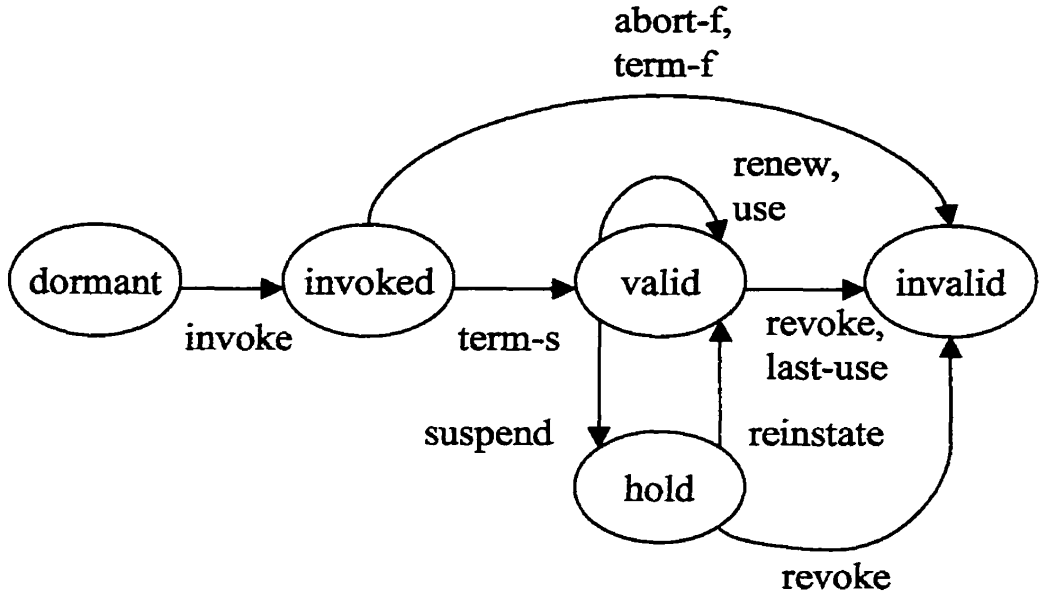

FIGURE 2.1. Basic processing states for an authorization-step

authorization-step is invoked.

An authorization is not static under TBAC. Each authorization-step has a life-cycle associated with it. An authorization step moves through a series of processing states during its lifetime. Figure 2.1 illustrates a simplified set of processing states. An authorization step is dormant (or non-existent) when it has not been invoked (requested) by any task. Once invoked an authorization-step begins to be processed by moving to the invoked state. If invocation completes successfully the authorization-step moves to the valid state. If the invocation fails (*e.g.* some criteria for authorization are not satisfied) the authorization-step moves to the invalid state. While in the valid state the authorization-step and its permissions may be used as specified by the validity-and-usage specifications in the protection-state. At some point the authorization-step will reach the end of its lifetime and enter the invalid state. It is also permitted that a valid authorization-step be put on hold temporarily. While on

hold the permissions associated with the authorization-step are inactive and cannot be used to provide access to an object.

The authorization-steps do not stand alone in a system specification. They are related to and depend on each other in order to fulfill higher level security policy requirements. There are existential, temporal and concurrency dependencies defined for the model. In the following definitions let A1 and A2 be authorization-steps and *state1* and *state2* be some processing state for these two authorization-steps respectively.

$A1^{state1} \rightarrow A2^{state2}$	If A1 transitions into *state1*, then A2 must transition into *state2*.
$A1^{state1} < A2^{state2}$	If both A1 and A2 transition into states *state1* and *state2* respectively, then A1's transition must occur before A2's
$A1^{state1} \# A2^{state2}$	A1 cannot be in *state1* concurrently when A2 is in *state2*.
$A1^{state1} \vert\vert\vert A2^{state2}$	A1 must concurrently be in *state1* when A2 is in *state2*.

$TBAC_1$ adds support for *composite authorizations*. A composite authorization consists of a set of component authorization-steps. The component authorization-steps are related to each other via dependencies. Component authorization-steps are visible only within the scope of their containing authorization-step.

$TBAC_2$ includes static and dynamic constraints. Static constraints are specified for a kind of authorization-step and all instances of that kind of authorization-step must meet the constraint. Dynamic constraints apply to an instance of an authorization-step and can be evaluated only as the authorization-step is processed.

2.3 Review of Object-oriented Analysis and Design Issues

2.3.1 Information Captured by Current OO Methods

The object-oriented paradigm is based on a logical view of a system as a set of cooperating objects (a more general use of the term object than the subject-object relationship of the previous section). The objects in the system are vehicles for information hiding [Par72] and each encapsulates some information. That information can be a data structure, device, algorithm, etc. Access to the object is provided only via a well-defined interface. Usually the interface is defined as a set of methods that can be used to manipulate the object. The interface methods can be thought of as operations that can be invoked on the object or alternatively as messages that an object can receive. The messages may carry information to the receiving object via message parameters and can provide information back to the sending object by way of return parameters. Messages can be used to alter the state of an object in some well-defined way, or to provide some information about the state of an object. Objects can only interact via message passing and an object only responds to the messages defined for its interface. The net result is that access to an object's secret is controlled. In object-oriented analysis and design the description of a problem and its solution are entirely in terms of objects passing messages. Object-oriented models specify the kinds of objects which exist in a system, the kinds of messages which make up the object interfaces, and how objects can be combined to cooperate in message exchange scenarios that solve some portion of the larger problem. The problem specification can be captured in terms of these scenarios.

Current object-oriented analysis (OOA) and design (OOD) methods such as Booch [Boo94], OMT [R+91], and UML [RJB98] provide notations and procedures for specifying object-oriented models. There are also automated tools associated with most OOA/OOD methods to facilitate model building. The models produced usually have

a view that describes the types or kinds of objects that may be instantiated in the system (*i.e.* classes in a class diagram [RJB98]). This view also describes how the types of objects are related to each other, *e.g.* containment, sub-typing, interface use. Other views in the model describe how objects interact (*i.e.* what messages are exchanged in a message sequence diagram or an object interaction diagram [RJB98]) in a given scenario. The information being captured by these modeling methods is similar to the components of typed access-control models. If the specification and modeling of the security aspects of a system are to become routine and efficient then they must complement and extend contemporary practices in system analysis and design. It seems likely that much of the data needed to model security (concerning the classes of objects, their components, the relationships between interacting objects, and the kinds of messages they can exchange) is already routinely captured by contemporary OOA/OOD methods.

There is usually a limited number of different kinds of scenarios in an object-oriented model and each kind of scenario has a limited number of ways in which it can be combined with other scenarios. This is because human beings design the various scenarios and the ways the scenarios are to be combined. They must be able to cope with the complexity of system design. Object-oriented decomposition and object interaction scenarios are organized with the purpose of restricting the complexity of system design. Essentially, it is the scenario-to-scenario interaction which specifies, and limits, the behaviour of the system. This can be the basis for defining a security policy. Scenario-scenario relationships are not always captured well in current object-oriented modeling methods and therefore it is difficult to tell with assurance what object will have access to any other object in the system as execution unfolds. The scenario-based access-control model proposed extends the current object-oriented models to bring more rigor to the relationship between scenarios. The limited ways in which objects interact in these scenarios are used later to form the

basis of a technique for safety analysis.

A strength of OO modeling is that it allows specification of the system in terms of the entities and interactions of the problem domain [Boo94]. This makes OO modeling in general, and the scenario-based access-control modeling technique specifically, a suitable tool for representing problem domain tasks. Problem domain tasks are specified in terms of scenarios. Since the scenarios drive the security policy, the policy is task-based.

2.3.2 Message Sequence Charts (MSCs)

Object-oriented analysis and design methods have borrowed the notation of message sequence charts (MSCs) from the telecommunications protocol design community. Message sequence charts are used, often in combination with the Specification Description Language (SDL), in the specification of system protocol requirements, and for testing [Mau96]. The International Telecommunications Union (ITU) has standardized both SDL and MSCs [Int88, Int94]. This subsection provides a brief introduction to message sequence charts. MSCs are related to the scenarios used in SBAC modeling so some background is presented and some of the issues associated with MSCs are addressed here.

MSCs have both a textual and graphical representations. The graphical representation is most commonly used. A system is represented by a set of communicating processes. Processes are represented by vertical lines. Signals sent between processes are represented by arrows connecting the vertical process lines. Each process's vertical line is a time-line. *I.e.,* send and receive events for signals are ordered temporally from the top of the line to the bottom. Communication can be synchronous or asynchronous as defined for the system. Figure 2.2 presents an example of a simple MSC. The first process sends a signal of type a to the second process, which after receiving

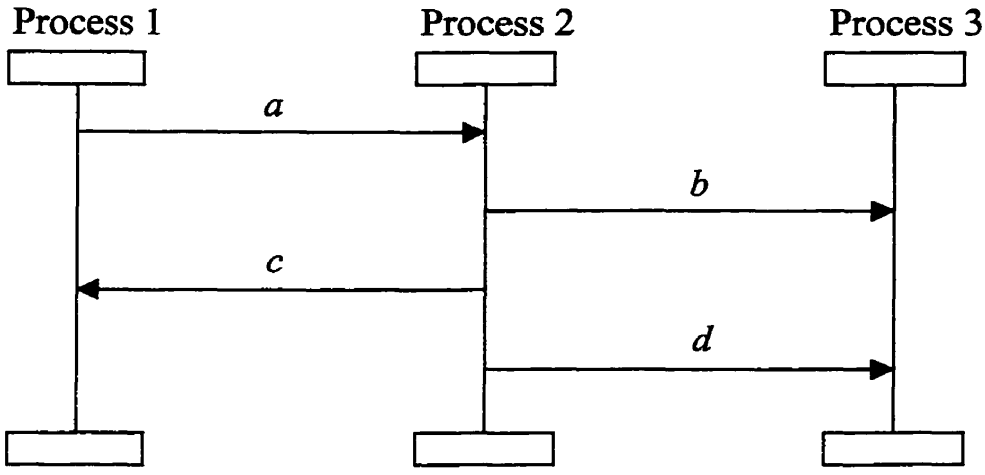

FIGURE 2.2. A Simple MSC

it sends a signal of type b to the third process, a signal of type c is then sent back to the first process, and finally a signal of type d is sent to the third process.

Multiple MSCs may be used to specify a system. In [LL94] MSCs are 'joined' by the use of *conditions*. A condition is represented as an elongated symbol spanning the process axes. The conditions have labels and are constrained to be placed as the first event or last event on the process time-lines. The system is defined to behave as though MSCs with identically-labeled conditions are joined at the condition. A MSC may be joined to itself at these conditions to create a non-terminating loop. Conditions may also be used to specify non-determined behaviour, such as conditional branching or conditional loops. This occurs if a terminal condition of one MSC shares a label with the initial condition of two or more MSCs. The time-lines for the processes can branch at that point taking one of the possible paths represented by the joined MSCs.

The meanings of MSCs have been formalized by Mauw [Mau96] and by Ladkin and Leue [LL94, LL95b]. The definition of the semantics of MSCs was addressed after the original publication of the standards for MSCs [Int88, Int94]. The original MSC

specification included only a semi-formal description of the meaning of the charts. The syntactic features of MSCs raise some issues concerning the interpretation of the charts. [LL95a] presents some specific issues with respect to the semantics of MSCs which have some significance in the context of scenario-based access-control modeling since there are strong parallels between the two. The issues are introduced here and will be revisited in the discussion presented in Chapter 6.

One of the concerns is whether systems represented by MSCs have some finite set of global states with respect to message passing behaviour. Even given a finite number of control states for the participating processes, there may be an unbounded number of asynchronous messages 'in the system' (*i.e.* sent but not received). This could be an argument for a non-finite set of global states. It is demonstrated for MSCs that the set of global states is in fact finite [LL95a]. Transitions between states are effected by atomic message-passing actions, which can be used to define a state transition function. This is a useful property for analysis of the systems being specified. This issue needs to be addressed in the context of scenario-based access control as an unbounded number of current permissions for object interactions is possible.

For general MSCs the use of conditions to join MSCs introduces non-determinism. At a condition, individual processes must continue with behaviour as specified by one of the joined MSCs. In some cases this requires a choice of behaviours by a process that does not depend solely on its own process state. Such *non-local choices* require either un-bounded history variables to keep track of control choices (non-finite-state control) or MSCs which lead to non-local choices must be considered as ill-formed. SBAC proposes a type of joining mechanism on message sequences. The issue of non-local choices will be considered in the context of this thesis.

In the brief description of MSCs above there was no restriction specified that would preclude the crossing of message arrows. Crossings can lead to messages being

received in a different order than that in which they were sent (an 'overtaking' of one message by another). This is allowed in the MSC specification. In some cases it may not be possible to detect the possible occurrence of such an overtaking by using the process specifications alone. If the possibility of a crossing is not an aspect of the process specifications then it must be a property of the system environment. *I.e.*, a property of the environment must account for the crossover but is unspecified in the system specification. Environmental properties are not usually explicit in the system specifications. The existence of such undefined system properties is not desirable in a specification language.

The last issue addressed in [LL95a] relates to the completeness of the information available in MSCs to specify *liveness* properties. The authors argue that liveness properties are difficult to specify with MSCs alone and such properties are better specified in many cases by temporal logic formulae provided in addition to the MSCs.

2.3.3 Document Release Example

This sub-section provides an example using a set of scenarios to illustrate ways in which objects can interact to provide the solution to a problem. The example is a non-monotonic security policy (*cf.* Sandhu, [SG94]). A company scientist prepares a paper for publication. Before the scientist is allowed to publish his work he must clear it with a patent officer. The patent officer can authorize the paper for publication or she can return it to the scientist for revision. The scientist is initially able to modify the content of the paper but loses that right while the paper is under review. The scientist is also not able to alter the content of a paper authorized for publication by the patent officer. Figures 2.3 to 2.10 provide a simple set of diagrams that specify the object interactions that might be allowed in such a system. The diagrams are presented as a complete set here so that they can be understood in context with each

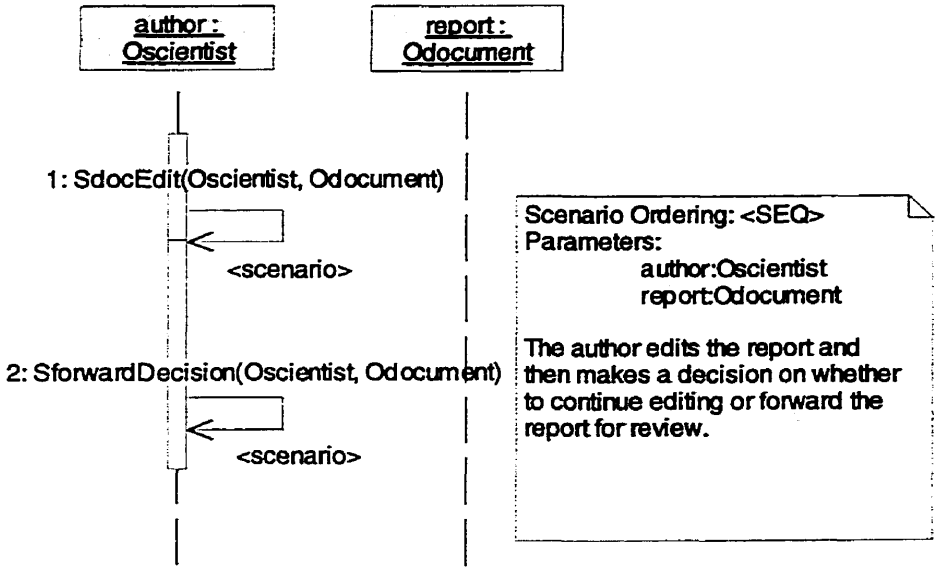

FIGURE 2.3. Scenario Type Sinitail

other. The diagrams are used in examples later in this dissertation and a deeper understanding of their meaning will come as a result of referring to them in the context of those examples.

The diagrams are UML [RJB98] message sequence diagrams. The diagrams have been produced using Rational Rose [Rat98], a popular industrial OOA/OOD tool. The tool captures basic information about a model, *e.g.* object types, object identifiers, message types, message parameter types, and the ordering of messages.

Message sequence diagrams (MSDs) are a standard notation for specifying object interactions in a system. Message sequence diagrams are a restricted form of message sequence chart (Section 2.3.2). Information from these diagrams can be combined with additional information to describe how scenarios combine and interact.

The vertical lines with boxes at the top represent objects. The boxes are labelled with an object identifier and an object type. The dashed line extending below the box is the life-line for the object. The arrows between object life-lines are object in-

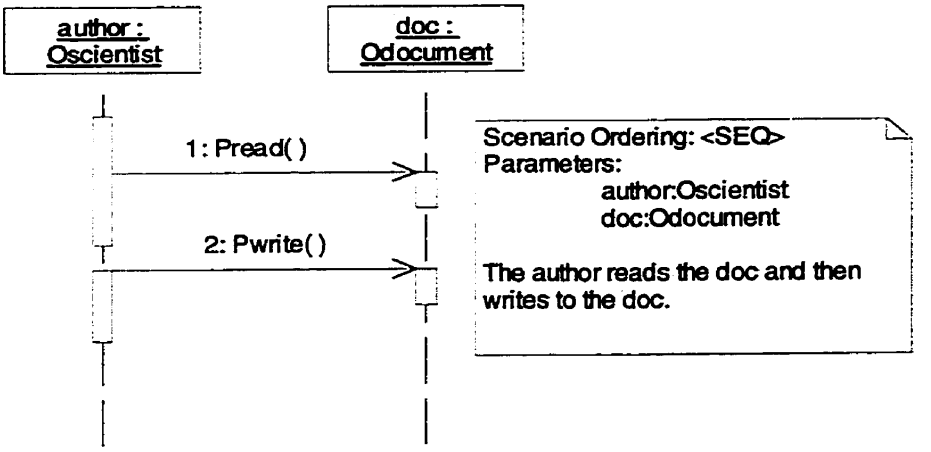

FIGURE 2.4. Scenario Type SdocEdit

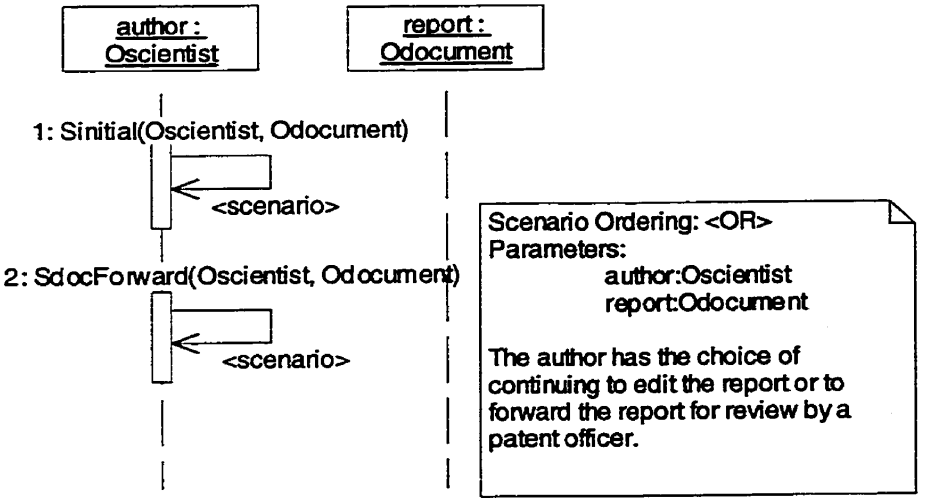

FIGURE 2.5. Scenario Type SforwardDecision

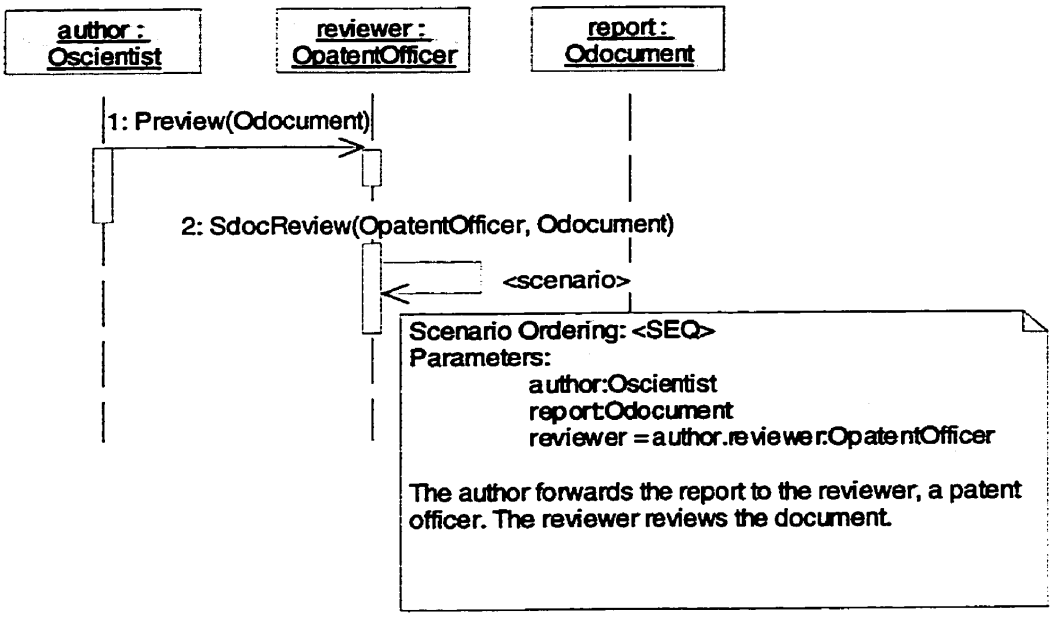

FIGURE 2.6. Scenario Type SdocForward

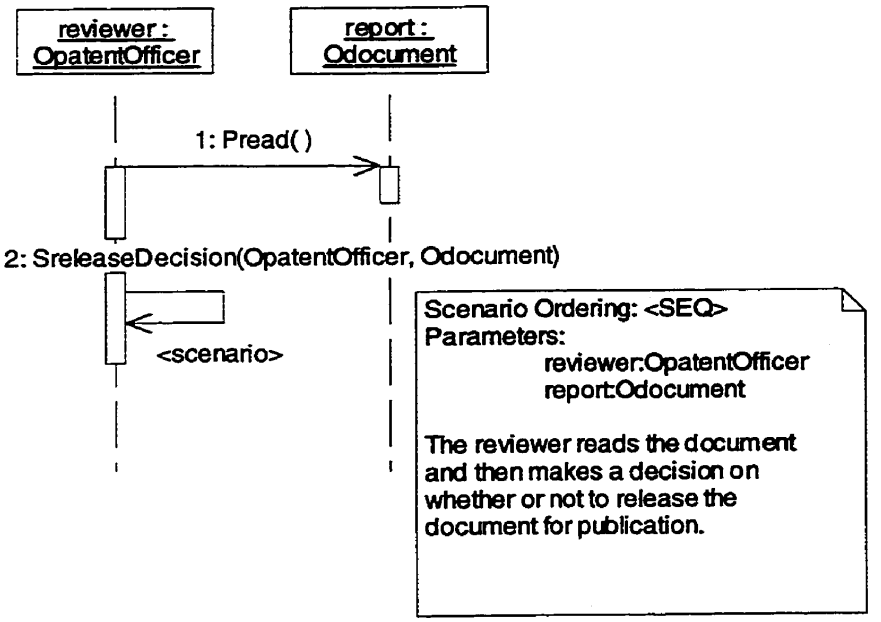

FIGURE 2.7. Scenario Type SdocReview

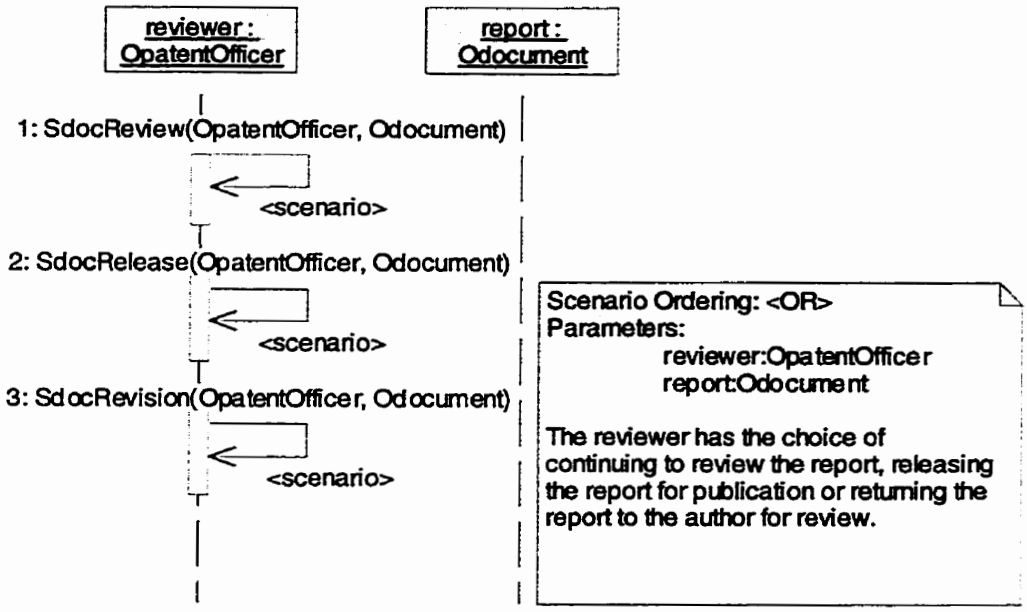

FIGURE 2.8. Scenario Type SreleaseDecision

teractions (messages). As with MSCs the life-lines are time-lines, so messages further down a life-line occur after messages further up the life-line.

Each message sequence diagram by itself describes a particular type of scenario consisting of one or more messages. The messages may only occur between objects of the types specified by that scenario type. The messages may only occur in an order consistent with the order specified by the scenario type. For example, scenario *SdocEdit* in Figure 2.4 specifies that an author of a document may, in sequence, read from and then write to some document.

The MSC standard allows modular design via sub-MSCs and decomposed process instances [Mau96]. This decomposition is process-based, and does not move across well to UML-style message sequence diagrams since the diagram is object-based and not process based. Many objects in an MSD can (and often do) belong to the same process. Modular design is denoted in the context of SBAC modeling by using what

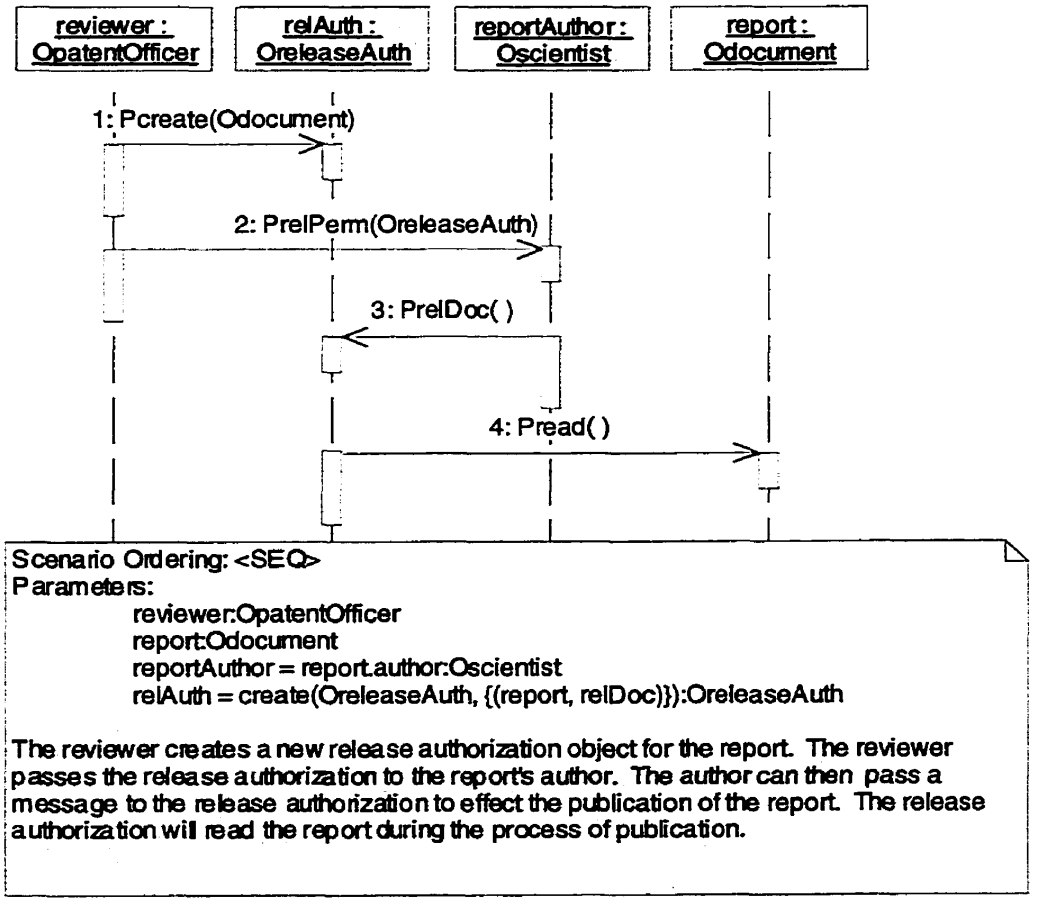

FIGURE 2.9. Scenario Type SdocRelease

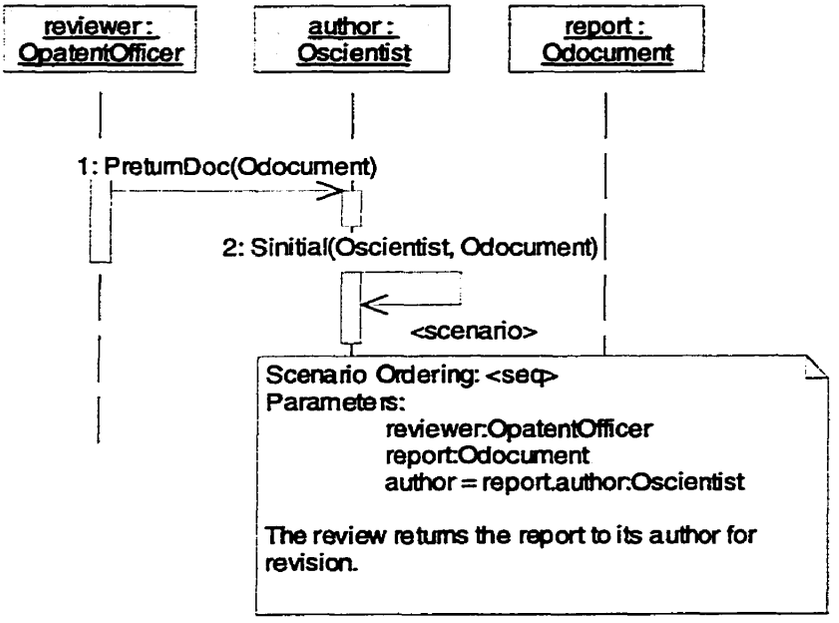

FIGURE 2.10. Scenario Type SdocRevision

are in effect sub-scenarios (or child scenarios), which in MSCs (if allowed) would be interpreted as decomposed message events. This is the intent of the messages marked here with the <scenario> tag. In these cases the tag indicates the creation of another scenario. For example the scenario *Sinitial* in Figure 2.3 indicates that a *SdocEdit* scenario is created followed by the creation of an *SforwardDecision* scenario. This notation for capturing the composition of message sequence diagrams is not part of UML or of ITU message sequence charts. It is used to support the scenario composition mechanism proposed in this dissertation for scenario-based access control. In Rational Rose, the <scenario> tags are normally captured in a data structure associated with the message documentation, but in this example they are presented on the message sequence diagrams to make them visible to the reader. Other features of the diagrams will become apparent later in the dissertation.

The semantics of MSDs can be interpreted using the proposed scenario-based

access-control modeling scheme. One can then use instances of the modeling scheme to provide a safety analysis for the system being specified. This ties security modeling to contemporary software engineering techniques. This dissertation does not provide a formal semantics for MSDs based on SBAC modeling (although this is likely possible and worthwhile). It focuses on the presentation of SBAC modeling and uses the vehicle of message sequence diagrams as a tool for intuitive understanding of SBAC models. That is, MSDs are intuitively straightforward and useful for thinking about and describing object scenarios, which are then captured by the SBAC models. As well, the analysis tool developed as part of this dissertation uses the support for message sequence diagrams built into the Rational Rose modeling tool to help with capture of SBAC models.

Chapter 3

Modeling Scenarios

3.1 Introduction

With the shift in emphasis in research on access-control modeling away from security requirements for confidentiality in government/military systems towards integrity-based requirements in commercial systems, there is an increasing requirement to model legitimate use in secure systems. This can been seen in the motivation for RBAC and TBAC presented in Chapter 2. The development of the scenario-based access-control scheme presented here is driven by two main goals. The first is to provide a scheme that will provide efficient safety analysis for systems modeling legitimate use policies. This implies efficient analysis of non-monotonic systems. This is because legitimate use policies that employ some kind of just-in-time availability of access-control permissions are inherently non-monotonic. The second goal is to provide a modeling scheme that complements contemporary software engineering modeling techniques. The objective is to leverage the information that is already being captured by such techniques and to provide security modeling as an extension to existing software engineering methods. This eliminates duplication of effort in security modeling and may serve to encourage the wider use of security modeling.

Significant success in providing safety analysis for security models for monotonic systems has been achieved by exploiting the concept of maximal state. As can be seen in many of the modeling schemes presented in Chapter 2, the strategy for analysis is to allow system state to expand (permissions to be added) until no further expansion is possible. The resulting maximal state can then be inspected to provide the safety analysis. The analysis scheme must show that expansion of the permission state is

controlled, to ensure that a maximal state always exists and is computable.

One of the chief difficulties in modeling non-monotonic systems is that the existence of a maximal state does not seem likely. In fact, the concept of maximal state seems counter-intuitive in such systems. If access permissions are allowed to be created and revoked then it seems that many mutually exclusive evolutions of execution for the system are possible. For example many different users may come to be the owner of an object, but none of them may be owners of it simultaneously. As well, the nature of the safety problem seems to change slightly. History, or context, becomes important in non-monotonic systems. It is important to know not only if a permission for a subject to access an object is possible but when that permission becomes available. By 'when' it is meant 'when in relation to the existence of other permissions.' For example, two subjects may both be allowed to have a certain access to an object, but they may only be able to have that access when they also have the owner permission for the object. A single maximal permission state does not seem to be an adequate basis for analysis of such safety criteria.

This dissertation was inspired by the progression of access-control models culminating in Sandhu's SPM [San88] and Sandhu and Ganta's TRM [SG94]. In particular, the unfolding mechanism used by these modeling schemes to control the state explosion inherent in subject creation is a very powerful technique. It seemed that a successful adaptation of this technique might be to use such an unfolding scheme to limit the state explosion problem inherent in the expanding histories of non-monotonic systems. That is, to try to define some maximal set of possible histories instead of a maximal set of permissions.

Scenario-based access control was developed independently from, but is related to, task-based access control [TS94, TS97]. As is the case with TBAC, this research recognizes that an obvious basis for secure workflow management is a just-in-time policy of granting permission, based on providing just those rights that are needed,

when they are needed, to accomplish a legitimate task. Role-based access control and Clark-Wilson-based models [CW87] provide for least privilege and the fine grain definition of permission types. Clark-Wilson goes as far as to specify transformation procedures which limit the kinds of operation that can be applied to data. What is missing from these modeling schemes is the notion of context, or order, in which operations are permitted to take place.

SBAC begins with the observation that when using object-oriented techniques, software analysts and designers may specify a system by describing a set of scenarios. The amount of detail presented in the scenarios depends in part on the level of abstraction of the specification. Early in the system life cycle analysts use scenarios to describe the nature of the problem they are working on. Later in the system life cycle designers use scenarios to describe how they are going to solve the problem, *i.e.* the specification of the software solution. In both cases they use scenarios to specify a set of mechanisms that describe how objects will interact. As the OOA/OOD review material of Section 2.3 highlighted, scenarios are specifications of a particular set of object interactions. More generally, a specific kind (or type) of scenario can be used to describe what kinds (classes/types) of objects are involved and what kinds of messages are exchanged. Many objects are instantiated in the life of a system but they are intended to follow a set of behaviours as laid out in the scenario descriptions. The designers use scenarios to describe what object interactions are necessary to meet system requirements. With a change of perspective these scenarios provide the basis for security modeling. Under SBAC, the designer is also using scenarios to describe what object interactions are permitted by the system. That is, the designer is now specifying scenarios that are necessary and sufficient to meet the system requirements; no other object interactions will be permitted. A permission can be generated for each step in a scenario and consumed as the specified object interaction takes place. Such scenario-based security models are inherently non-monotonic.

This approach seems to have several strengths. In SBAC scenarios are seen as a natural vehicle for the specification of the context required in expressing a just-in-time security policy. One of the stated goals of this research was to provide efficient analysis for non-monotonic systems. As will be seen, the creation relationship between scenarios may be exploited to provide control over the state explosion of expanding histories of non-monotonic systems. The use of contemporary OO modeling techniques and tools meets the second goal stated for this research; information that is already being captured by using OO methods can be used to provide security modeling as an extension to an existing software engineering process. OO techniques are used at high and low levels of abstraction. An advantage of OO methods and SBAC security models based on such methods is that model constructions tend to remain closely related to problem domain entities. Especially at the higher levels of abstraction, designers strive to make the objects and their behaviours abstractions of problem domain entities. This provides intuitive semantic content that takes advantage of natural human cognitive skills [Boo94]. This makes it easier for the designer to cope with the complexity inherent in a system.

The chapter is organized in the following way. The sections at the beginning of the chapter present the components of basic SBAC modeling. The sections define the basic components necessary to model scenarios, scenario types, and the interaction between scenarios. Each set of components will be discussed first, then a formal definition of the components will be presented. An example based on the document release example of Chapter 2 will be included in each case to help develop intuition for the modeling scheme. Later sections of the chapter add complexity to the basic modeling scheme. Components will be added to provide support for representing visibilities between objects, for representing object creation, and for supporting information hiding between scenario types.

3.2 Modeling Scenarios

The usual approach in security modeling is to define sets of subjects (entities which require access to information, devices, programs, etc.) and objects (entities that can be accessed). In SBAC an object-oriented decomposition of system entities is used and all entities in the system will be referred to as objects. Objects here may serve in either of the usual security model roles of subject or object. An object might possibly request a service of another object in the system, or might itself be accessed by other objects; *i.e.* an object may play the role of message sender or receiver at different times. Each object has an object type. Object types are fixed for the life of the object.

The most basic or primitive scenario describes a single message pass between two objects. To send a message to an object, visibility is required. That is, the sending object must know the name of, or have a reference to, the receiving object. The message being sent must also conform to the interface of the receiving object. The basic element of access control is based on a primitive scenario describing exactly one message pass. The scenario will specify which two objects are involved, the types of those objects, and the type of the message to be passed. The concept of a message here is general and describes the interaction of a sending object with an interface of the receiving object. A message can carry information to the receiving object, modify the state of the receiving object, and return information dependent on the state of the receiving object. The receipt of a message can invoke some behaviour in the receiving object. This definition provides fine-grain access control similar to the permissions defined by RBAC modeling or the transformation procedures described by Clark-Wilson. Note that although messages may alter the state of an object, the state of objects is not directly modeled in SBAC. The state of an object is hidden, but the interface of an object is modeled through the specification of the messages it

exchanges with other objects.

Complex scenarios (non-primitive scenarios) are defined using collections of sub-scenarios (child scenarios). Child scenarios can be primitive or non-primitive. Each child scenario can have at most one parent scenario. This allows the model to describe the relationship between scenarios. Each scenario has a scenario type, which is fixed for the life of the scenario. A scenario type specifies the types of the objects which participate, or interact, in a scenario instance of its type. A scenario type also specifies the types of the scenario's children and the order in which the child scenarios are permitted to be created.

A complex scenario can be thought of as the root of a tree of child scenarios with primitive scenarios at the leaves of the tree. Such a tree specifies a permissible set of interactions for a collection of system objects. The tree describes which objects are allowed to participate, the types of the objects, the message instances involved, the types of the messages, and the ordering of messages. The topology of the tree is constrained by the scenario types involved. This is because each scenario type specifies what type of child scenarios it can create and the order in which the scenario creations are permitted to take place. Ultimately the tree specifies the message passes which are permitted to take place, the objects participating in the message passes, and the order in which they can occur. Scenario types and object types are statically defined and form the basis of a mandatory security policy.

3.2.1 Objects and Object Types

The object is one of the most fundamental abstractions in the scheme being described. Every object in a system has a type. Presentation of the modeling scheme will begin by defining sets of object types and objects. A set of identifiers is also defined. Identifiers are used to name individual instances of object types and objects as well

as other model constructions.

An object type is assigned when an object is created (instantiated). The assignment does not change. In the document release example presented in Chapter 2 object types might be specified for *Oscientist*, *OpatentOfficer*, *OreleaseAuth* and *Odocument*. The function τ_O defines type assignments for specific objects. The set $\mathcal{PB}$ defines the domain of parameter binding pairs, *i.e.* identifier-object pairs. In such a pair an identifier is bound to a specific object in a model. A pair $(author, oalice)$ denotes that the identifier *author* is bound to the object *oalice* in some context. A similar set of binding pairs is defined by the set $\mathcal{PTB}$. In this case the identifiers are bound to object types. For example the pair $(author, Oscientist)$ denotes that the identifier *author* is bound to the object type *Oscientist* in some context. The following definition formalizes these components of the model.

$\mathcal{OT}$	a finite set of object types
$\mathcal{O}$	a finite set of objects
τ_O	object type function, $\tau_O : \mathcal{O} \rightarrow \mathcal{OT}$
$\mathcal{I}$	a finite set of identifiers
$\mathcal{PB}$	a finite set of parameter binding pairs, $\mathcal{I} \times \mathcal{O}$
$\mathcal{PTB}$	a finite set of parameter type binding pairs, $\mathcal{I} \times \mathcal{OT}$

3.2.2 Scenario Types

The static structure of a security policy is based on how objects are permitted to interact in scenarios. The modeling scheme defines a set of scenario types that are used to specify how different types of objects may interact. Scenario types typically specify common or recurring kinds of behavior. A scenario type can be thought of as a template. The template specifies how actual objects may combine and interact in an actual instance of a scenario of that type during the evolution of a system. As a

system evolves, real objects can only interact in scenario instances defined using one of these scenario types.

The *authorization of a scenario* of some scenario type means that a new scenario of that type is instantiated (created). Newly authorized scenarios are added to the set of existing scenarios, $\mathcal{S}$. Authorization of a scenario permits security relevant actions to take place. Two kinds of security relevant actions may be permitted by a scenario. They are, the authorization of another new scenario, and the authorization of a message pass between two specific objects. The *authorization of a message pass* means that a specific kind of message is permitted to be sent from one specific object to another specific object. The only other security-relevant action is the sending of a message. Scenarios are not directly involved in the sending of messages. The actual system objects collaborate in the sending of messages. The mechanism of message passing is not directly modeled by SBAC.

When a new scenario is authorized an initial set of actions is permitted. The occurrence of an action in the system can cause the permitted actions associated with a scenario to change. *I.e.,* some actions may no longer be permitted and new actions may become permitted. A scenario is *authorized* when there are security relevant actions permitted by it. A scenario is *terminated* (no longer authorized) when there are no longer any permitted actions associated with a scenario. Once terminated a scenario cannot become authorized again. The discussion of the SBAC modeling scheme will usually refer to scenario authorization instead of scenario creation because authorization implies creation or instantiation of a new scenario, and the permissions for an initial set of actions.

In this modeling scheme, primitive scenario types model a single message pass between objects. Primitive scenario types describe the types of the two objects involved with the message pass and the types of parameter objects associated with the message. The only security relevant action permitted for an instance of a primitive

scenario type is the authorization of a message pass between two specific objects. Only instances of a primitive scenario type can authorize a message pass between two objects. The only action permitted by non-primitive scenarios is the authorization of new scenarios. The message authorization is consumed (revoked) when the message pass takes place. Revoked authorizations are not necessarily recoverable. This is why the scheme is inherently non-monotonic. Message passes which are not currently permitted by some scenario (*i.e.* do not occur in the model) are prohibited. Primitive scenarios and primitive scenario types are described in more detail in Section 3.2.4.

Each scenario type is a member of a finite set $\mathcal{ST}$. A scenario type has a number of properties associated with it. There is a set of parameters that specify the types of the objects which participate in the scenario. The function $para_{ST}(\Phi)$ specifies a finite set of parameter bindings for a given scenario type Φ. The scenario type provides a context, or namespace which acts as a scope for these bindings. Since scenario types are meant to describe the interactions among objects of specific types, the identifiers can be thought of as roles that certain types of objects play in the context of the scenario type in which they are defined, with one, and only one, object for each role. The type of the object which plays the role is specified by the parameter binding. More than one binding to the same object type is allowed in a scenario type. This corresponds to different objects of the same type interacting in a scenario instance by filling different roles. Conversely, the same object may play multiple roles in a scenario instance provided the roles are of the same type.

A finite set of scenario descriptors specifies the child scenarios that may be authorized by scenarios of each type. Each descriptor specifies the type of a child scenario and a mapping between the parameters of the parent scenario and the child scenario. Scenario descriptors are triples. The first two elements of a scenario descriptor specify a scenario type and a parameter mapping to be used in authorizing a scenario of that type. A finite set of identifier pairs maps parameter identifiers in the context of

the parent scenario to parameter identifiers in the context of the child scenario being authorized. Effectively, this maps roles played by objects in the parent scenario to roles played by objects in the child scenario. The type bindings of the identifiers in the context of their respective scenarios must agree. Also, for a scenario descriptor defining a scenario of type Φ, there must be an identifier mapping provided for each binding defined in set $para_{ST}(\Phi)$. The scenario descriptor identifier mappings and the parent scenario's actual parameters uniquely identify the objects that will participate in the child scenario (the child scenario's parameter set).

The third element of the scenario descriptor triple is a boolean value that indicates whether the child scenario is concurrent ($True$) or not concurrent ($False$) with the parent. A concurrent scenario allows a separate thread of authorization orderings to begin with that scenario. The evolution of authorization orderings is described in more detail in the definition of the $order_{ST}$ function below.

The function $child_{ST}$ specifies a sequence of scenario descriptors. The descriptors specify the kind of child scenarios that may be authorized by the parent scenario.

Each scenario type also has an ordering that specifies whether the child scenarios are executed in sequence, are mutually exclusive (only one child authorization can occur), or are all authorized without particular regard to order. The function $order_{ST}$ specifies the ordering of child scenarios. The ordering specifies when the authorization of a child scenario occurs. The ordering of message pass actions depends on the evolution of the system execution. The pattern and combination of child scenarios which may be authorized by a scenario are constrained by the ordering specified by the scenario's type. When a certain set of conditions is met, a scenario terminates. A terminated scenario is no longer permitted to perform any action. A primitive scenario terminates when its message authorization is consumed (*i.e.* when the message pass takes place). The termination of non-primitive scenarios depends on the ordering specified by their scenario type. There are three kinds of orderings defined by the

modeling scheme (*seq*, *or*, and *and*).

Upon authorization of a scenario which has a sequential, *seq*, ordering, the child scenario defined by the first scenario descriptor in the $child_{ST}$ sequence is immediately authorized. When this child scenario terminates, the child scenario defined by the second scenario descriptor in the $child_{ST}$ sequence is authorized, and so on for the rest of the sequence. A *seq* ordered scenario terminates when the child scenario defined by the last scenario descriptor in the $child_{ST}$ sequence terminates.

Upon authorization of a scenario which has an *or* ordering, all scenarios defined by the scenario descriptors in the $child_{ST}$ sequence are immediately authorized. A message pass action associated with any one of these child scenarios (or the child's descendants) causes the termination of all the other child scenarios. That is, messages are effectively only permitted for one of the child scenarios defined by the $child_{ST}$ sequence. This is because message pass authorizations associated with the other child scenarios are revoked when the scenarios are terminated. An *or* ordered scenario terminates when the one remaining child scenario terminates.

Upon authorization of a scenario which has an *and* ordering, all scenarios defined by the scenario descriptors in the $child_{ST}$ sequence are immediately authorized. A message pass action authorized by any one of these child scenarios (or the child's descendants) may occur in any order. An *and* ordered scenario terminates when all its child scenarios terminate.

By construction there can be no outstanding authorizations for a scenario when it terminates.

The following definition formalizes these components of the model.[1]

$\mathcal{ST}$	a finite set of scenario types.
$para_{ST}$	a function defining the parameter type bindings associated

[1]The symbol $\rightharpoonup$ is used to denote a partial function

	with a scenario type; $para_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $para_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$
$\mathcal{SD}$	a finite set of scenario descriptors, $\mathcal{ST} \times 2^{\mathcal{I} \times \mathcal{I}} \times 2^{1}$
$child_{ST}$	a function defining the scenario descriptors associated with a scenario type, $child_{ST} : \mathcal{ST} \rightarrow \mathcal{SD}^{n}$, where n is finite
$order_{ST}$	a function defining the ordering of child scenarios associated with a scenario type, $order_{ST} : \mathcal{ST} \rightarrow \{seq, or, and\}$

As an example of scenario type specification, the scenario type *SdocReview* from the document release example in Chapter 2 is defined in the following way. This scenario type is repeated in Figure 3.1 with some additional detail.

$$SdocReview \in \mathcal{ST}$$
$$para_{ST}(SdocReview) =$$
$$\{(reviewer,\ OpatentOfficer), (report, Odocument)\}$$
$$child_{ST}(SdocReview) =$$
$$\langle (Pread, \{(reviewer, sender), (report, receiver)\}, False),$$
$$(SreleaseDecision, \{(reviewer, reviewer), (report, report)\}, False)\rangle$$
$$order_{ST}(SdocReview) = seq$$

The scenario type has two parameter roles *reviewer* and *report* of types *OpatentOfficer* and *Odocument* respectively. These roles can be seen in Figure 3.1 as object boxes at the top of the figure.[2] The scenario type specifies the authorization of two child scenarios. The first is a primitive scenario of type *Pread*. When creating this child, the *reviewer* role of *SdocReview* is mapped to the *sender* role of the child and the *report* role is mapped to the *receiver* role of the child. These role mappings

[2]The MSDs presented in this chapter are intended to be an aid to the reader in developing intuition for scenario-based modeling. The MSDs and their interpretation are not part of the formalism but provide a set of parallel examples. Introducing MSDs here also provides familiarity with the notation. Later, examples of SBAC modeling will be expressed using MSDs.

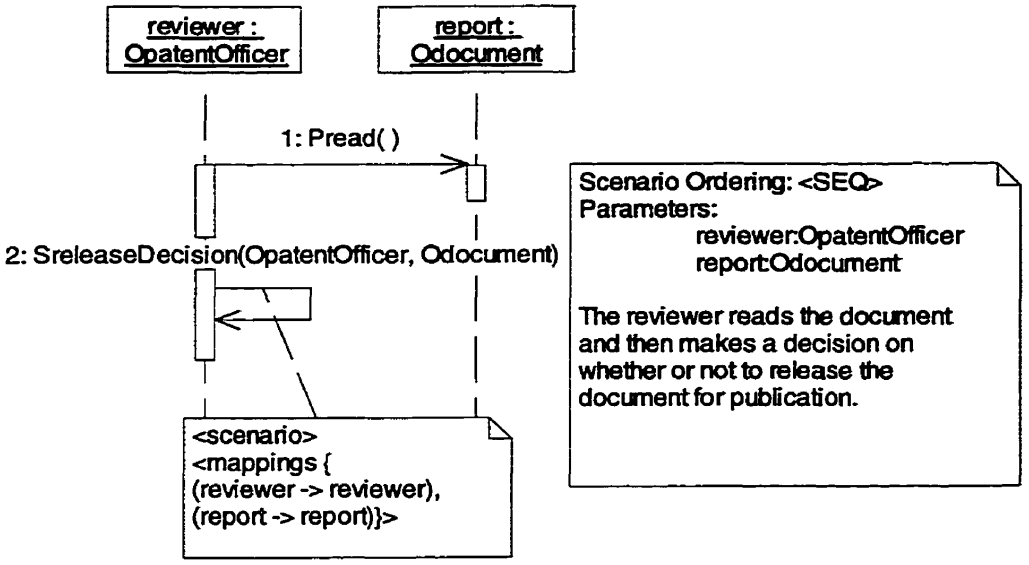

FIGURE 3.1. Scenario SdocReview with detail

are implicit in the topology of the diagram and are not marked as parameters of *Pread*. More detail with respect to the *reviewer* and *sender* role mappings of primitive scenarios will be presented in Section 3.2.4. The *Pread* child is non-concurrent with its parent as indicated by the third element of its scenario descriptor, *False*. The second child to be created is of type *SreleaseDecision*. The comment box associated with this child scenario in the figure contains information that is normally contained in a message description data structure in the Rational Rose model from which the figure was generated. To make it visible to the reader the information is shown here in a comment box. The <scenario> tag indicates that this is a non-primitive scenario. Such a distinction is not required in the formal presentation above. In the formal presentation and in the corresponding mappings presented in the comment box it can be seen that the *reviewer* and *report* roles of *SdocReview* are mapped to *reviewer* and *report* roles of the child. The roles just happen to have the same names in this case. The *SreleaseDecision* child is also non-concurrent. Concurrency would be

indicated in the figure by specifying the <concurrent> flag for the message in much the same way as the <scenario> flag is specified. The ordering of the *SdocReview* scenario type is *seq*. Therefore, the authorization of the *SreleaseDecision* child will occur only upon termination of the *Pread* child.

3.2.3 Scenario Instances

Scenario types provide static restrictions on the interactions permitted in a system between objects of various types. Evolution of the system proceeds via the authorization of actual scenarios and performing the permitted security-relevant actions. Each scenario is a member of the finite set $\mathcal{S}$. A scenario has a number of properties associated with it.

Every scenario in a system has a type specified by the scenario type function τ_S. Scenario types are assigned when a scenario is authorized and do not change. For each scenario there is a set of parameters which specify the objects which participate in the scenario. $para_S$ specifies bindings to the object instances participating in the scenario. A scenario is a context, or namespace which acts as a scope for these bindings. Again, the identifiers can be thought of as roles certain objects play in the context of the scenario in which they are defined. The identifiers are the same identifiers used in the specification of $para_{ST}$ for the scenario's type. That is, for each parameter binding of a scenario instance there will be a parameter type binding in its scenario type such that the bindings have the same identifier. The type of the object specified for an identifier in a binding in $para_S$ must agree with the object type specified for the same identifier in $para_{ST}$.

Each scenario also has a set of child scenarios which it has authorized. $child_S$ is a function mapping scenarios to sequences that specify the children the scenarios have created. As child scenarios are authorized they are appended to the sequence mapped

to the parent scenario by $child_S$ (initially null). A new scenario is authorized using a scenario descriptor belonging to the parent's type. The scenario descriptor(s) used to authorize a new scenario(s) at a particular point in a scenario's life depends on the scenario's ordering. A scenario descriptor specifies the type of the child scenario and the parameter mapping to be used in authorizing that scenario. The set of identifier pairs in the scenario descriptor maps identifiers in the context of a parent scenario to identifiers in the context of a child scenario being authorized. Since the role identifiers are the same for the bindings of the scenario types and scenarios instances this maps objects playing roles in the parent scenario to objects playing roles in the child scenario. New parameter bindings are created, which bind role identifiers for the child scenario to the identified objects participating in the parent scenario. These new bindings specify the members of $para_S$ for the child scenario.

The actions permitted by a child scenario may proceed within the parent's sequence of actions, or those actions may proceed concurrently with the parent scenario's sequence of actions. The predicate con_S defines whether a separate, concurrent sequence of authorization orderings begins with a specific scenario ($True$), or whether the scenario's authorization orderings are part of the parent's sequence ($False$).

The following definition formalizes these components of the model.

$\mathcal{S}$	a finite set of scenarios.
τ_S	scenario type function, $\tau_S : \mathcal{S} \rightarrow \mathcal{ST}$
$para_S$	a function defining the parameter bindings associated with a scenario; $para_S : \mathcal{S} \rightarrow 2^{\mathcal{PB}}$, such that for $\phi \in \mathcal{S}$, $para_S(\phi) : \mathcal{I} \rightharpoonup \mathcal{O}$
$child_S$	a function defining the child scenarios associated with a scenario, $child_S : \mathcal{S} \rightarrow \bigcup_{i=0}^{n} \mathcal{S}^i$, where n is finite
con_S	a predicate defining the concurrency associated with

a scenario, $con_S : \mathcal{S} \rightarrow 2^1$

To illustrate how scenarios are specified, and how the scenario creation mechanism works, an example will be presented, which is based on the *SdocReview* scenario type presented at the end of Section 3.2.2 and in Figure 3.1. First an instance of a *SdocReview* scenario type will be modeled, then a new *SreleaseDecision* scenario will be created using the appropriate scenario descriptor. An instance *sdocReview1* is modeled as follows:

$$sdocReview1 \in \mathcal{S}$$
$$\tau_S(sdocReview1) = SdocReview$$
$$para_S(sdocReview1) = \{(reviewer, oalice), (report, odoc)\}$$
$$child_S(sdocReview1) = \langle pread2 \rangle$$
$$con_S(sdocReview1) = False$$

Scenario *sdocReview1* is of type *SdocReview*. The *reviewer* role is mapped to an object instance *oalice* and the *report* role is mapped to an object instance *odoc*. It can be seen here that the scenario has already authorized one child scenario, *pread2*. Although the example does not provide detail for *pread2*, it is presumably a primitive scenario that would authorize a read message from the object *oalice* to the object *odoc*. From its scenario type *sdocReview1* has a *seq* scenario ordering. Therefore when the read message from *oalice* to *odoc* takes place, and consumes its authorization (terminating *pread2*), *sdocReview1* will authorize a new scenario. The authorization of the new scenario will be based on the second scenario descriptor in $child_{ST}(SdocReview)$. The new scenario will be of type *SreleaseDecision*. The role *reviewer* of the new scenario will be mapped to the object playing the role of *reviewer* in *sdocReview1*, *i.e. oalice*. The role *report* of the new scenario will be mapped to the object playing the role of *report* in *sdocReview1*, *i.e. odoc*. Again, note that the identifiers for the roles

in the parent and child scenario do not have to be the same, as they are here. The mapping between roles is provided by the scenario descriptor. The new scenario will not have authorized any new scenarios itself yet. Its set of child scenarios is initially null. The authorizations generated by the newly authorized scenario will not proceed concurrently with those of *sdocReview1*. *I.e.,* in this case *sdocReview1* will not proceed with its own actions until the new scenario terminates. Let the new scenario be *sreleaseDecision3*. It is modeled as follows:

$$sreleaseDecision3 \in \mathcal{S}$$
$$\tau_S(sreleaseDecision3) = SreleaseDecision$$
$$para_S(sreleaseDecision3) = \{(reviewer, oalice), (report, odoc)\}$$
$$child_S(sreleaseDecision3) = \langle\rangle$$
$$con_S(sreleaseDecision3) = False$$

3.2.4 Primitive Scenario Types

As noted above, messages in SBAC modeling are primitive scenarios that describe a single distinct interaction between objects. Scenario types for these primitive scenarios describe the types of the two objects involved with the message pass and the types of parameter objects associated with the message. These are specified by the parameter type bindings defining the roles participating in the scenario. Each primitive scenario type is associated with a particular kind of message. $para_{ST}$ defines the object roles and types associated with the message. $para_S$ defines the actual object parameters. With respect to $para_{ST}$ the identifiers *sender* and *receiver* are reserved for the object types filling the role of the message sender and receiver respectively. With respect to $para_S$, identifiers *sender* and *receiver* are reserved for the actual objects filling the roles of message sender and receiver. Any other bindings specified by $para_{ST}$ ($para_S$) specify the object types for remaining message parameters. $child_{ST}$

and $child_S$ for primitive scenario types are always the null set. $order_{ST}$ for primitive scenario types is undefined.

Upon authorization of an instance of a scenario of a primitive type, a message of the kind associated with that primitive type is authorized to be sent from the *sender* to the *receiver*. The message parameters are specified by $para_S$ for the primitive scenario. The authorization for the message is consumed when the associated message is sent. A primitive scenario is considered to be terminated when the associated message is sent (*i.e.* when the authorization for the message is consumed).

The modeling scheme does not explicitly model message types or messages. The mapping from primitive scenario types to message types is a bijection. The authorization for a primitive scenario is immediately followed by an authorization for the respective message. The termination of a primitive scenario (revoking its authorization) occurs immediately following the revocation of the authorization for the associated message. For access-control modeling purposes, there is no loss in expressiveness in only considering primitive scenario types and leaving the messages as implicit entities to be defined in the implementation of the access-control mechanism.

3.3 Object Visibilities

As described so far in the modeling scheme, the objects participating in a child scenario have all been specified by the parent scenario when the child scenario is authorized. Using a scenario descriptor, the authorization mechanism maps objects filling roles in the parameter set of the parent to objects filling roles in the new child scenario. So far, these parent supplied objects are the only objects specified to fill roles in the child. Therefore, all subsequent scenario authorizations by the child must use these objects to fill scenario roles. *I.e.,* the set of objects provided as parameters to a child scenario must be a subset of those provided to its parent. Viewed in a

different way, every object which is involved in some interaction in a system must be specified in a parameter binding of the system's initial scenario.

This is a cumbersome and restrictive way of managing a scenario's access to system objects. Scenarios would be obliged to carry object roles as placeholders for children further down the scenario tree, even if those roles are not involved in any action in the current scenario. Also, so far, the modeling scheme does not allow for the creation of objects. All objects involved in system interactions must be available in the initial system scenario. Providing objects with visibilities mitigates these problems.

3.3.1 Defining Object Visibility

This section expands the modeling scheme to allow the expression of object *visibilities* and object creation. It would be convenient if objects could play roles with respect to each other. Objects might be provided with bindings to other objects in association with these roles. These bound objects would be *visible* to the object holding the bindings. Consequently, when an object is provided as a parameter in the creation of a scenario, not only could it become involved in the interactions of the scenario, but any objects visible to that object could be named and could participate.

New functions are defined which allow the modeling scheme to define the visibility objects have of other objects, and to identify an object by using an identifier associated with a set of parameter bindings. The function vis_O defines the visibility between objects by specifying a set of parameter bindings for each object. Objects are a context, or namespace, for a set of parameter bindings. Again, the binding identifiers can be thought of as roles certain objects play in the context of the object in which the binding is defined. The object which plays the role is specified by a parameter binding. A similar relationship is defined for object types.

The types of objects for which a specific object type may have visibility and the

roles those object types can play are defined by the function vis_{OT}. For each object type, the roles other objects are permitted to play in the context of that object type and the types of objects which may fill those roles are specified by parameter type bindings. The identifiers specified for an object's visibility bindings by vis_O are the same identifiers used in the specification of vis_{OT} for the object's type. That is, for each parameter binding of an object instance there will be a parameter type binding in its object type such that the bindings have the same identifier. The type of the object specified for an identifier in a binding in vis_O must agree with the object type specified for the same identifier in vis_{OT}.

The function *objid* dereferences parameter bindings by returning the object being referred to by a specific identifier in the context of a set of parameter bindings. The context providing the parameter bindings may be an object's visibilities (defined by vis_O), or it may be a scenario's parameters (defined by $para_S$). The parameter binding context for some object a defined by $vis_O(a)$ is specified by a partial function from identifiers to objects. Similarly, the parameter binding context for some scenario ϕ defined by $para_S(\phi)$ is specified by a partial function from identifiers to objects.

The following definition formalizes these components of the model.

vis_{OT}	A function defining the types of objects visible to objects of a specific type, $vis_{OT} : \mathcal{OT} \rightarrow 2^{\mathcal{PTB}}$
vis_O	A function defining the visibility between objects, $vis_O : \mathcal{O} \rightarrow 2^{\mathcal{PB}}$
objid	A function which provides an object given an identifier and parameter binding context, $objid : \mathcal{I} \times (\mathcal{I} \rightharpoonup \mathcal{O}) \rightarrow \mathcal{O}$

In the document release example a document might have a role associated with it called *author*. A specific document *odoc*, written by the scientist *oalex*, might have a parameter binding $(author, oalex) \in vis_O(odoc)$. Now, given the document, *odoc*, and

the role, *author*, the scientist object filling the author role can be dereferenced using *objid*, $objid(author, vis_O(odoc)) = oalex$. Constructions having this syntax may be abbreviated as $odoc.author = oalex$.

In a related example suppose that the document object, *odoc*, takes part in a scenario, *sdocumentRelease*1, of type *SdocRelease*. Assume that *odoc* is already bound to an identifier in the scenario, $(report, odoc) \in para_S(sdocumentRelease1)$. An author of the document can be dereferenced using the identifier binding for the document. In the context of the example scenario, $objid(author, vis_O(objid(report, para_S(sdocumentRelease1)))) = oalex$. Constructions having this syntax may also be abbreviated as $report.author = oalex$.

Although the 'dot' syntax is similar to that used in the last example, there is a difference. In the first example the dot denotes dereferencing an identifier in the context of an object, *odoc*. In the second case the dot denotes dereferencing an identifier in the context of an identifier for an object, *report*. This context identifier is then itself used to specify an object in the scenario *sdocumentRelease*1. The second form of using the dot syntax only makes sense when it is obvious what scenario is appropriate for the context. The two usages can be differentiated by the type of the first operand (object or identifier).

3.3.2 Defined Scenario Parameters

Adding object visibilities to the modeling scheme means they can be used in the specification of scenarios. The approach taken is to partition the set of parameter type bindings specified for some scenario type Φ by $para_{ST}(\Phi)$ into two disjoint subsets. One subset specifies the parameters that are part of the external interface for the scenario type and the other subset specifies the parameters that are defined internally for the scenario type and are not part of the external interface for the

scenario type. These are respectively called $paraext_{ST}(\Phi)$, and $paraint_{ST}(\Phi)$. The parameters of the external interface for the scenario type are the parameters used when defining scenario descriptors for that scenario type. *I.e.,* they are used by a parent to define a new child scenario of that type. The internal parameters are not used by the parent to define a new child. These parameters, how they are used, and the fact that they are used at all are a secret of their own scenario type.

For some scenario ϕ, the set $para_S(\phi)$ is also partitioned into two disjoint subsets. The two subsets are $acquiredpara_S(\phi)$ and $definedpara_S$ and respectively reflect object bindings acquired from the parent scenario and bindings defined in terms of object visibilities.

The bindings in the set $acquiredpara_S(\phi)$ are created using objects from the parent's context. That is, the set of identifier pairs in the scenario descriptor of the parent is used to map objects in the context of a parent scenario to objects in the context of a child scenario being authorized. Scenario descriptors in parent scenario types must specify mappings for all identifiers in $paraext_{ST}(\Phi)$. This means that for every binding in $paraext_{ST}(\Phi)$ there will be a binding in $acquiredpara_S(\phi)$ with the same identifier.

The objects specified in parameter bindings in $definedpara_S(\phi)$ are defined in terms of other identifiers available to the scenario. Objects in $definedpara_S(\phi)$ are specified using identifiers found in $para_{ST}(\Phi)$. By using *objid* in the context of the actual scenario instance ϕ these identifiers can be used to identify an object instance, which is playing a role in ϕ. That object in turn can be used as a parameter bindings context that can be used to indirectly identify another object by using *objid* to dereference an object visibility. To do this, a parameter definition is specified using *objid* and an identifier from vis_{OT}. In the context of the actual object instance the identifier refers to some other object instance. A parameter definition specified in this way uses identifiers provided by parameter type bindings and the definition is not dependent

on a specific scenario or object instance.[3] In the context of an actual scenario instance the parameter definition refers to an object instance when *objid* is applied using the actual objects filling the roles for that instance. Therefore, the parameter definitions in $definedpara_S$ are specified once for a scenario type and used by instances of every actual scenario of that type. For every binding in $paraint_{ST}$ there must be a binding specified in $definedpara_S$.

The following definitions formalize these components of the model.

$paraext_{ST}$	a function defining the external parameter type bindings associated with a scenario type; $paraext_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $paraext_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$
$paraint_{ST}$	a function defining the internal parameter type bindings associated with a scenario type; $paraint_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $paraint_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$
$acquiredpara_S$	a function defining the parameter bindings associated with a scenario that are acquired from the scenario's parent; $acquiredpara_S : \mathcal{S} \rightarrow 2^{\mathcal{PB}}$, such that for $\phi \in \mathcal{S}$, $acquiredpara_S(\phi) : \mathcal{I} \rightharpoonup \mathcal{O}$
$definedpara_S$	a function defining the parameter bindings associated with a scenario type that are defined using identifiers available to the scenario; $definedpara_S : \mathcal{ST} \rightarrow 2^{\mathcal{PB}}$, such that for $\Phi \in \mathcal{ST}$, $definedpara_S(\Phi) : \mathcal{I} \rightharpoonup \mathcal{O}$

for $\Phi \in \mathcal{ST}$, $\phi \in \mathcal{S}$, $\tau_S(\phi) = \Phi$:

[3]Note that the creation of scenarios and objects requires that roles defined by the parameter type bindings in scenario and object types be filled by object instances specified by parameter bindings that use the same role identifiers. This one-to-one correspondence using the same identifier name makes parameter definition using scenario and objects types possible.

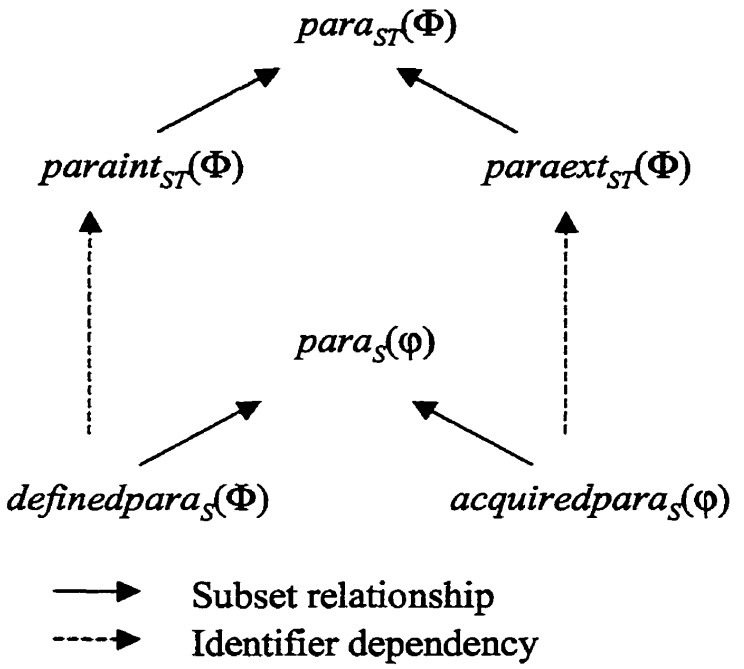

FIGURE 3.2. Relationship between $para_{ST}$ and $para_S$ modified for defined scenario parameters, where $\tau_S(\phi) = \Phi$

$$para_{ST}(\Phi) = paraext_{ST}(\Phi) \cup paraint_{ST}(\Phi)$$
$$para_S(\phi) = acquiredpara_S(\phi) \cup definedpara_S(\Phi)$$
$$paraext_{ST}(\Phi) \cap paraint_{ST}(\Phi) = \{\}$$
$$acquiredpara_S(\phi) \cap definedpara_S(\Phi) = \{\}$$

The relationships among the described parameter sets are illustrated in Figure 3.2. The diagram indicates the subset relationships and the identifier dependencies among the sets. The identifier dependencies between $acquiredpara_S$ and $paraext_{ST}$, and between $definedpara_S$ and $paraint_{ST}$ are defined to mean that they must respectively have the same sets of identifiers. As before, the types of the actual objects in $para_S$ must agree with the object type specified in $para_{ST}$ for its role.

Consider again an example based on a scenario *sdocumentRelease1* of scenario type

SdocRelease. The message sequence diagram for this scenario type is presented again in Figure 3.3 with added detail. Comment boxes have been added to the diagram to illustrate how defined parameters may be specified. There are four objects that participate in any scenario of this type. The object which plays a role *reportAuthor* is associated with a *report* object via a visibility relationship. Perhaps it is not desirable to explicitly define the object playing the author role each time we authorize a scenario of type *SdocRelease*. Specifying the *report* object should be sufficient because it has its *author* associated with it. A defined scenario parameter allows this to be specified. The parameter type bindings for the role *reportAuthor* is a member of the set $paraint_{ST}(SdocRelease)$. The set $definedpara_S(SdocRelease)$ must specify a parameter binding using this identifier. The parameter definition corresponding to the *reportAuthor* identifier is shown in the comment boxes using the <DEFPARA> tag. In this case the <DEFPARA> tag indicates that that object is a defined parameter bound to *report.author*. We will ignore the other comment boxes and the <PARAIN> tag for now.

More formally, when the scenario *sdocumentRelease*1 was created it would have been provided the following object bindings from the parent's external environment:

$$(reviewer, oalice), (report, odoc) \in acquiredpara_S(sdocumentRelease1)$$

$definedpara_S$ for *sdocumentRelease*1 would specify an object binding for the author of *report* in this way:

$$(reportAuthor, report.author) \in definedpara_S(SdocRelease)$$

In scenario *sdocumentRelease*1 the identifier *reportAuthor* refers to the object playing the *author* role with respect to the object bound to the identifier *report*. In the previous example (Section 3.3.1) the report author was *oalex*, so *reportAuthor* would refer to *oalex* in this case.

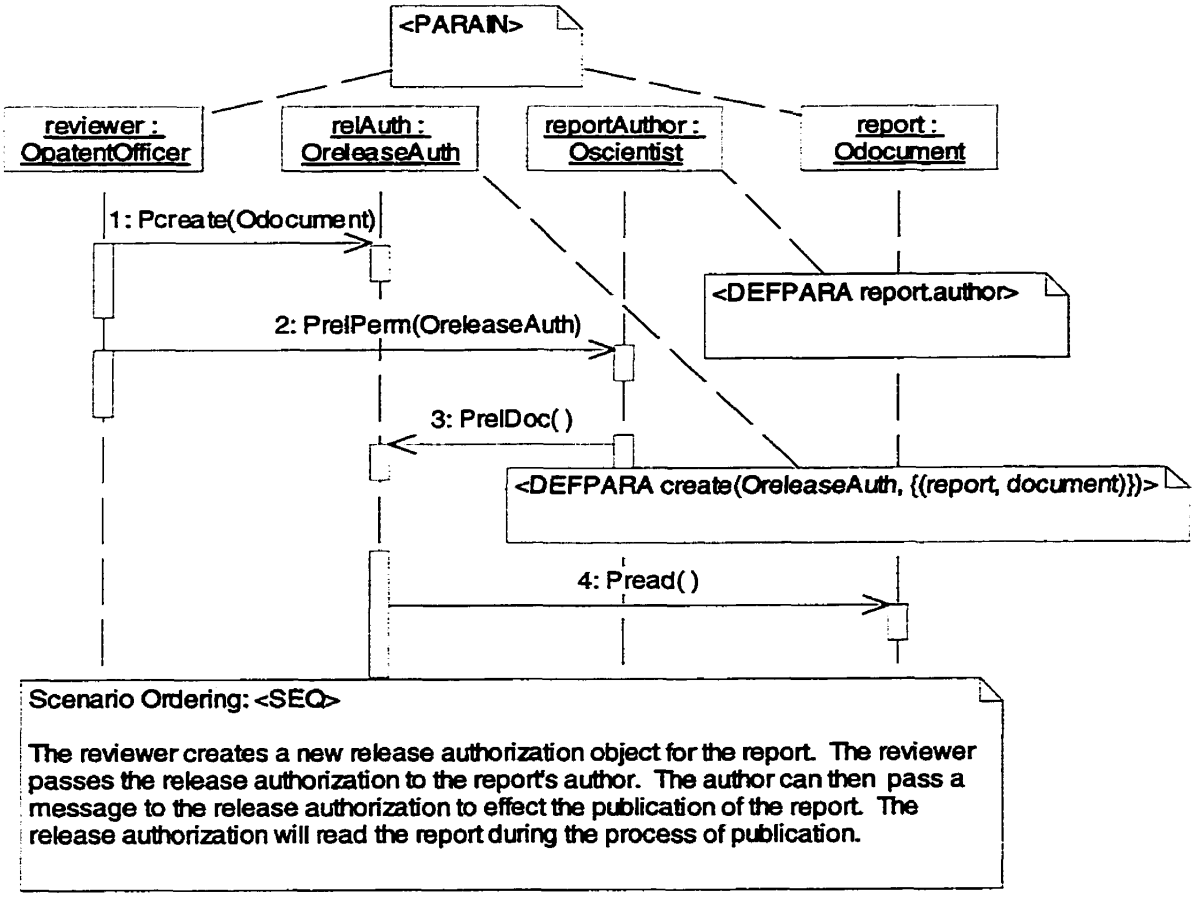

FIGURE 3.3. Scenario SdocRelease with detail

3.3.3 Object Creation

New objects created by a scenario to be used in its interactions can be handled by the same mechanism introduced to support defined scenario parameters. A new function is introduced in the modeling scheme to specify object creation. The *create* function is applied to an object type and a set of identifier mappings. It returns a new object of the specified type. This object is added to the set $\mathcal{O}$. At creation time the visibilities of the new object are specified by the set of identifier mappings. The identifier mappings map roles specified by $para_S$ of the creating scenario to roles in the new object. The parameter bindings of vis_O for the new object are created accordingly. A mapping must be specified for each identifier of vis_{OT} for the object type being created. The following definition formalizes the *create* function.

create — A function returning a new object of a specified type,
$create : \mathcal{OT} \times 2^{\mathcal{I}\times\mathcal{I}} \rightarrow \mathcal{O}$

To introduce a new object into a scenario, a parameter binding in $definedpara_S$ is specified using the *create* function. Consider again the *SdocRelease* example of Figure 3.3. A scenario *sdocumentRelease*1, of type *SdocRelease*, requires the creation of an object of type *OreleaseAuth* (a release authorization). The following specification indicates that an *OreleaseAuth* object is created and bound to the identifier *relAuth*.

$$(relAuth, create(OreleaseAuth, \{(report, relDoc)\})) \in definedpara_S(sdocumentRelease1)$$

For this new object the role identifier *relDoc* will be bound to the object specified by the role identifier *report* of the *sdocumentRelease*1 scenario. Note in Figure 3.3 the comment box associated with the object *relAuth* contains a <DEFPARA> tag.

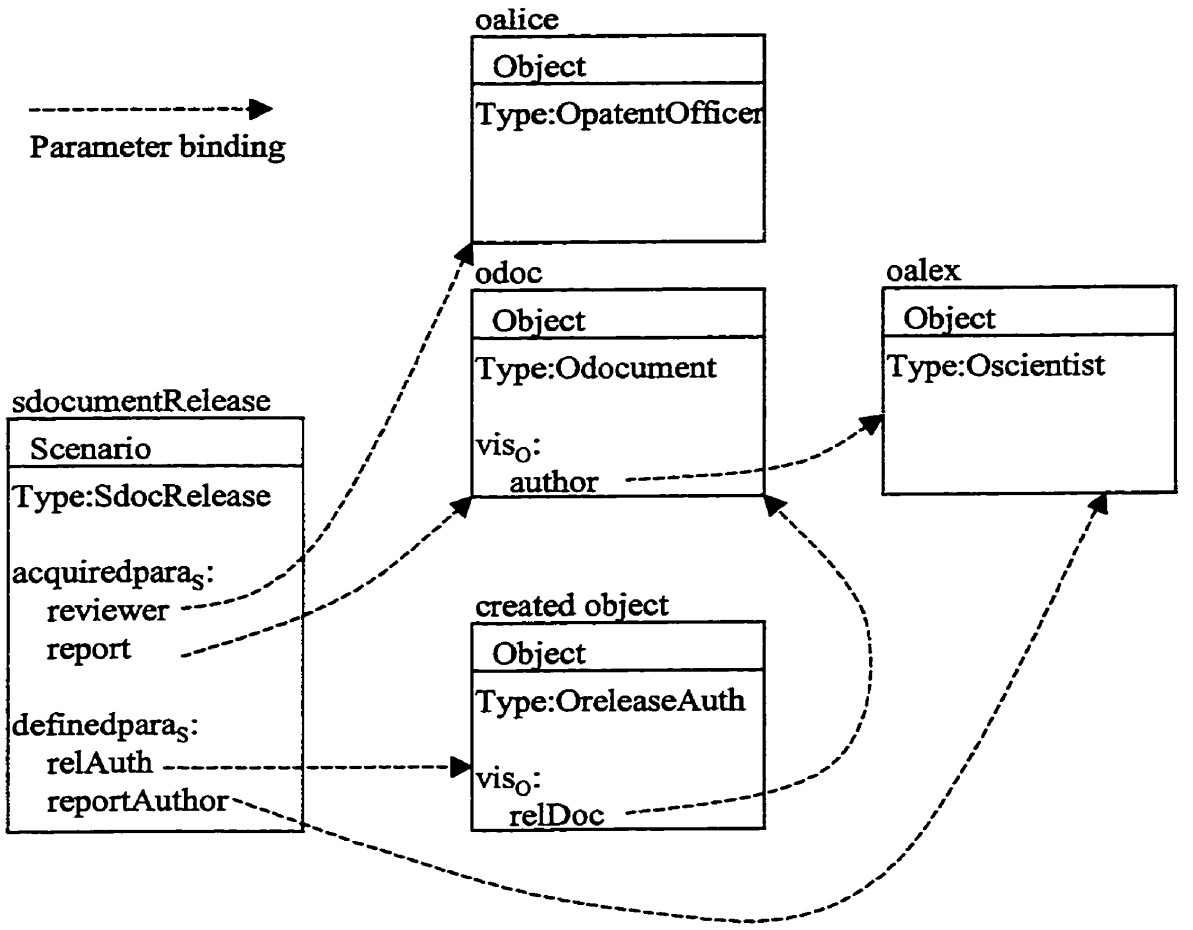

FIGURE 3.4. Parameter bindings for SdocRelease example

In this case the tag indicates that the object is a newly created object with the necessary identifier mapping to provide the required object binding. Figure 3.4 illustrates graphically the parameter bindings specified for the *sdocumentRelease*1 example.

3.3.4 Acquiring Object Bindings from a Child Scenario

The addition of object visibilities and object creation to the modeling scheme allows the expression of a richer set of models. Specifically the modeling scheme can now express models in which scenarios have interactions with objects which are not pro-

vided explicitly by the scenario's creator. Scenarios can now be specified to interact with newly created objects, and with any object visible to an object provided as a parameter to scenario authorization. However, there are still classes of systems which are difficult to express with the modeling scheme. Consider the case of a parent scenario which authorizes a number of child scenarios. It may be desirable to have an object created in one child scenario take part in the object interactions specified for a subsequent scenario in the parent. This is not expressible in the present scheme because a parent scenario cannot acquire object bindings from a child. Child scenarios acquire object bindings from their parent upon authorization, but there is so far no flow of object visibility in the other direction. Being able to acquire object bindings from a child scenario is the only way in which visibility to newly created objects can be moved back up the scenario creation hierarchy.

This feature also helps to support information hiding [Par72] in the decomposition of scenarios in the requirements or design model being specified, in the following way. With the exception of newly created objects, all objects which take part in the interactions of a child scenario are theoretically visible to the creator of that scenario, as are the objects of its children and its children's children, etc. This is because the parent scenario can see all the objects it provided to the child scenario as parameters. It can also use any of the indirect object visibilities those parameters have in the same way the child can. However, the interactions that take place in a child scenario should be a secret of that scenario. The internal objects which participate in the interactions and the mechanism by which their visibility is acquired should also be a secret. The use of an object by a child scenario may be possible by exploiting visibilities associated with an external parameter object. It is not appropriate for the parent to do so as well in order to gain access to an object being used by the child. This is because the parent scenario would have to understand the nature of the object visibility and therefore something of the nature of the child scenario's

interactions. This is information about the implementation of a child scenario and should be protected. If a child scenario needs to identify some object for use by the parent there should be a mechanism which does not require the parent scenario to understand how the child came to have visibility of the object.

The binding mechanism, as discussed so far, is for object bindings to be made at scenario authorization time. The bindings have two sources. The members of $acquiredpara_S$ are specified when the scenario is authorized using bindings supplied by the scenario's parent. As well, after authorization the members of $definedpara_S$ specified for the new scenario type can be used to refer to objects in the context of the new scenario. Another mechanism is now introduced such that bindings can be made at a child scenario's termination. *I.e.,* a scenario can acquire bindings after its own creation from its children as they terminate. For some scenario ϕ of type Φ, $acquiredpara_S(\phi)$ is extended to include both those bindings acquired from ϕ's parent and those bindings acquired from ϕ's children. To separate the bindings acquired from the parent and those acquired from a child scenario the set of acquired bindings, $acquiredpara_S(\phi)$ is divided into two disjoint subsets, $childacquiredpara_S(\phi)$ and $parentacquiredpara_S(\phi)$.

For scenario ϕ, the set $para_S(\phi)$ is decomposed into three subsets: $childacquired-para_S(\phi)$, $parentacquiredpara_S(\phi)$, and $definedpara_S(\tau_S(\phi))$. The basis of decomposition for $para_S(\phi)$ is the source of the object bindings for a scenario. For a scenario type Φ the definition of parameter type bindings is decomposed into an external component and and internal component, $paraext_{ST}(\Phi)$ and $paraint_{ST}(\Phi)$ respectively. The basis of the decomposition is different in this case. The basis of decomposition for $para_{ST}(\Phi)$ is information hiding.

There are some parameter roles that are used directly by the parent of a scenario of type Φ. There are other parameter roles that are secrets of scenarios of that type. This decomposition is further refined to reflect the effects of child-acquired bindings.

The internal bindings specified by parameter definitions are separated from those which are to be acquired from a child scenario. The set of parameter type bindings, $paraint_{ST}(\Phi)$ is divided into two subsets, $definedpara_{ST}(\Phi)$ and $childpara_{ST}(\Phi)$.

The parameter type bindings specified in $paraext_{ST}(\Phi)$ define the kinds of objects which are presented as the external interface to a scenario of type Φ. *I.e.,* the objects provided by a parent scenario as parameters for the authorization of a scenario of type Φ or those objects passed back to the parent as the scenario terminates. To separate the external bindings acquired from the parent when a scenario of type Φ is authorized and those provided to the parent when the scenario terminates the set of parameter type bindings, $paraext_{ST}$ is divided into two disjoint subsets, $parain_{ST}(\Phi)$ and $paraout_{ST}(\Phi)$. The actual objects which fill the $parain_{ST}(\Phi)$ parameter roles for an instance of a scenario are provided as before, by the parent at authorization time. The actual objects which fill the $paraout_{ST}(\Phi)$ parameter roles are defined by $definedpara_S(\Phi)$ at scenario termination time.

$parain_{ST}$	a function defining the *in* parameter type bindings associated with a scenario type; $parain_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $parain_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$
$paraout_{ST}$	a function defining the *out* parameter type bindings associated with a scenario type; $paraout_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $paraout_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$
$definedpara_{ST}$	a function defining the internal parameter type bindings associated with roles specified by defined parameters; $definedpara_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $definedpara_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$

$childpara_{ST}$ a function defining the internal parameter type bindings associated with roles specified by terminating child scenarios; $childpara_{ST} : \mathcal{ST} \rightarrow 2^{\mathcal{PTB}}$, such that for $\Phi \in \mathcal{ST}$, $childpara_{ST}(\Phi) : \mathcal{I} \rightharpoonup \mathcal{OT}$

$parentacquiredpara_S$ a function defining the parameter bindings associated with a scenario that are acquired from the scenario's parent; $parentacquiredpara_S : \mathcal{S} \rightarrow 2^{\mathcal{PB}}$, such that for $\phi \in \mathcal{S}$, $parentacquiredpara_S(\phi) : \mathcal{I} \rightharpoonup \mathcal{O}$

$childacquiredpara_S$ a function defining the parameter bindings associated with a scenario that are acquired from the scenario's children; $childacquiredpara_S : \mathcal{S} \rightarrow 2^{\mathcal{PB}}$, such that for $\phi \in \mathcal{S}$, $childacquiredpara_S(\phi) : \mathcal{I} \rightharpoonup \mathcal{O}$

for $\Phi \in \mathcal{ST}$, $\phi \in \mathcal{S}$:

$$paraext_{ST}(\Phi) = parain_{ST}(\Phi) \cup paraout_{ST}(\Phi)$$

$$paraint_{ST}(\Phi) = definedpara_{ST}(\Phi) \cup childpara_{ST}(\Phi)$$

$$acquiredpara_S(\phi) = childacquiredpara_S(\phi) \cup parentacquiredpara_S(\phi)$$

$$parain_{ST}(\Phi) \cap paraout_{ST}(\Phi) = \{\}$$

$$definedpara_{ST}(\Phi) \cap childpara_{ST}(\Phi) = \{\}$$

$$childacquiredpara_S(\phi) \cap parentacquiredpara_S(\phi) = \{\}$$

The relationships between the described parameter sets are illustrated in Figure 3.5. The diagram indicates the subset relationships and the identifier dependencies between the sets. The subsets of the $para_{ST}(\Phi)$ hierarchy specify the type of objects which may participate in a scenario of type Φ. This structure is static for a system. Since the scenarios for a system describe all its permitted object interactions, this hierarchy restricts the ways in which types of objects can interact in a

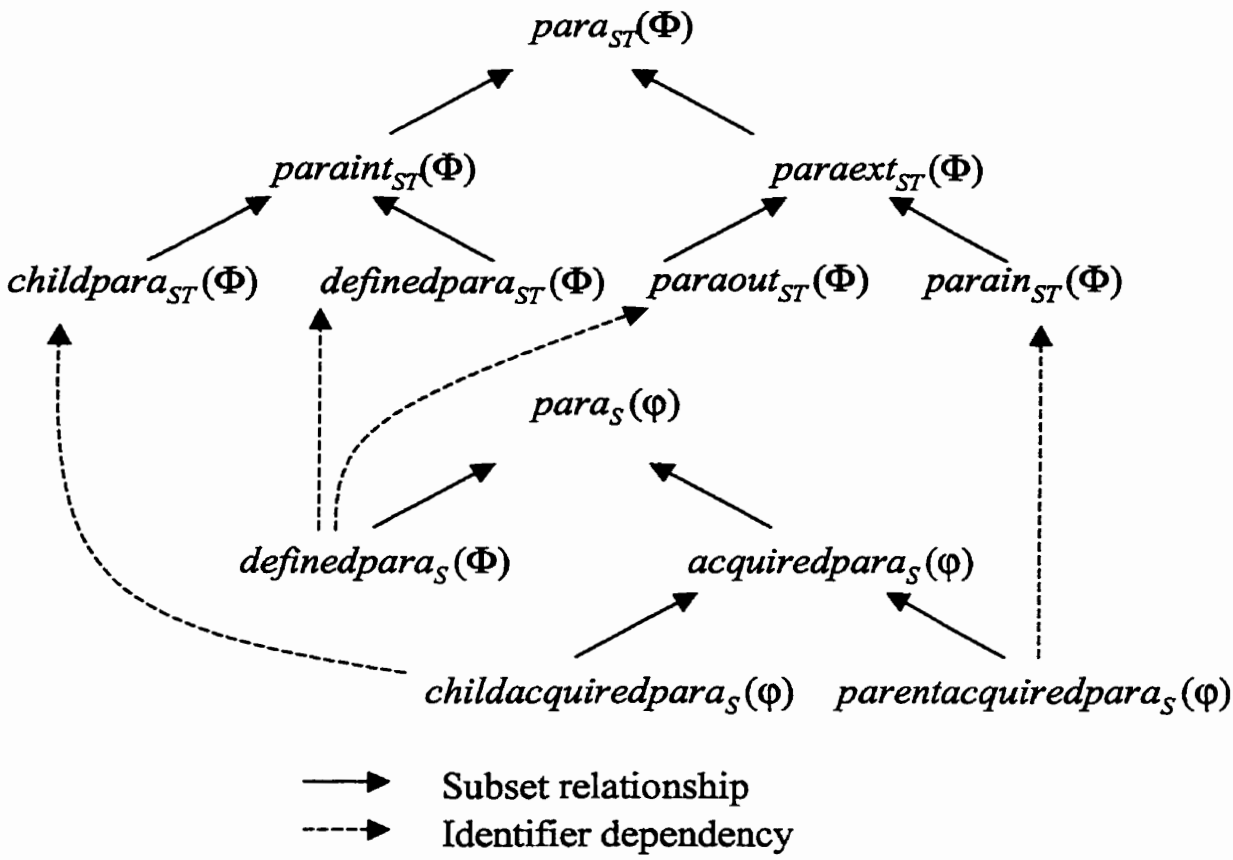

FIGURE 3.5. Relationship between $para_{ST}$ and $para_S$ modified for object bindings acquired from child scenarios, where $\tau_S(\phi) = \Phi$

given system. The subsets of the $para_S(\phi)$ hierarchy provide a mechanism for modeling the relationships between an instance ϕ of a scenario of type Φ and its parameter object instances as a system evolves. An object parameter in ϕ bearing some specific identifier (*i.e.* a member of the $para_S(\phi)$ hierarchy) may exist iff there exists in Φ an object type parameter bearing the same identifier (*i.e.* a member of the $para_{ST}(\Phi)$ hierarchy). This is an identifier dependency. As always, the type of an object instance in $para_S(\phi)$ must agree with the object type specified in $para_{ST}(\Phi)$ for its role.

The identifier dependencies between $parentacquiredpara_S(\phi)$ and $parain_{ST}(\Phi)$, and between $childacquiredpara_S(\phi)$ and $childpara_{ST}(\Phi)$ indicate that they must have the same sets of binding identifiers. *I.e.,* a scenario acquires an object from its parent for each of the bindings specified in $parain_{ST}(\Phi)$ and a scenario acquires an object from a child for each of the bindings specified in $childpara_{ST}(\Phi)$. The $parentacquired-para_S(\phi)$ bindings are made at authorization time and the $childacquiredpara_S(\phi)$ bindings are made as the scenario evolves and child scenarios terminate.

The identifier dependency between the set $definedpara_S(\Phi)$ and the sets $defined-para_{ST}(\Phi)$ and $paraout_{ST}(\Phi)$ indicates that for every binding identifier in the union of $definedpara_{ST}(\Phi)$ and $paraout_{ST}(\Phi)$ there must be a parameter definition with the same identifier in $definedpara_S(\Phi)$ and *vice versa*.

The construction of scenario descriptors is modified to accommodate mappings for both the $parain_{ST}$ parameters and the $paraout_{ST}$ parameters of a new child scenario. A scenario descriptor is now a 4-tuple. The first and last elements of the 4-tuple are as before.

The set of identifier mappings provided as the second element of the tuple is the $parain_{ST}$ parameter mapping. The mapping behaves as before. The set of identifier pairs maps identifiers in the context of a parent scenario to identifiers in the context of a child scenario being authorized, at the time the child scenario is being authorized. New parameter bindings are created, which bind role identifiers for the child scenario

to the identified objects participating in the parent scenario. These new parameter bindings become members of the $parentacquiredpara_S$ function for the new child scenario. Again, for a scenario descriptor defining a scenario of type Φ, there must be an identifier mapping provided for each identifier role defined in set $parain_{ST}(\Phi)$.

The set of identifier mappings provided as the third element of a scenario descriptor 4-tuple contains $paraout_{ST}$ parameter mappings. These behave in the following way. As before, the set of identifier pairs maps identifiers in the context of a parent scenario to identifiers in the context of a child scenario. However, in this case the new parameter bindings bind role identifiers of the parent scenario to objects participating in the child scenario. The bindings are created at the time the child terminates. The newly created parameter bindings become members of the $childacquiredpara_S$ function for the parent scenario. For a scenario descriptor defining a new scenario of type Φ, there must be an identifier mapping provided for each identifier role defined in set $childpara_{ST}(\Phi)$.

The following definition formalizes the construction of the modified scenario descriptor.

$$\mathcal{SD} \qquad \text{Set of scenario descriptors, } \mathcal{ST} \times 2^{\mathcal{I}\times\mathcal{I}} \times 2^{\mathcal{I}\times\mathcal{I}} \times 2^{1}$$

With these modifications to the modeling scheme, visibility of an object provided by an 'out' parameter when a scenario terminates can be used by scenarios created after that termination. The visibility of that object is also available to pass on to the parent's parent scenario by providing it again as an 'out' parameter.

The document release example as specified in Chapter 2 does not include a case where a scenario acquires an object binding from a child. To illustrate such a case the scenario type *SdocRelease* can be modified so that it provides an object binding to its parent scenario. The object playing the role *relAuth* will now be provided as a $paraout_{ST}$ parameter. In the existing example there is already a parameter definition

for this role because the object bound to this identifier is created by the scenario. Because the object is currently an internal object for the scenario (hidden by the scenario) the parameter type binding for this role is currently in $definedpara_{ST}$. With the proposed modification the object bound to the *relAuth* role will become available to the parent (an external object). This is achieved by moving the parameter type binding for this role to $paraout_{ST}$. In the example, scenario type *SreleaseDecision* specifies the authorization of a scenario of type *SdocRelease* (*i.e.* it is a parent for *SdocRelease*). With the proposed modification to the example, *SreleaseDecision* will have to specify a mapping for a *relAuth* object in the $paraout_{ST}$ parameter mappings of its scenario descriptor for its *SdocRelease* child. A specification for the two scenario types is as follows.

$$SreleaseDecision \in \mathcal{ST}$$
$$parain_{ST}(SreleaseDecision) =$$
$$\{(reviewer,\ OpatentOfficer), (report, Odocument)\}$$
$$childpara_{ST}(SreleaseDecision) = \{(releaseAuth, OreleaseAuth)\}$$
$$child_{ST}(SreleaseDecision) =$$
$$\langle (SdocReview, \{(reviewer, reviewer), (report, report)\}, \{\}, False),$$
$$(SdocRelease, \{(reviewer, reviewer), (report, report)\},$$
$$\{(releaseAuth, relAuth)\}, False),$$
$$(SdocRevision, \{(reviewer, reviewer), (report, report)\}, \{\}, False)\rangle$$
$$order_{ST}(SreleaseDecision) = or$$

$$SdocRelease \in \mathcal{ST}$$
$$parain_{ST}(SdocRelease) =$$
$$\{(reviewer,\ OpatentOfficer), (report, Odocument)\}$$
$$paraout_{ST}(SdocRelease) = \{(relAuth, OreleaseAuth)\}$$

$definedpara_{ST}(SdocRelease) = \{(reportAuthor, Oscientist)\}$

$definedpara_{S}(SdocRelease) =$

$\{(reportAuthor, report.author),$

$(relAuth, create(OreleaseAuth, \{(report, relDoc)\}))\}$

$child_{ST}(SdocRelease) =$

$\langle(Pcreate, \{(reviewer, sender), (relAuth, receiver), (report, doc)\},$

$\{\}, False),$

$(PrelPerm, \{(reviewer, sender), (reportAuthor, receiver),$

$(relAuth, authorization)\}, \{\}, False),$

$(PrelDoc, \{(reportAuthor, sender), (relAuth, receiver)\}, \{\}, False),$

$(Pread, \{(relAuth, sender), (report, receiver)\}, \{\}, False)\rangle$

$order_{ST}(SdocRelease) = seq$

Chapter 4
Safety Analysis

4.1 Introduction

In the safety analysis of a system model it is important to be able to determine whether or not it is possible for a certain message to be authorized and in what order it can be sent relative to other messages. The existence and ordering of messages in the model can be analyzed by inspection of the scenario tree. This may not be computationally feasible for large scenario trees but it can form the conceptual basis of an analysis method. In any state the leaves of the tree will be either primitive scenarios or scenarios which are authorized to perform some action. Child scenarios are added to the $child_S$ sequence of the parent in order (denoted here as left to right) as they are authorized. The intuition for examining message sequencing is to perform a depth-first left-to-right search of the scenario tree. The order in which primitive scenarios are discovered is the order of messages as the system evolves. This intuition works well for scenarios with *seq* ordering. The definition of the message ordering relation must be modified to capture the semantics of all three scenario ordering properties (*seq*, *or*, and *and*).

The rest of this chapter starts with a definition of how security properties of systems are modeled using scenario-based access control. The next section defines the authorization properties for a specific evolution of system execution. Equivalence will be defined for system states based on the authorization properties associated with the states. It will be shown that for any system, there exists a maximal system state that describes all possible authorizations that can be generated by the system and all possible orderings. An algorithm is presented which provides for the construction of

a scenario tree which is a maximal system state. Inspection of this maximal scenario tree provides safety analysis for the system being modeled.

4.2 Modeling a System

Recall from the scenario modeling scheme that an evolution of execution for a system depends on the authorization of scenarios. Each message between objects is authorized by an instance of a primitive scenario type. These primitive scenario instances are authorized as children to a parent scenario which specifies a context for their interaction. The parent in turn is authorized as a child to another scenario; and so on back to some initial scenario. When a parent scenario becomes authorized to create a new scenario, this authorization is associated with a scenario descriptor. The scenario descriptor specifies the type of the new scenario, parameter mappings from the parent to the child, and whether or not there is concurrency. The new scenario is then authorized to perform some action(s) as specified by its scenario ordering, and its own set of scenario descriptors. As new scenarios are authorized they are added, in order, to the $child_S$ sequence of the parent. Thus at any point in the evolution of a system model, the scenarios form a tree structure.

The evolution of a system model is restricted by the security policy for the system and the initial state of the model. A security policy, $\mathcal{P}$, is defined by the sets $\mathcal{OT}$, and $\mathcal{ST}$, and the functions $para_{ST}$, $definedpara_S$, $child_{ST}$, $order_{ST}$, and vis_{OT}. The security policy is static. It is defined only in terms of type and is independent of any actual objects, messages, or scenarios in a specific instance of a system. Formally a security policy is a 7-tuple.

$$\mathcal{P} = (\mathcal{OT}, \mathcal{ST}, vis_{OT}, para_{ST}, definedpara_S, child_{ST}, order_{ST})$$

For some security policy a *protection state* (or *state*) is defined by the membership

of the sets $\mathcal{S}$ and $\mathcal{O}$, and the functions specifying the attributes of subjects and objects in those sets. Formally a protection state is a member of the set $\mathcal{V}$, where

$$\mathcal{V} \qquad \text{set of states, } \mathcal{O} \times \mathcal{S} \times \tau_O \times \tau_S \times vis_O \times para_S \times child_S \times con_S$$

The evolution of the system proceeds from an initial state with the occurrence of security relevant actions. The security relevant actions defined for the modeling scheme are the authorization of a new scenario, the authorization of a message pass between two specific objects, and the sending of a message. In modeling protection state it is not necessary to model the three kinds of action explicitly. Protection state can be modeled by considering the only authorization of new scenarios explicitly. The effects of the other actions is captured implicitly.

Recall from the discussion in Section 3.2.4 that for access-control modeling purposes, there is no loss in expressiveness in only considering primitive scenario types and leaving the messages as implicit entities to be defined in the implementation of the access-control mechanism. This can be done because the modeling scheme is concerned with modeling protection state and not with modeling the actual execution of a system. The actual ordering of the message events with respect to other events in the evolution of an actual system is not relevant. What is relevant is the *possible* ordering of the message events. The authorization properties defined in Section 4.3 model the ordering of the message events by considering when the associated primitive scenario becomes authorized.

The occurrence of a scenario authorization moves the system from one state to a new state. A scenario authorization involves the creation of a new scenario, and possibly the creation of new objects as scenario parameters. The creation of scenarios and objects adds members to the sets $\mathcal{S}$ and $\mathcal{O}$, and adds new mappings to the functions specifying the attributes of subjects and objects in those sets.

The convention of superscripting a set or function identifier with a state will be used to identify the evolution context being considered. For example, $\mathcal{S}^g$ and $para_S^h$ signify the set $\mathcal{S}$ and the function $para_S$ in the states g and h respectively. State 0 is used to signify the initial state. Where there is no confusion $\mathcal{S}^0$ may be used to denote the initial scenario.

The initial state for the model of a system is defined by the initial membership of the sets $\mathcal{O}$ and $\mathcal{S}$, the type functions τ_O and τ_S, and the functions $para_S$ and vis_O. In the initial state there is one and only one member of $\mathcal{S}$. This is the initial scenario for the model. $child_S$ is initially null for the initial scenario and con_S is defined as *true*. Thus, a system model is an implementation of a security policy for some specific set of objects involved in some initial scenario. Formally a system, Σ, is a 7-tuple.

$$\Sigma = (\mathcal{P}, \mathcal{O}^0, \mathcal{S}^0, \tau_O^0, \tau_S^0, para_S^0, vis_O^0)$$

An evolution of system execution is represented by a sequence of system states. There is a specific instance of a scenario tree associated with a system state. Therefore, the evolution of the protection state of a system can thought of as an evolving scenario tree. A scenario tree defines a *history* for a system in a specific state. The evolutions of execution possible for a system, beginning at its initial state are its *possible histories.* A history captures all the actions that have taken place in the system evolution. If a scenario or object creation action takes place as a result of a scenario authorization it is said to be *permitted.* When a system state evolves such that all scenario and object creations giving rise to the state are permitted, the state is said to be a *derivable state* and the associated scenario tree a *derivable scenario tree.*

4.3 Authorization Properties

When considering what is permitted by a system, the kinds of messages that may be authorized, and their order, must be determined. The message ordering relation for a

scenario tree is a pre-order of the authorized messages defined by the tree. Messages on a scenario tree are always associated with a primitive scenario. The authorization properties of the tree can therefore be described by a pre-order on the primitive scenarios present in the tree.

As described in the previous section, when *seq* orderings for scenario authorization are considered, a depth-first left-to-right search produces an acceptable ordering. This is because the $child_S$ sequence is defined be a left-to-right ordering of scenario authorization. In the *seq* ordering one child scenario must terminate before the next scenario is authorized. Recursively, the entire sub-tree of scenarios must therefore have terminated before the next scenario is authorized.

This intuition does not hold for scenario trees with *or* and *and* scenario orderings or with concurrent scenarios. In the case of *and* scenario orderings, all child scenarios are authorized at the same time. It is not a requirement for this ordering that one child scenario terminate before another may begin. This means that the messages associated with the sub-tree of one child scenario do not necessarily precede messages associated with the sub-tree of another child scenario. The situation is similar for concurrent scenarios. The concurrent child scenario begins a new thread of scenario authorization orderings that continues in parallel with the original thread of scenario authorization orderings. So again, messages associated with the sub-tree of the child scenario do not necessarily precede messages associated with the sub-tree of the parent scenario after the action authorizing the child. The difference between *and*-ordered scenarios and concurrent scenarios is that an *and*-ordered scenario is not able to terminate until all its children have terminated, while in the case of a parent with one or more concurrent children, the termination state of the children has no bearing on the termination state of the parent. *I.e.,* the parent authorizes each child and carries on with its specified ordering of actions without regard to the termination of the concurrent child.

The case for *or* scenario orderings is different. In this case the child scenarios of the parent are mutually exclusive. Only one child scenario may generate messages so the messages associated with the sub-tree of one child scenario are not comparable to messages associated with the sub-tree of another child scenario. In actual derivable scenario trees only one sub-tree can exist so it may not seem important to consider how two mutually exclusive sub-trees are related. However, it is an advantage later to be able to consider non-derivable scenario trees during analysis so the rules for such associations are defined when considering authorization ordering.

The following definitions and rules describe the pre-order on permitted authorizations more formally. For a state h,

PA	set of primitive scenarios permitted to be authorized by a system, $PA = \{\phi \in \mathcal{S}^h \mid \phi \text{ is a primitive scenario}\}$
R	scenario pre-order relation on PA, $R \subseteq PA \times PA$

For primitive scenarios $\phi, \psi \in PA$, consider their respective paths through the tree hierarchy to the tree root, $\mathcal{S}^0$. At some point the paths to root for ϕ and ψ must join. Say they join at some scenario σ. $\phi R \psi$ denotes that the message associated with ϕ may precede the message associated with ψ.

1. If the scenario ordering of σ is *or* and ϕ and ψ belong to sub-trees of different children of σ then ϕ and ψ are not comparable.

2. If the scenario ordering of σ is *and* then $\phi R \psi$ and $\psi R \phi$.

3. If the scenario ordering of σ is *seq* then:

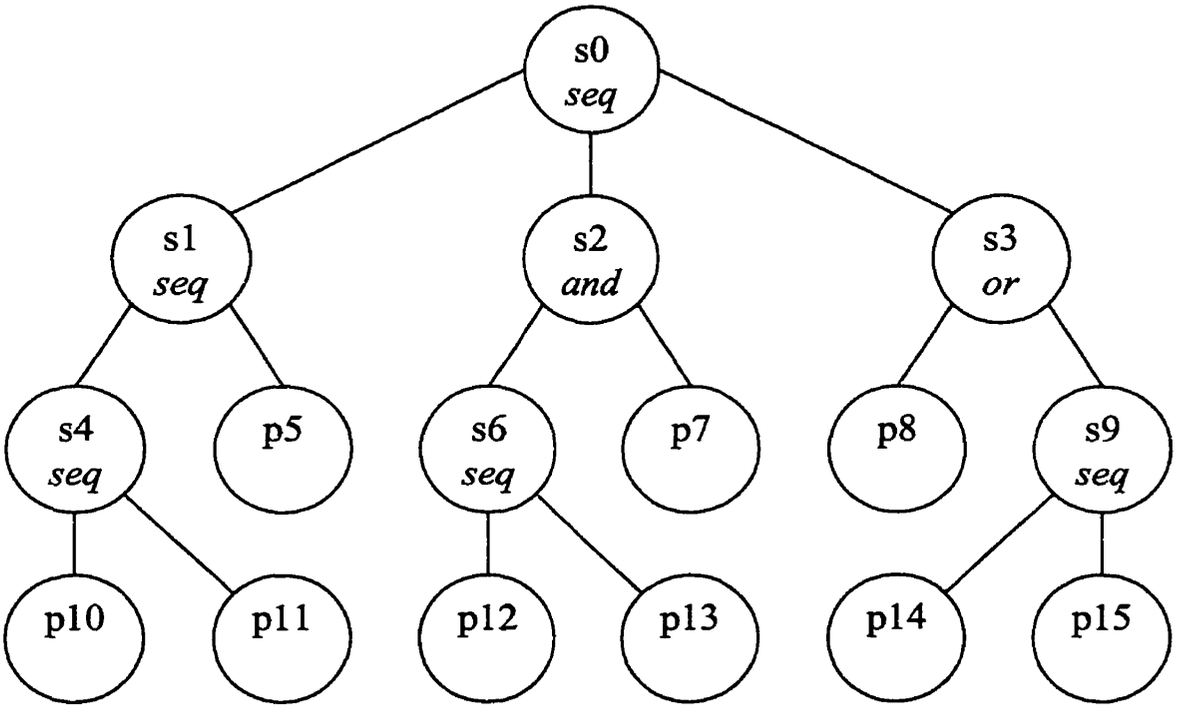

FIGURE 4.1. Scenario ordering example

(a) $\phi R \psi$ if for $\alpha_i, \alpha_j \in child_S(\sigma)$, ϕ belongs to the sub-tree rooted at scenario α_i and ψ belongs to the sub-tree rooted at scenario α_j, and α_i precedes α_j in $child_S(\sigma)$;

(b) $\psi R \phi$ if for $\alpha_i, \alpha_j \in child_S(\sigma)$, ϕ belongs to the sub-tree rooted at scenario α_i and ψ belongs to the sub-tree rooted at scenario α_j, and α_i precedes α_j in $child_S(\sigma)$ and there is a concurrent scenario on the path to root from ϕ to σ.

It is apparent that PA and R are monotonic with respect to a system history. Scenarios are added to the scenario tree but are not removed.

As an example consider the scenario tree presented in Figure 4.1. In the example primitive scenarios are denoted pi and non-primitive scenarios are denoted sj. The

scenario ordering has been specified for non-primitive scenarios. s0 is the initial scenario (root). The following are some of the authorization properties that apply to this example. The scenarios p10, p11, and p5 will always be the first primitive scenarios authorized by the system and the authorizations will occur in that order. These scenarios will always precede the scenarios associated with the sub-trees of scenario s2 and s3. Scenario p12 will always precede scenario p13. Scenario p10 will always precede scenario p12. All of these cases involve a *seq* ordering at the scenario where the paths to root join, and a depth-first left-to-right ordering applies. In the case of scenarios p7 and p12 their paths to root join at scenario s2, which has a *and* ordering. This means p7 and p12 can come in either order with respect to each other. The same applies for p7 and p13. In the case of p8 and p14 their paths to root join at scenario s3, which has an *or* ordering. In a derivable scenario tree p8 and p14 could not both occur because the permitted possible histories will allow only the actions associated with either the sub-tree of p8 or the sub-tree of s9. Therefore scenarios p8 and p14 are incomparable under the pre-order relation R.

If it was the case that s4 was a concurrent scenario then p10 and p11 may be delayed for some arbitrary length of time. Therefore, this would allow p5 to proceed before p10 and p11. It is also possible in this case that p10, or p10 and p11 could occur before p5.

4.4 Safety Analysis

4.4.1 Scenario Equivalence

Consider the safety problem. The objective is to determine in a given situation whether or not a subject can acquire a particular access right to an object. When considering the safety problem in the context of scenario-based access-control models the basic element of access control is based on a primitive scenario describing exactly

one message pass. Therefore, a solution to the safety problem must identify the primitive scenarios that may be generated as a system evolves. The safety question can be formulated by the following definition.

Definition 1. *The simple saftey question is defined as: for some system,* Σ, $\Phi \in \mathcal{ST}$, $\{o_1, o_2, \ldots, o_n\} \in \mathcal{O}^0$, *is there a possible history such that there exists a state* h, *a scenario* ϕ *and* $\phi \in PA^h$, *where* $\tau_S(\phi) = \Phi$ *and* $parentacquiredpara_S(\phi) = \{(id_1, o_1), (id_2, o_2), \ldots, (id_n, o_n)\}$

If such a question can be answered it could be used to identify the occurrence of a specific primitive scenario (*i.e.* authorization of a specific message type). The sending and receiving objects of the message instance are specified by the scenario's parent-acquired parameter bindings (*i.e.* the objects bound to the identifiers *sender* and *receiver*). All other parameters to the message that make it unique are also specified by the object bindings.

For non-monotonic security policies the ordering of messages must also be considered. In such cases a version of the safety question which accounts for message ordering can be formulated by the following definition.

Definition 2. *The saftey question for non-monotonic security policies is defined as: for some system,* Σ, $\{\Phi, \Psi\} \in \mathcal{ST}$, $\{o_1, o_2, \ldots, o_n, o'_1, o'_2, \ldots, o'_m\} \in \mathcal{O}^0$, *is there a possible history such that there exists a state* h, *scenarios* ϕ *and* ψ, $\{\phi, \psi\} \in PA^h$, *and* $\phi R^h \psi$, *where* $\tau_S(\phi) = \Phi$, $\tau_S(\psi) = \Psi$ *and* $\text{parentacquiredpara}_S(\phi) = \{(id_1, o_1), (id_2, o_2), \ldots, (id_n, o_n)\}$ *and* $\text{parentacquiredpara}_S(\psi) = \{(id'_1, o'_1), (id'_2, o'_2), \ldots, (id'_n, o'_m)\}$

Note that the specified parameters in both of these definitions are members of $\mathcal{O}^0$. A safety question is formulated in terms of objects that exist in the initial state.

Objects that may be created during the evolution of the system do not yet exist and cannot be named. The types of objects that might be involved in system interactions may be interesting but only $\mathcal{O}^0$ objects can be specified for a particular analysis.

Many different evolutions of the system are possible in which different objects are created after the initial state. Similar scenarios can be repeated a number of times. However, many of the states produced may agree on the existence and ordering of messages with respect to how $\mathcal{O}^0$ objects are involved.

For example, in the document release example an initial state might include an *Oscientist* object, an *Odocument* object, and an *OpatentOfficer* object. The scientist author is allowed to read then write to the document. Then the author makes a decision whether to continue to edit the document or to forward the document for review. The scenario creation and ordering constraints for the scenario types *Sinitial*, *SdocEdit*, and *SforwardDecision* allow an arbitrary number of creations of a *SdocEdit* scenario instance (the scenario which authorizes the read and write messages) before proceeding to document review. After the author has performed a read/write on the document a few times any further instances of *SdocEdit* do not add any messages that involve $\mathcal{O}^0$ objects in new or different ways. As well, similar kinds of messages involving $\mathcal{O}^0$ objects are not presented in a new order by the repeated scenario instances. Therefore, only a limited number of instances of *SdocEdit* are interesting from a safety analysis perspective.

Messages can be considered to be equivalent when reduced to their relationship to $\mathcal{O}^0$ objects. That is, they can be considered equivalent if they treat $\mathcal{O}^0$ objects in the same way. 0-reducible equivalence will be defined for messages by defining 0-reducible equivalence on primitive scenarios. 0-reducible equivalence for primitive scenarios is subsumed by the definition of 0-reducible equivalence for scenarios in general. 0-reducible equivalence will also be defined for other model constructions.

Often when it is not confusing *0-reducible* will be dropped and model constructions will be referred to as being equivalent.

Before defining 0-reducible equivalence for scenarios it is necessary to define 0-reducible equivalence for objects. The messages authorized by a scenario tree can depend on the object visibilities of the objects participating in the scenarios. If objects of the same type have the same direct and indirect visibility of $\mathcal{O}^0$ objects they are said to be 0-reducible equivalent.

The intuition behind object equivalence follows from the fact that object visibilities are specified at object creation time. The visibilities therefore refer to objects existing at the time of creation. The transitive visibility relationships for an object therefore form a finite tree structure. If two objects of the same type have the same visibility tree topology and the same $\mathcal{O}^0$ objects occupying the same positions in their respective trees, then the objects are 0-reducible equivalent. $\mathcal{O}^0$ objects are trivially equivalent to themselves. Object equivalence is formally defined as follows.

Definition 3. *Two objects a and b are 0-reducible equivalent, written $a \equiv_0 b$, iff $\tau_O(a) = \tau_O(b)$ and either,*

1. *$a, b \in \mathcal{O}^0$ and $a = b$, or*
2. *$a, b \notin \mathcal{O}^0$, and for each parameter binding $(i_a, c_a) \in vis_O(a)$, there exists a parameter binding $(i_b, c_b) \in vis_O(b)$ such that $i_a = i_b$ and $c_a \equiv_0 c_b$.*

The definition of object equivalence is recursive. For any object the depth of the recursion is finite. This can be proven inductively by noting that all objects are either $\mathcal{O}^0$ objects or are created during system evolution.

Lemma 4. *For a system Σ, and any two objects $a, b \in \mathcal{O}$, the definition of $a \equiv_0 b$ is finite.*

Proof. The fan out of any node in the visibility tree is finite because the size of vis_{OT} for any object type is defined to be finite. The proof of finite depth of recursion of the definition is by induction on the number of object creations which have taken place in the system history. The inductive hypothesis is that for any system state in which n new objects have been created the lemma holds. The basis case holds trivially. In a state in which the $n+1$st object is created its equivalence to another object is defined by determining an equivalence for the finite set of objects in its visibility set, vis_O. These objects must have been created before the $n+1$st object. By the induction hypothesis the lemma holds for each of these objects. Therefore the lemma holds for systems with $n+1$ object creations. □

Now 0-reducible equivalence of scenarios can be defined. Scenarios are 0-reducible equivalent iff they agree on type and their parameter bindings are equivalent. Scenario equivalence is formally defined as follows.

Definition 5. *Two scenarios ϕ and ψ are equivalent, written $\phi \equiv_0 \psi$, iff $\tau_S(\phi) = \tau_S(\psi)$ and for each parameter binding $(i_\phi, a_\phi) \in para_S(\phi)$, there exists a parameter binding $(i_\psi, a_\psi) \in para_S(\psi)$ such that $i_\phi = i_\psi$ and $a_\phi \equiv_0 a_\psi$.*

Given this definition, three useful properties of scenario equivalence are presented as lemmas. The first property is that equivalent scenarios will produce equivalent child scenarios. The child scenarios may evolve in different ways, but they will be equivalent at the time of creation. This follows because by definition the construction of child scenarios depends only upon the type of the parent scenario and the parent's parameter bindings. The property that equivalent scenarios will produce equivalent child scenarios is proven for the following lemma.

Lemma 6. *For a system Σ, and scenarios ϕ and ψ, if $\phi \equiv_0 \psi$ at the time of their ith child creations, ϕ' is the scenario produced using the ith scenario descriptor in*

$child_{ST}(\tau_S(\phi))$, and ψ' is the scenario produced using the ith scenario descriptor in $child_{ST}(\tau_S(\psi))$, then $\phi' \equiv_0 \psi'$.

Proof. From the equivalence of ϕ and ψ, $\tau_S(\phi) = \tau_S(\psi)$. Therefore ϕ and ψ share the ith scenario descriptor and $\tau_S(\phi') = \tau_S(\psi')$. Recall that $para_S(\phi') = parentacquiredpara_S(\phi') \cup childacquiredpara_S(\phi') \cup definedpara_S(\phi')$ and similarly for $para_S(\psi')$. Bindings in $parentacquiredpara_S(\phi')$ and $parentacquiredpara_S(\psi')$ are defined by identifier mappings in the shared scenario descriptor and the sets $para_S(\phi)$ and $para_S(\psi)$ respectively. By the equivalence of ϕ and ψ, if these mappings must generate bindings $(i_\phi, a_\phi) \in para_S(\phi')$ and $(i_\psi, a_\psi) \in para_S(\psi')$, then $i_\phi = i_\psi \rightarrow a_\phi \equiv_0 a_\psi$. At the time of creation of ϕ' and ψ', $childacquiredpara_S(\phi') = childacquiredpara_S(\psi') = \{\}$. Since $\tau_S(\phi') = \tau_S(\psi')$, $definedpara_S(\tau_S(\phi')) = definedpara_S(\tau_S(\psi'))$. Therefore, an object defined by a $definedpara_S$ binding for ϕ' will be equivalent to an object defined for ψ' using the same binding. This follows from $parentacquiredpara_S(\phi') \equiv_0 parentacquiredpara_S(\psi')$ and Definition 3. Therefore, the lemma holds. □

Scenarios can change state over time and with the change in state their equivalence class may change. More precisely, the mapping specified by $para_S$ for a scenario may change as child-acquired scenario bindings are added, mapping the scenario to a new function. It can now be proven that it is possible for scenarios that are equivalent upon creation to evolve in such a way that they remain equivalent. The property that equivalent scenarios can evolve in equivalent ways is proven for the following lemma.

Lemma 7. *For a system Σ, and scenarios ϕ and ψ, where $\phi \equiv_0 \psi$ at the time of their creation, for any derivable state h there exists a derivable state g such that $\phi^h \equiv_0 \psi^g$.*

Proof. The proof is by induction on the depth of the scenario sub-tree rooted at ϕ. The induction hypothesis is that for a derivable state h where the depth of

the scenario sub-tree rooted at ϕ is n there exists a derivable state g such that for any $(i, a) \in childacquiredpara_S^h(\phi)$ there exists $(i, a') \in childacquiredpara_S^g(\psi)$ and $a \equiv_0 a'$. The basis case for sub-tree depth of 0 holds trivially since there are no children to contribute to $childacquiredpara_S(\phi)$. Consider the case where sub-tree depth equals $n+1$. By Lemma 6, for any child scenario created by ϕ an equivalent scenario can be created by ψ. By the induction hypothesis there exists a state g such that these children have equivalent termination states. Therefore an equivalent child acquired binding is available to ψ as required and the lemma holds. □

Another useful property associated with the equivalence of scenarios arises from the observation that equivalent scenarios should have an equivalent set of possible histories. By equivalent possible histories it is meant that it should be possible to derive scenario sub-trees from equivalent scenarios such that the sub-trees rooted at those scenarios have the same topology and equivalent scenarios at each position in the sub-trees. The property that equivalent scenarios can evolve equivalent sub-trees is proven for the following lemma.

Lemma 8. *For a system Σ, and scenarios ϕ and ψ, if $\phi \equiv_0 \psi$, and ϕ' is a scenario in a derivable sub-tree rooted at ϕ, then for the sequence $(\phi, \omega_1, \omega_2, \ldots, \omega_m, \phi')$ describing the path in the sub-tree from ϕ' to ϕ, it is possible to derive a state such that there exists a sub-tree of ψ which contains a path defined by the sequence $(\psi, \omega_1', \omega_2', \ldots, \omega_m', \psi')$ and $\omega_i \equiv_0 \omega_i'$ for all $i \in 1 \ldots n$, and $\phi' \equiv_0 \psi'$.*

Proof. The proof is by induction on the length of the sequence describing the path in the sub-tree from ϕ' to ϕ. The induction hypothesis is that the lemma holds for sequences of length n. The basis case for sequences of length 0 holds trivially. Consider the case where the length of the sequence is $n+1$. The sequence would have the form, $(\phi, \omega_1, \omega_2, \ldots, \omega_{n-1}, \phi')$. From the equivalence of ϕ and ψ, Lemma 6, and Lemma 7, a sceanrio ω_1' is derivable such that ω_1' is a child of ψ and $\omega_1' \equiv_0 \omega_1$. By

the induction hypothesis the required sequence $(\omega'_1, \omega'_2, \ldots, \omega'_{n-1}, \psi')$ is derivable and the lemma holds. □

0-reducibility and equivalence are also defined for states. 0-reducibility is defined with respect to the permitted authorization pre-order on PA.

Definition 9. *A state h is 0-reducible to a state g, written $h \leq_0 g$, iff*

1. *for all primitive scenarios $\phi \in PA^h$, there exists $\phi' \in PA^g$ such that $\phi \equiv_0 \phi'$ and*

2. *for all $\phi R^h \psi$, there exists $\phi' R^g \psi'$ such that $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$.*

Two states g and h are 0-reducible equivalent, written $g \equiv_0 h$, iff $h \leq_0 g$ and $g \leq_0 h$.

Note that equivalent states would generate equivalent messages and the possible orderings of equivalent messages with respect to each other would be the same.

4.4.2 Maximal States

Recall that PA and R are monotonic. Scenarios trees grow from an initial scenario. Branches are added as scenario authorization actions occur, moving the system from state to state. Branches are never removed. This is why a history for the system is associated with a specific scenario tree.

For some system Σ, if there exist histories with derivable scenario trees represented by states g and h, such that the scenario tree for state h is an extension of the scenario tree for state g through a series of permitted scenario authorizations, it can be said that h can be derived from g. h can be derived from g will be written $g \rightarrow h$.

It would be useful for safety analysis if there existed some derivable *maximal state* such that for any further messages generated by the system there are already equivalent messages in PA and equivalent pairs in R.

Definition 10. *For a system Σ, m is a maximal state iff m is derivable and for all derivable states h such that $h \rightarrow m$ and all derivable states g such that $m \rightarrow g$, $h \leq_0 m$ and $g \leq_0 m$.*

For any system there may be more than one such maximal state. All maximal states are not necessarily equivalent. The existence of maximal states is proven by the following lemma.

Lemma 11. *For a system Σ, for every derivable state h there exists a maximal state m such that $h \rightarrow m$.*

Proof. Let S^h be the set of states derivable from h. For the case in which $S^h = \{\}$, h is a maximal state derivable from itself in zero steps. The number of equivalence classes of states in S is finite. This is because the number of equivalence classes of primitive scenarios in PA is finite since there are a finite number of scenario types and $\mathcal{O}^0$ objects. There are a finite number of possible orderings (for a representative member of each of these scenario equivalence classes) that can be represented in R. Therefore, for a state h there are a finite number of equivalence classes of derivable states. Let T^h be a set of states such that there is a representative member from each of the equivalence classes of S^h. If for every $g \in T^h$, $g \leq_0 h$, then h is a maximal state. Otherwise there exists some state $g \in T^h$ such that $g \not\leq_0 h$. Consider evolving the system to state g. Form S^g and T^g similarly to S^h and T^h. $S^g \subseteq S^h$. By the monotonicity of PA any state in T^g previously 0-reducible to h is 0-reducible to g. Since the number of elements in T^g must be finite the system can continue to be evolved in a similar way until all derivable equivalence classes can be reduced. This state will be a maximal state for h. □

It is apparent that for any system state, there may be one or more non-equivalent maximal states. This is because as a scenario tree for a system evolves some scenarios

may have orderings of type *or*. These scenarios are decision making scenarios and the resulting possibilities for scenario tree evolution are mutually exclusive. Therefore, as a system evolves, the set of derivable states it can reach may decrease. Different decisions at a scenario with an *or* ordering may lead to different sets of reachable states and therefore to different maximal states which are not equivalent. For example, consider a system in a state for which it is possible to authorize a scenario with an *or* ordering. When this scenario becomes authorized it is possible for further evolution of the system to continue with the actions associated with any one of the child scenarios. In this state there is a possible history associated with each one of the child scenarios. Once an action is taken on one of these possible history paths the other paths are no longer possible. *I.e.,* some of the possible histories of the previous state are no longer possible. A different maximal state may have been associated with each of the mutually exclusive possible histories. Once the system begins to evolve along one of these paths the maximal states associated with the other possible histories may no longer be reachable. *I.e.,* some kinds of messages, or message orderings, may no longer be possible to generate.

It would be more useful for safety analysis if there was a *system maximal state* which would describe all possible occurrences of messages and their respective orderings. Inspection of PA and R for such a state would yield an answer to the safety problem. However, it seems that such a state would not be derivable if there are scenarios with *or* orderings.

Consider construction of a scenario tree for a state in the normal way but allowing authorization of scenarios which are normally mutually exclusive by way of an *or* ordering. Such a state would not be derivable but the set PA would include all messages possible on either mutually exclusive branch of the tree. The message ordering relation R for this new scenario tree would not contain any orderings which were not possible in some derivable scenario tree. This is because, by the definition

of R, messages on the mutually exclusive branches are not comparable and would not contribute new pairs to the relation.

It is possible that such a non-derivable state could be constructed such that all maximal states for a system could be 0-reducible to that state. The existence of such a state is proven for the following theorem.

Theorem 12. *For a system Σ, there is a state m^*, which may not be derivable, such that:*

1. *there exists $\phi', \psi' \in PA^{m^*}$ and $\phi' R^{m^*} \psi'$ where $\phi' \equiv_0 \phi$ and $\psi' \equiv_0 \psi$ if there exists $\phi, \psi \in PA^m$ and $\phi R^m \psi$ for some maximal state m, and*

2. *there exists $\phi', \psi' \in PA^{m^*}$ and $\phi' R^{m^*} \psi'$ only if there exists $\phi, \psi \in PA^m$ and $\phi R^m \psi$ for some maximal state m, where $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$.*

Proof. The proof is by induction over the number of equivalence classes of maximal states in a system. The basis case is one equivalence class of maximal states and in this case m^* can be any member of the equivalence class. Consider a set T which contains one member from each equivalence class. Assume the theorem holds for T of size n and consider T of size $n+1$. $T = S + \{r\}$ where $|S|$ is n. By the induction hypothesis there is a state, m, which satisfies the theorem for S. Create a new state, m' by superimposing the scenario tree for m and the tree for state r. *I.e.,* add to the tree for m any branches which exist in r but do not already exist in m. Adding such branches means that where there is a path between a scenario and root in the tree for r, equivalent scenarios corresponding to that path are created in the tree for scenario m if they do not already exist. This is possible by Lemma 8. For the *if* case, the induction step must show that for m (or r) , such that $\phi, \psi \in PA^m$ (or $\in PA^r$) and $\phi R^m \psi$ (or $\phi R^r \psi$), there exists $\phi', \psi' \in PA^{m'}$ and $\phi R^{m'} \psi'$ where $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$. This is so because, by the construction of m' and the definition of PA

and R, m' and m (or r) have equivalent paths joining at equivalent scenarios. For the *only-if* case of the induction step it must be shown that for $\phi', \psi' \in PA^{m*}$ and $\phi' R^{m*} \psi'$ there exist an m such that $\phi, \psi \in PA^{m}$ and $\phi R^{m} \psi$ such that $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$. Again, this case follows by the construction rules for scenario trees and the definitions of PA and R. Consider the paths from ϕ' and ψ' to root. If $\phi' R^{m*} \psi'$ exists then the paths are not mutually exclusive and it is possible to construct a derivable state h with equivalent paths. Let m be a maximal state for h. By the definition of maximal state, Definition 10, ϕ and ψ exist in m as required. □

Corollary 13. *State m^* as defined in Theorem 12 is a system maximal state such that for all maximal states of the system, m, $m \leq_0 m^*$.*

For a system there may be more than one system maximal state but it is evident from Corollary 13 that all such states are equivalent.

4.4.3 Unfolded State

Theorem 12 proves the existence of system maximal states but does not provide an algorithm for construction of such a state. To proceed with safety analysis, an algorithm will be proposed to construct a characteristic state for a system under analysis. It will then be proven that such a state produced by the algorithm is equivalent to the system maximal states for the system. Safety analysis can then be conducted by inspecting this state. The analysis strategy presented here owes much to Sandhu's work on the Schematic Protection Model [San88]. In SPM, analysis proceeds using an unfolding algorithm which creates a representative of any subject that might exist in the system. The mechanism for transfer of access rights is applied iteratively wherever allowed until the systems stabilizes. This results in a worst case which identifies what access rights a subject might possibly get. This analysis must

assume monotonic security policies as rights are continually added to the system to reach the worst case, a maximal state.

The strategy here is different in that the unfolding algorithm is not creating representative subjects but instead creates representative scenarios. As system execution evolves, scenarios are created but are never destroyed. Scenarios that are not currently active are just past history. History cannot be destroyed. Therefore, the unfolding algorithm is monotonic in scenario creation over time. Each scenario represents an ordering of child scenarios. These scenarios are by definition only authorized at a certain time during the evolution of the parent scenario. The set of currently authorized scenarios is non-monotonic over system execution. This change in focus allows the use of an analysis strategy similar to that used with SPM to be used with systems with non-monotonic security policies.

The possible histories for actual systems can grow arbitrarily large. To provide efficient analysis an algorithm is needed to provide a method for the construction of an unfolded state. The algorithm should be designed to limit the complexity of the scenario tree that is produced for the purpose of safety analysis.

The intuition behind the construction of the unfolding algorithm is based on the property that equivalent scenarios may result in equivalent scenario trees (Lemma 8). If this is the case then recursive occurrences of equivalent scenarios may lead to redundancies in the unfolded state tree. *I.e.,* the redundant portions of the tree do not contribute to PA or R and need not be represented. Consider the following cases.

In the first case consider the occurrence of more than two equivalent concurrent scenarios on a path to root. Note that a message associated with a primitive scenario in a sub-tree rooted at a concurrent scenario can occur before or after any message associated with a primitive scenario in a sub-tree rooted at another concurrent scenario. If there are already two such equivalent scenarios on a path to root, another occurrence of a scenario from this equivalence class will not add to PA or R. This is

because the sub-trees rooted at these two scenarios will act as surrogates for sub-trees rooted at any arrangement of preceding and following equivalent scenarios along a path to root. By the concurrency of the sub-trees, scenarios from the equivalence classes of primitive scenarios represented in the sub-trees can come in any order with respect to each other. Further occurrences of equivalent sub-trees would not add any new orderings.

In the second case consider the occurrence of more than two equivalent non-concurrent scenarios on a path to root. Consider specifically the case where there are no intervening concurrent scenarios between the occurrences of the equivalent scenarios. If there are already two such equivalent scenarios on a path to root, another occurrence of a scenario from this equivalence class will not add to PA or R. Again, this is because the sub-trees rooted at these two scenarios will act as surrogates for sub-trees rooted at any arrangement of preceding and following equivalent scenarios along a path to root unbroken by a concurrent scenario.

The following definition describes a *fully unfolded state* that captures all possible scenario equivalence classes and eliminates redundant, recursive scenarios.

Definition 14. *For a system Σ, the fully unfolded state u is defined by applying the unfolding algorithm to the initial scenario, $\mathcal{S}^0$. The unfolding algorithm is defined by the following pseudo code:*

unfold($\mathcal{S}^0$)

Where:

unfold(S_{parent} : *scenario*)
begin
  for(each scenario descriptor, d, in $child_{ST}(\tau_S(S_{parent}))$ in turn)
  begin
    create a child, S_{child}, using d and $para_S(S_{parent})$

if($con_S(S_{child})$ **and** ∃ scenarios ϕ, ψ on the path to root for S_{child}
s.t. $\phi \equiv_0 \psi \equiv_0 S_{child}$ **and** $con_S(\phi)$ **and** $con_S(\psi)$) **then**
continue
elseif($\neg con_S(S_{child})$ **and** ∃ scenarios ϕ, ψ on the path to the last concurrent
scenario s.t. $\phi \equiv_0 \psi \equiv_0 S_{child}$) **then**
continue
else
unfold(S_{child})
end
end

The algorithm halts the evolution of a branch of the scenario tree when recurring scenarios are detected. This ensures termination of the algorithm.

Lemma 15. *The construction of definition 14 terminates.*

Proof. The scenarios created by the application of the algorithm form a tree rooted at the initial scenario $\mathcal{S}^0$. By the definitions of $child_S$ and scenario authorization each scenario can only have a finite number of children. At the most, a scenario can have a child corresponding to each of the scenario descriptors specified by the function $child_{ST}$ for that type of scenario. The evolution of tree branches is halted whenever more than two concurrent equivalent scenarios are on the path to root or if more than two non-concurrent equivalent scenarios are on the path to the last concurrent scenario. There are a finite number of equivalence classes of equivalent scenarios since there is a finite number of scenario types and a finite number of objects in $\mathcal{O}^0$. The length of every path in the construction must therefore be finite. □

An important property of state u is that it contains all possible sequences of non-repeating scenarios (up to the point of scenario equivalence). This property becomes

important later in proving that u is a system maximal state. The property of u containing all possible scenario sequences is proven for the following lemma.

Lemma 16. *For a system Σ, let u be the state produced by definition 14. For a sequence of scenarios, $(\phi_1, \phi_2, \ldots, \phi_n)$, that defines a path through the scenario tree for some state h, such that there are no equivalent scenarios in the sequence; there exists a sequence of scenarios, $(\phi'_1, \phi'_2, \ldots, \phi'_n)$, that defines a path through the scenario tree for u, where $\phi_1 \equiv_0 \phi'_1$, $\phi_2 \equiv_0 \phi'_2, \ldots, \phi_n \equiv_0 \phi'_n$.*

Proof. Consider a point in the execution of the algorithm where evolution beyond a scenario ϕ is halted because it is equivalent to some scenario ψ on the path to root for ϕ. By lemma 7, for any sub-tree rooted at ϕ an equivalent sub-tree is also possible for ψ. By definition 14, for each scenario all possible children are authorized (up to the point of repetition) in the unfolding of state beyond ψ. Therefore, any path sequence created by unfolding the state beyond ϕ (again up to the point of repetition) has an equivalent path sequence beginning at ψ. *I.e.,* further repetitions do not produce any new equivalence classes of non-repeating sequences, and state u contains all possible equivalence classes of non-repeating sequences. State h and state u are both evolved from $\mathcal{S}^0$. Therefore, for any non-repeating sequence in h an equivalent sequence can be found in u. □

4.4.4 Proof of u as a Maximal State

Now, it must be proven that a fully unfolded state u produced by the algorithm of definition 14 yields a state equivalent to a system maximal state. That is, it must be proven both that a state h, which is a system maximal state, is 0-reducible to u and that u is 0-reducible to such an h.

Lemma 17. *For a system Σ, let u be the state produced by definition 14. For any state h, which is a system maximal state, $h \leq_0 u$.*

Proof. It must be proven that:

1. for all $\phi \in PA^h$, there exists $\phi' \in PA^u$ such that $\phi \equiv_0 \phi'$ and
2. for all $\phi R^h \psi$, there exists $\phi' R^u \psi'$ such that $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$.

Part 1 follows directly from lemma 16. u has equivalent sequences for all non-repeating sequences that are present in h. u must then have equivalent scenarios for all scenarios that are present in h.

The proof for part 2 is more involved. All elements of PA and R in a system maximal state can be produced by some derivable state. It will suffice then to prove part 2 true for any derivable state h. Proof is by induction on the number of scenario authorization operations resulting in state h. The induction hypothesis is that part 2 holds for n scenario authorizations. The basis holds for the initial state and zero scenario authorization operations. Consider a state, h, with $n+1$ scenario authorizations. $\mathcal{S}^h = \mathcal{S}^g \cup \{\phi\}$, where state g has n authorizations and ϕ is the $n+1^{st}$ scenario authorized. By the induction hypothesis, part 2 holds for R^g and therefore for any pair in R^h which does not involve scenario ϕ. If ϕ is not a primitive scenario then $R^h = R^g$ and the induction step holds. If ϕ is a primitive scenario consider $\psi \in PA^g$. Now for the induction step it must be shown that part 2 holds for $\phi R^h \psi$ or $\psi R^h \phi$ if they exist (those pairs involving the new scenario). Let the paths to root for ϕ and ψ join at scenario σ. On the paths from ϕ and ψ to σ and on the path to root from σ there may be zero or more equivalent scenarios. Construct new paths for these three path segments by eliminating the segments of the paths between concurrent equivalent scenarios. For example, consider a path sequence from σ to ϕ, $(\sigma, \lambda_1, \ldots, \lambda_n, \theta_1, \gamma_1, \ldots, \gamma_k, \theta_2, \delta_1, \ldots, \delta_m, \phi)$, where $\theta_1 \equiv_0 \theta_2$ and θ_1

and θ_2 are concurrent. By lemma 8, it is possible to create a derivable scenario sequence $(\sigma, \lambda_1, \ldots, \lambda_n, \theta_1, \delta'_1, \ldots, \delta'_m, \phi'')$, where $\delta'_1 \equiv_0 \delta_1, \ldots, \delta'_m \equiv_0 \delta_m$ and $\phi'' \equiv_0 \phi$. Continue to eliminate segments of paths between concurrent equivalent scenarios until no such segments remain. Apply the same technique to eliminate segments of paths between non-concurrent equivalent scenarios that lie between any two concurrent scenarios. *I.e.,* for a path sequence $(\sigma, \lambda_1, \ldots, \lambda_n, \mu, \lambda_{n+1}, \ldots, \lambda_m, \theta_1, \gamma_1, \ldots, \gamma_j, \theta_2, \delta_1, \ldots, \delta_k, \omega, \delta_{k+1}, \ldots, \delta_l, \phi)$, where μ and ω are concurrent, form a new sequence $(\sigma, \lambda_1, \ldots, \lambda_n, \mu, \lambda_{n+1}, \ldots, \lambda_m, \theta_1, \delta'_1, \ldots, \delta'_k, \omega', \delta'_{k+1}, \ldots, \delta'_l, \phi'')$. Again, continue to eliminate segments until no such segments remain. In a similar way create new path sequences between σ and ψ'', and $\mathcal{S}^0$ and σ''. By lemma 8 it is possible to construct equivalent path sequences between σ'' and ψ''', and σ'' and ψ''', where $\phi''' \equiv_0 \phi''$, and $\psi''' \equiv_0 \psi''$. The resulting paths to root for ϕ''' and ψ''' have no more than two concurrent scenarios from any equivalence class. As well there will be no more than one scenario from any equivalence class between two concurrent scenarios. By the construction of definition 14 and lemma 16 there must be equivalent paths to root from ϕ' and ψ' in the scenario tree for u, where $\sigma \equiv_0 \sigma' \equiv_0 \sigma'''$, $\phi \equiv_0 \phi' \equiv_0 \phi'''$, and $\psi \equiv_0 \psi' \equiv_0 \psi'''$. Therefore by the definition of R^u, $\phi' R^u \psi'$ or $\psi' R^u \phi'$ will exist as required. □

Lemma 18. *For a system Σ, let u be the state produced by definition 14. For any state h, which is a system maximal state, $u \leq_0 h$.*

Proof. It must be proven that:

1. for all $\phi' \in PA^u$, there exists $\phi \in PA^h$ such that $\phi \equiv_0 \phi'$ and
2. for all $\phi' R^u \psi'$, there exists $\phi R^h \psi$ such that $\phi \equiv_0 \phi'$ and $\psi \equiv_0 \psi'$.

To prove part 1 consider a primitive scenario $\phi' \in PA^u$. By definition 14 the path to root for ϕ' is derivable. Therefore, there exists a state h' for which the scenario tree

contains an equivalent path. From lemma 11 and theorem 12 there exists a system maximal state h such that $h' \leq_0 h$. Therefore, there exists a scenario $\phi \in PA^h$ as required.

To prove part 2 consider a pair $(\phi', \psi') \in R^u$. The paths to root for ϕ' and ψ' join at scenario σ'. By definition 14 the paths to root for ϕ' and ψ' are derivable. By the definition of R, $order_{ST}(\tau_S(\sigma)) \neq or$. Therefore, the branch paths to ϕ' and ψ' are not mutually exclusive. A derivable state h' can be constructed such that its scenario tree has paths equivalent to these two. The new tree will contain scenarios ϕ, ψ and σ such that $\phi \equiv_0 \phi'$, $\psi \equiv_0 \psi'$ and $\sigma \equiv_0 \sigma'$, and the paths to root for ϕ and ψ join at σ. From lemma 11 and theorem 12 there exists a system maximal state h such that $h' \leq_0 h$. Therefore, there exists a scenario pair $(\phi, \psi) \in R^h$ as required. □

Theorem 19. *For a system Σ, let u be the state produced by definition 14. For any state h, which is a system maximal state, $u \equiv_0 h$.*

Proof. The proof follows directly from lemmas 17 and 18, and definition 9 for state equivalence. □

4.4.5 Complexity of Safety Analysis

The fully unfolded state u is a system maximal state. The safety question formulated by definition 2 can be answered by constructing u and inspecting the authorization properties PA^u and R^u. This method will determine whether it is possible for two scenarios ϕ and ψ (or their equivalents) to exist and if the ordering $\phi R \psi$ is possible. The corresponding paths in the scenario tree for state u (the *analysis tree*) also provide a history of system events that make such an occurrence possible.

The complexity of performing such an analysis depends on the complexity of the two main operations, construction of the analysis tree and inspection of the analysis tree.

The unfolding algorithm described by definition 14 that produces the analysis tree is controlled by the procedure **unfold()**. The procedure has a main loop that creates a child for each scenario descriptor belonging to the $child_{ST}$ set defined for the scenario passed as a parameter to the procedure. The body of this loop performs two searches along the path to root for the newly created child scenario and may recursively invoke itself on the new scenario. The searches along the paths to root for repetitions of scenario equivalence classes can be done in constant time by maintaining a hash of the equivalence classes along the current path to root. Due to the recursive procedure call, the main loop of the procedure will be executed once for each scenario created by the construction of the analysis tree. This is in the order of $|\mathcal{S}|$. $|\mathcal{S}|$ is constrained by the number of child scenarios that can be created by each parent scenario (the fan-out at each node of the tree) and the depth of the paths in the tree (controlled by the parent-child relationships defined by $child_{ST}$ and the conditions controlling the recursion of the procedure).

The fan-out for a particular scenario, ϕ, is defined by the cardinality of $child_{ST}(\tau_S(\phi))$. This is finite by definition but is otherwise unconstrained by the modeling scheme. However, it is not normal for human designers to work with scenarios that have more than a few child scenarios.

The depth of the paths in the analysis tree are controlled by the parent-child scenario-type-to-scenario-type relationships defined statically by the scenario descriptors in $child_{ST}$. These relationships control how the analysis tree unfolds. The depth of the tree in bounded by the termination of evolution paths associated with the detection of repeated scenario equivalence classes on a path back through the tree to root. An upper bound for tree depth depends on the (finite) number of equivalence classes for scenarios possible in the system. The number of possible equivalence classes is characterized by the number of scenario types defined for a system, and the number of $\mathcal{O}^0$ objects which can act as parent acquired parameters to the scenarios.

The number of parent acquired parameters is defined by the modeling scheme to be finite but is otherwise unconstrained. There can be at the most two concurrent equivalent scenarios on a path to root and a maximum of two non-concurrent equivalent scenarios between any two concurrent scenarios on a path to root. Therefore the maximum path length (tree depth) is $(2 \times n_{equiv})^2$, where n_{equiv} is the number of possible equivalence classes for scenarios in a system.

The worst case analysis tree can be approximated by a k-ary tree. The number of internal nodes in a complete k-ary tree, where all nodes have degree (fan-out) k, and the depth of the tree is h is $(k^h - 1)/(k - 1)$. Clearly, the worst case upper bound is intractable even for small systems. However, it is expected that in actual systems the number of parent-child scenario-type-to-scenario-type relationships defined by $child_{ST}$ will be very much smaller than the complete connectivity implied by the worst case. This is the case for inter-scenario relationships in contemporary object-oriented analysis and design. There may be a large class of systems for which production of the analysis tree is tractable. The examples presented in Chapter 5 are representative of some interesting classes of system for which analysis is tractable.

The second of the two main operations that contribute to the complexity of performing safety analysis is the inspection of the analysis tree. In formulating a safety question in the format of definition 2 two scenarios, ϕ and ψ are defined. Occurrences of scenarios ϕ' and ψ' are identified in u, where $\phi' \equiv_0 \phi$ and $\psi' \equiv_0 \psi$. The joining scenario on the paths to root for each such ϕ' and ψ' is a scenario σ'. $order_{ST}(\tau_S(\sigma'))$ and $child_S(\sigma)$ are inspected to determine if the conditions are met for $\phi' R^u \psi'$. A full walk of an analysis tree of n scenarios ($|\mathcal{S}^u|$) visits each scenario twice and requires two scenario comparisons for each of the n scenarios (once each for ϕ and ψ). This complexity may be reduced by using an indexing scheme at the time of analysis tree creation. For a search that produces l scenarios ϕ' and m scenarios ψ', there are $l \times m$ checks required to determine if there exists a pair $\phi' R^u \psi'$. For each of these cases the

paths to root must be searched to find the appropriate σ'. This would require in the worst case h comparisons, where h is the depth of the paths to root. The complexity of the inspection operation is in the order of $2n + l \times m \times h$. The size of l, m, and h will be small relative to n. For an analysis tree that has tractable construction the inspection of the tree is also tractable.

Chapter 5
WORKED EXAMPLES

5.1 Introduction

This chapter presents a set of examples. The SBAC modeling scheme is applied to specific kinds of problems. The first four examples are taken from the literature. They provide a small set of interesting security policies that have been used in other work to demonstrate the ability of a modeling scheme to express useful system models. This is also the purpose of including such examples here. The last example is an SBAC adaptation of an object-oriented analysis for a military message system. The original non-SBAC model was done by a graduate student as part of the requirement for completion of a course in object-oriented analysis and design. This model provides an example of SBAC being used in conjunction with contemporary software engineering analysis and design techniques. This example also provides an example of a model of a larger system, and an example of how the safety analysis scheme performs with a larger size model.

A design capture and analysis tool was implemented to explore the definition of SBAC models and the construction of analysis trees for specific system instances of a model. The tool can be used to specify a security model directly using a windowing interface or it can accept a specially marked-up Rational Rose model file as input. This tool was used in the development of some of the examples presented. The last section of the chapter describes the modeling tool and some of the results achieved.

5.2 Document Release Example

The document release example was presented in Section 2.3.3 (*cf.* Sandhu, [SG94]). Recall that a scientist prepares a paper for publication. Before the scientist is allowed to publish the paper it must be cleared for publication by a patent officer. The patent officer can authorize the paper for publication or she can return it to the scientist for revision. The scientist is initially able to modify the content of the paper but loses that right while the paper is under review. The scientist is also not able to alter the content of a paper authorized for publication by the patent officer.

An SBAC analysis specification for this problem was presented in Chapter 2, Figures 2.3 to 2.10. The work flow of a document begins with a period allowing the scientist to work on the document, moves to a review period by the patent officer, and then back to the scientist, either for publication or rework. The first three scenario types presented in the set of figures are: *Sinitial*, *SdocEdit*, and *SforwardDecision*. These control the scientist's initial editing of the document and decision to forward the document for review. *Sinitial* is a sequential scenario and specifies that the scientist be allowed to edit the document. This is authorized by an *SdocEdit* scenario, which in turn authorizes a read of the document followed by a write (primitive scenarios). When an edit is complete *Sinitial* authorizes the scientist to make a decision about forwarding the document. *I.e.*, authorization of a *SforwardDecision* scenario. An *SforwardDecision* scenario authorizes both a recursive return to editing by authorization of another *Sinitial* scenario, and the forwarding of the document for review by authorization of an *SdocForward* scenario. The *SforwardDecision* scenario has an *or* ordering so the actions involved in these two child scenarios are mutually exclusive. Given this set of scenarios the scientist can continue to edit the document an arbitrary number of times before finally making the decision to forward it for review. The next scenario type presented is *SdocForward* in Figure 2.6. This scenario type

specifies a message pass from the scientist to his assigned patent officer requesting a review of the document. Following this message a scenario of type *SdocReview* is authorized, Figure 2.7. This scenario allows the patent officer to read the document and then authorizes a scenario of type *SreleaseDecision*, Figure 2.8. The *SreleaseDecision* scenario has an *or* ordering. This allows the patent officer to make one of three choices. She can continue to review the document (recursive authorization of another *SdocReview* scenario), or she can approve the document for release, or she can return it to the scientist for revision (authorization of scenarios of types *SdocRelease* and *SdocRevision* respectively). An *SdocRelease* scenario, Figure 2.9 creates a new release authorization object and passes it back to the scientist. The scientist can use this object to indirectly publish the document. An *SdocRevision* scenario, Figure 2.10 passes the document back to the scientist and recursively starts the whole editing and review process over again by authorizing another *Sinitial* scenario.

The only actions permitted by the system are those authorized by scenario instances. The example has been designed such that the scientist can either submit the document for review or continue to edit. One action prohibits the other. The scientist gets the authorization to edit back only if the document is returned for revision. If the document is approved for release, the scientist has no direct access to the document to edit. This is because there is no way to generate an *SdocEdit* scenario after the document has been approved for release. The only authorized action is to release the document for publication. Note that the role parameters for a scenario cannot be changed after the scenario is authorized. Therefore, the construction of scenario type *SdocRelease* ensures that the creation of the release authorization binds the approved document to the release authorization, and that the document release authorization is returned to the author of that document.

5.3 Project Management Example

The project management example is an example of role-based access control using hierarchical roles. Role-based access-control models are characterized by the definition of SBAC roles. Such roles were described in Section 2.2.18 and are separate from the concept of scenario/object roles associated with the scenario and object bindings defined for SBAC modeling in Chapter 3. In RBAC models a role is associated with a set of permissions. Users are associated with roles at the discretion of the system administrator. Users perform all activities within the scope of a session. A session may have one or more of the user's permitted roles active at any time.

The project management example (*cf.* [San98]) considers a system where there are multiple people working together on a project. There are a number of shared objects the different team members will interact with in different ways. The kinds of permissions the team members have for the shared objects can be broken down into three groups that can be associated with three different roles. Permissions needed by all team members can be held by a role *project member*. There are also roles for *test engineer* and *programmer*. An administrative role, *project supervisor* should inherit all permissions available for the shared project objects. This role hierarchy is a partial order and is depicted in Figure 5.1.

The permissions associated with roles are statically assigned and represent abstract authorizations. Typically, the permissions allow a subject to execute a specific program on a specific type of data item. It seems natural to model such permissions as an authorization to invoke an interface method of an abstract data type, *i.e.* an object. In this example permissions will be authorizations for primitive scenarios.

An RBAC role will be represented as a scenario type. Scenarios are defined by the modeling scheme to authorize a number of other scenarios. In this case a scenario representing an RBAC role authorizes a number of primitive scenarios that represent the

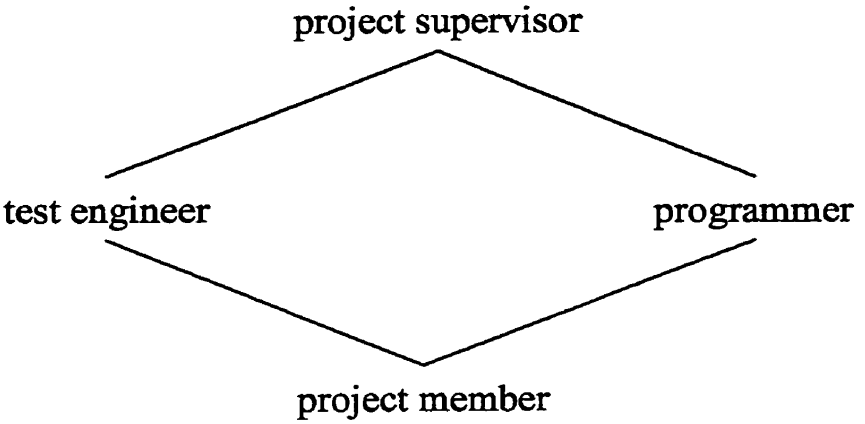

FIGURE 5.1. Project Management Role Hierarchy

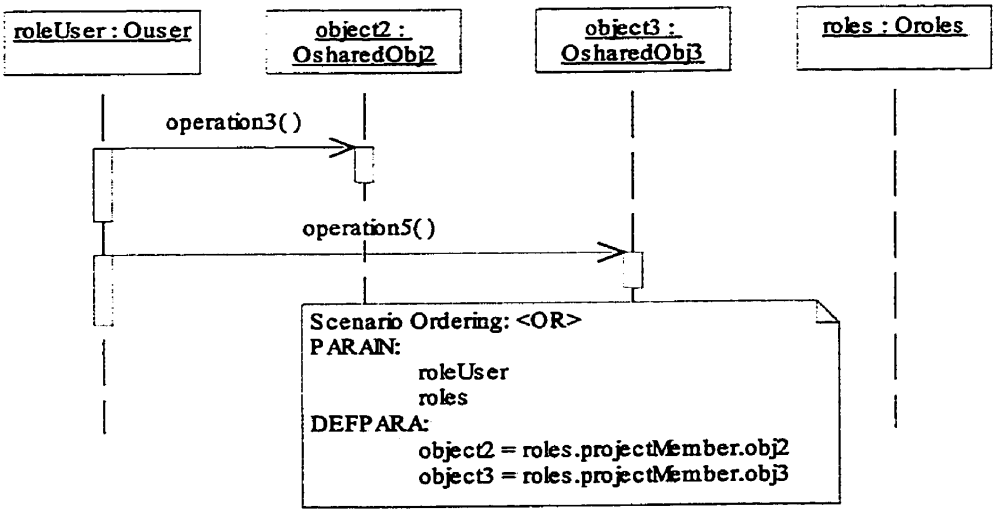

FIGURE 5.2. Scenario Type SprojectMember

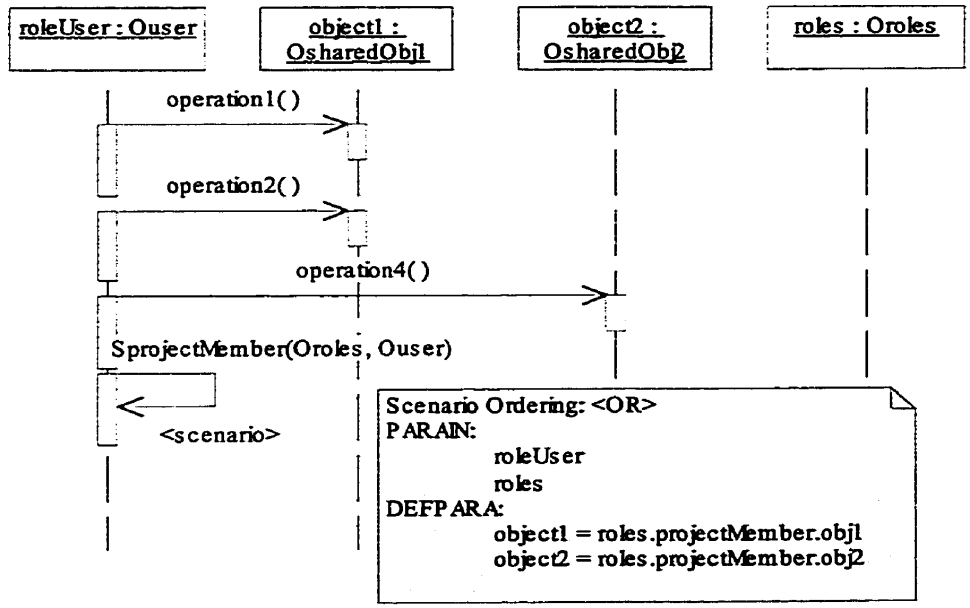

FIGURE 5.3. Scenario Type StestEngineer

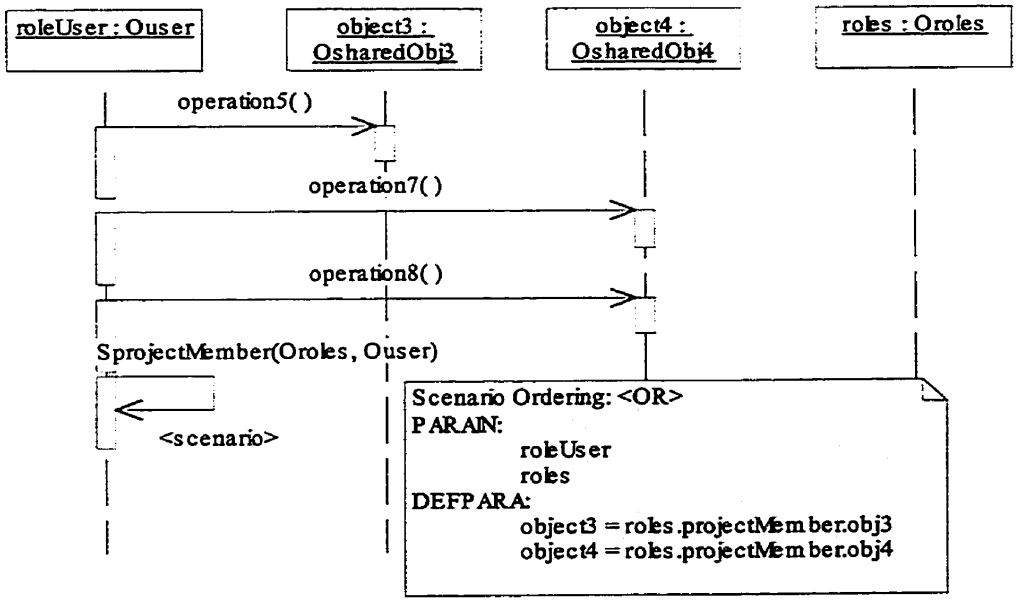

FIGURE 5.4. Scenario Type Sprogrammer

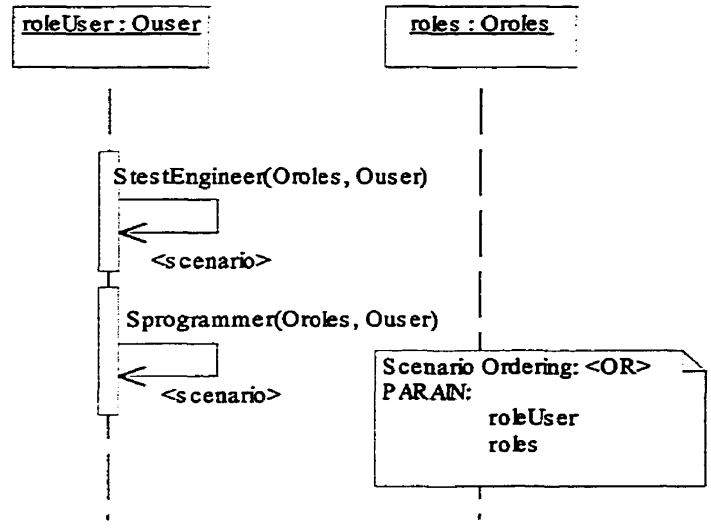

FIGURE 5.5. Scenario Type SprojectSupervisor

permissions associated with the RBAC role. A scenario representing an RBAC role has an *or* ordering. This allows all the permissions in a role to be authorized simultaneously, as is normal in RBAC. Roles may be arranged in hierarchies by including one RBAC role scenario type as a child of another. For example, see Figures 5.2 to Figures 5.5. The scenario type *SprojectMember* specifies that the role user can invoke operation3() on *object2* and and operation5() on *object3*. The object *roles* is present here only as a mechanism to provide visibility of the objects under access control. Scenario types *StestEngineer* and *Sprogrammer* also specify operations that a role user can invoke. These are different operations that are reserved for users who are acting as test engineers or programmers respectively. These scenario types also specify the authorization of a child scenario of type *SprojectMember*. The inclusion of this child makes the permissions specified for all project members available to users acting in the test engineer or programmer role. The scenario type *SprojectSupervisor* provides the role user with all available permissions by authorizing children of types *StestEngineer* and *Sprogrammer*.

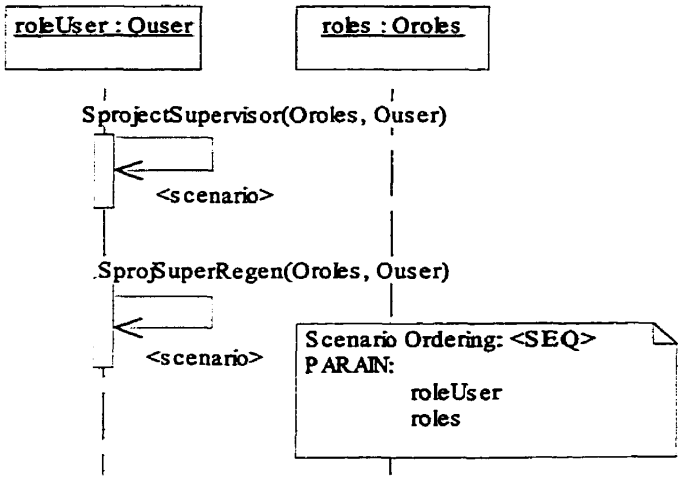

FIGURE 5.6. Scenario Type SprojSuperRegen

An active role also has associated with it a *role regenerator* scenario. When any one of the primitive scenarios representing an RBAC role permission is terminated (*i.e.* the permission is used) all the RBAC role scenarios in the hierarchy terminate because of the *or* scenario ordering specified for the scenario types. It is normal for a user to have all the permissions associated with a role for as long as the role is active in a session. Therefore, the role hierarchy needs to be regenerated to restore the permissions after one is used. Consider the scenario type *SprojSuperRegen* specified in Figure 5.6. This scenario type has a *seq* ordering. The first child scenario authorized is an RBAC role scenario (an *SprojectSupervisor* scenario in this case). If a permission is used some place in this role hierarchy the *SprojectSupervisor* scenario terminates and the *SprojSuperRegen* scenario recursively authorizes a new *SprojSuperRegen* instance, which restores the role permissions. For each role that a user may directly assume, a corresponding *role regenerator* is specified.

A session is managed in this example by a *session controller* object and set of *role initiator* scenario types. There is one *role initiator* for each RBAC role that a user may directly assume. A user controls what roles are active by sending mes-

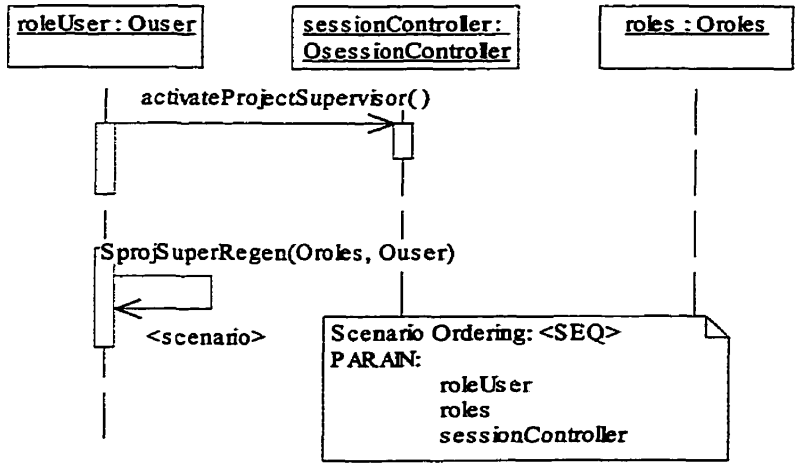

FIGURE 5.7. Scenario Type SprojSuperInitiator

sages to a *session controller.* An activate message is sent when the user wants to make a role active. Each of these messages, which activate different roles, is authorized by a different *role initiator.* The *role initiator* is a scenario type with a *seq* ordering. When an initiation message is sent, the authorization for that message is consumed and a *role regenerator* scenario for the associated role becomes authorized. For example, see Figure 5.7. This scenario type activates the *SprojectSupervisor* role. When the user sends an activateProjectSupervisor() message to the *sessionController* the *SprojSuperInitiator* scenario authorizes a *SprojSuperRegen* scenario that provides the required permissions.

The mappings from users to roles are specified by a *role selector* scenario type defined for each user. A *role selector* scenario authorizes a *role initiator* child scenario for each one of the roles permitted for the associated user. The *role selector* has an *and* scenario ordering. This means all the *role initiators* associated with the *role selector* are authorized at the same time. A role is not active until the activation message for its *role initiator* occurs. The recursive nature of the *role regenerators* authorized by *role initiators* means that the *role initiators* will never terminate. Therefore, a user's

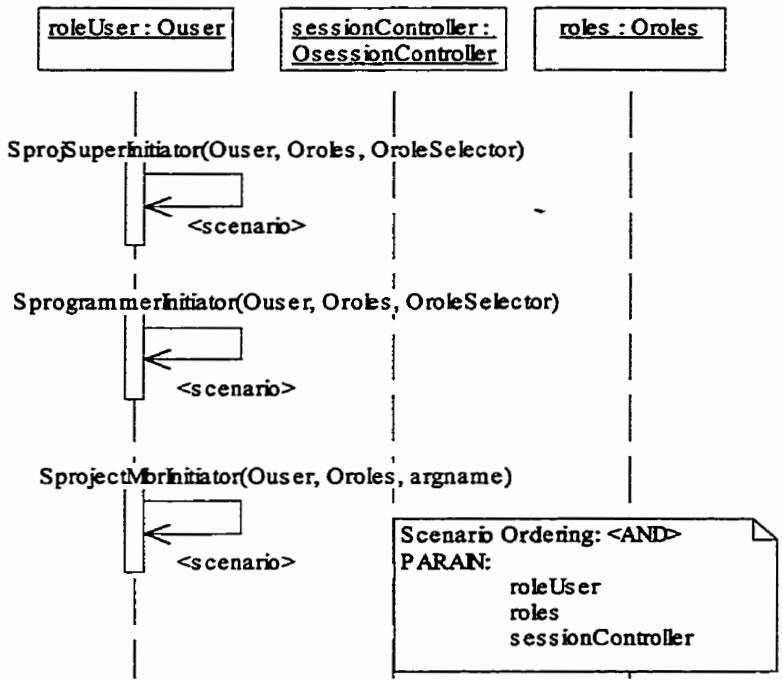

FIGURE 5.8. Scenario Type SbobsRoleSelector

role selector scenario is specified in this example to be a concurrent scenario. This is so that the scenario that spawns the *role selector* is not dependent on the selector's termination before it is able to carry on with other authorizations. Figure 5.8 illustrates a *role selector* for some user, Bob. Suppose Bob is authorized by his company to act as both the project supervisor and as a programmer on the project. He is also a member of the project team. Note that a scenario of type *SprojSuperInitiator*, which was defined above, is a child of a *SbobsRoleSelector*. Scenarios of type *SprogrammerInitiator* and *SprojectMemberInitiator* are also specified as children and would be defined similarly.

This example provides a model for a specific role-based security policy. The example also provides a general strategy for approaching RBAC style systems using an SBAC modeling scheme. The role hierarchy lattice can be captured by using hierarchical parent-child relationships between scenarios that are modeling role permissions. A security administrator can specify a set of user types, roles based on the set of user types, roles an individual user is permitted to activate, and permissions associated with each role. Although this example did not consider deactivation of roles in a user's session, it is easy to include such a requirement as part of the role regeneration mechanism.

5.4 Sales-order Processing Example

The purpose of this example is to demonstrate the use of SBAC to model task-based access control. A version of the classic sales-order processing example is presented in [TS94] to illustrate the modeling and management of task-based authorizations. One of the primary goals of scenario-based access control is to provide efficient safety analysis for systems modeling legitimate use policies. Legitimate-use is also a motivation for TBAC. A version of the sales-order processing example is used here as a

demonstration of SBAC being used to capture task-based access-control requirements.

Recall from Section 2.2.19 that the fundamental abstraction is an authorization-step. An authorization-step represents a primitive authorization processing step and is the analog of a single act of granting a signature in a paper-based system. A permission in TBAC is an element of $P \subseteq S \times O \times A \times U \times AS$, where S is the set of subjects, O is the set of objects, A is the set of actions or access rights, U is a set of usage and validity counts, and AS is the set of authorization-steps. Permissions are associated with exactly one instance of an authorization-step. An instance of an authorization-step is associated with exactly one instance of a task. Each authorization-step maintains a protection state that is the set of permissions currently valid for that authorization-step.

In this example primitive scenarios are used to model TBAC permissions. Scenarios are used to model authorization-steps. Tasks are groupings of authorization-steps and are also modeled in SBAC by scenarios. The task grouping relationship is modeled by the scenario parent-child relationship. A primitive scenario specifies a sender, a receiver, and a message type. This corresponds to the S, O, and A components of a TBAC permission. An SBAC primitive scenario authorizes a single message pass, which can be considered its usage and validity count. SBAC primitive scenarios are associated with the scenario that created them and provide an authorization for a message pass in the context of that scenario. This models the relationship between a TBAC permission and its associated authorization-step.

TBAC provides for existential, temporal and concurrency dependencies between authorization-steps. There are related dependencies explicitly specified in SBAC models. An existential dependency in TBAC specifies that the change in state of one authorization-step implies a change in state of some other authorization-step. In SBAC, the creation of a scenario can imply the creation of one or more of its children. As well, the termination of a child scenario can imply a change of state in its parent,

i.e. the creation of another child. A temporal dependency in TBAC specifies that a state transition for one authorization-step necessarily occurs before some state transition for some other authorization-step. Temporal dependencies in SBAC are specified by the order of occurrence of scenario descriptors in $child_{ST}$ and the scenario ordering specified by $order_{ST}$. Temporal dependencies between primitive scenarios (permissions) can be discerned by inspection of the authorization properties defined by the pre-order R on the set PA. Concurrency is modeled in SBAC by specifying that a child scenario is concurrent to its parent.

The sales-order processing example presented here is composed of five main authorization tasks: sale-terms, credit-terms, goods-removal, shipping-terms, and billing. These tasks track and control the progress of a merchandise order through the business structure of a vendor. There are one or more authorization-steps associated with each task. It is important that the steps occur in the correct order and that only certain specific people authorize the progress of an order through the various steps of the business process. An SBAC model for each of the five tasks will be presented in turn.

The task sale-terms is modeled by the scenario type *SsaleTerms*. There are three authorization-steps associated with this task. They are the creation of a new sales order, negotiation and authorization of the price, and negotiation and authorization of the delivery date. The steps must occur in this order. See Figure 5.9. The authorization-steps are modeled here as primitive scenarios. *I.e.,* in this case there is only one permission associated with the authorization-step, so it is not modeled as a separate scenario type containing a single primitive scenario. This can be thought of as an abstract representation of the authorization-step. If more detail is provided as the model is refined, these primitive scenarios may become complex and hide a number of permissions (or sub-authorizations) as required. Each of these authorization-steps is authorized by a sales clerk. Only the clerk associated with this task may make the

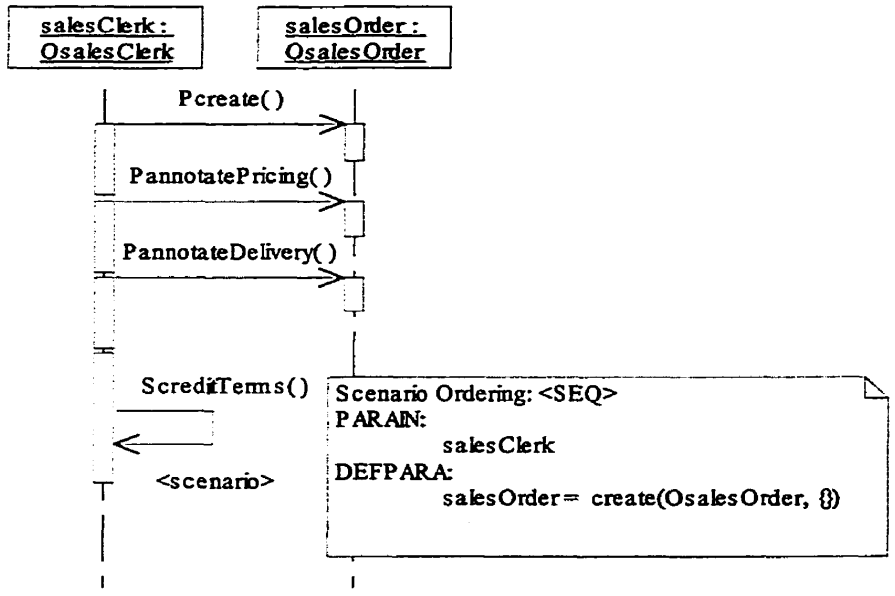

FIGURE 5.9. Scenario Type SsaleTerms

appropriate authorization. The last child specified is a non-primitive scenario of type *ScreditTerms*. This is an authorization for the next task, credit-terms.

The task credit-terms is modeled by the scenario type *ScreditTerms*, Figure 5.10. The task is composed of four authorization-steps. First, the sales clerk who prepared the order is authorized to forward the order to a credit clerk for credit checking. Next, the credit clerk is authorized to read and then perform a credit check on the client. Notice that *PperformCreditCheck* is a primitive scenario (it is not marked with a <scenario> tag). The credit clerk is sending a message to itself. This is an abstract operation and a likely candidate for further elaboration. The fourth authorization-step is modeled by a child of scenario type *ScreditCheck* and its children. *ScreditCheck* has an *or* scenario ordering. This is used to capture the non-determinism involved in the decision by the credit clerk to grant or not to grant credit to the customer. This would presumably be based on the customer's credit rating and account history which cannot be known until run-time. *ScreditCheck* results in two authorizations; however,

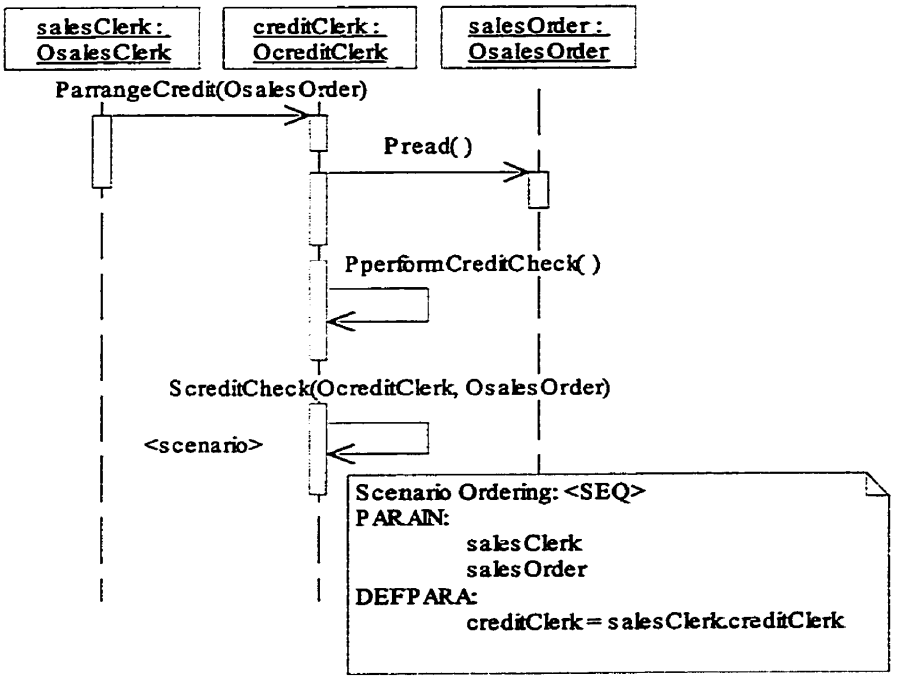

FIGURE 5.10. Scenario Type ScreditTerms

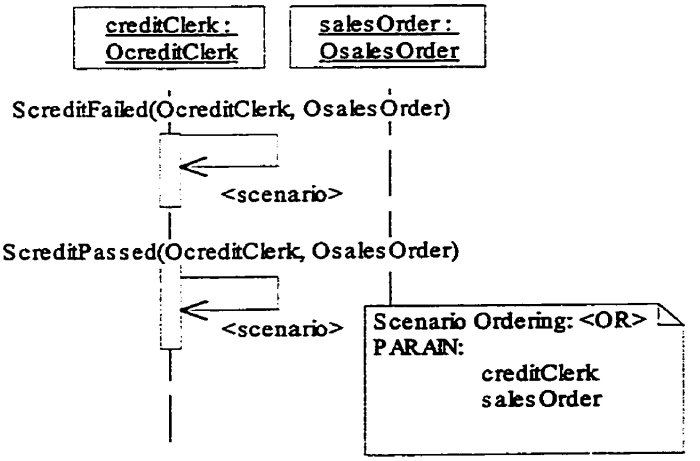

FIGURE 5.11. Scenario Type ScreditCheck

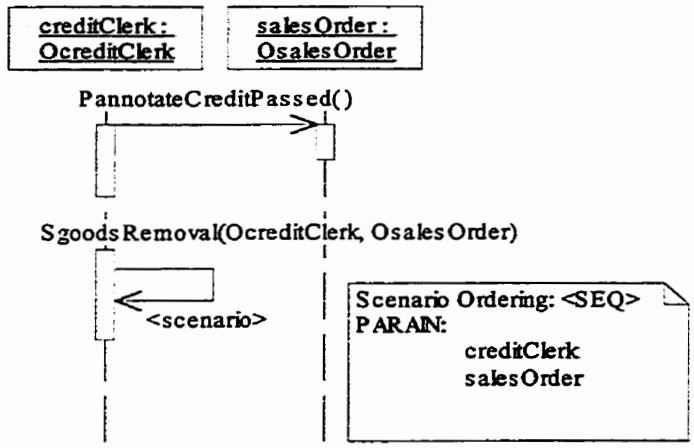

FIGURE 5.12. Scenario Type ScreditPassed

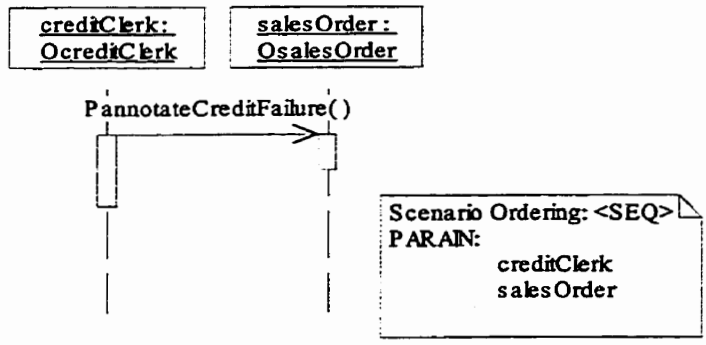

FIGURE 5.13. Scenario Type ScreditFailed

only one will be actioned. Effectively the credit clerk is authorized to either grant credit or refuse credit, but not both. The associated children are scenarios of types *ScreditPassed* and *ScreditFailed*, Figures 5.12 and 5.13. In the case of *ScreditPassed* the sales order is annotated as passed and the next task, goods-removal, becomes authorized. In the case of *ScreditFailed*, the sales order is annotated as failed and the scenario terminates. This causes its parents to terminate as well, backing up the scenario tree to the original *SsaleTerms* scenario. This happens because there are no more children to create in accordance with their scenario descriptor sets and scenario orderings. This effectively revokes all authorizations for the sales order. As the model is presented, the sales order is not able to participate in any further scenarios. The failed credit step stops any other processing of the sales order.

Note that the visibility for the credit clerk in this scenario type is provided by an object visibility from the sales clerk, *i.e.* via a defined parameter. This is a simple mechanism to illustrate this example. A more complex mechanism such as the role-based access-control scheme in the previous example is possible.

The task goods-removal is modeled by the scenario type *SgoodsRemoval*, Figure 5.14, and its children. The assumption in this example is that there are two warehouses associated with the merchandise vendor. For any sales order some product may come from each warehouse. The purpose of scenario type *SgoodsRemoval* is to create two sub-tasks to handle authorizations at the two warehouses. The authorizations at a warehouse are controlled by scenarios of type *Swarehouse*, Figure 5.15. Two instantiations of this scenario type are created for each sales order, one for each warehouse. The difference between the two instantiations is the warehouse manager provided as parameter to the scenario. The task is composed of three authorization-steps. Each *Swarehouse* scenario authorizes the credit clerk to forward the sales order to the warehouse manager. The warehouse manager is then permitted to read the sales order. The last authorization-step allows the warehouse manager to check the

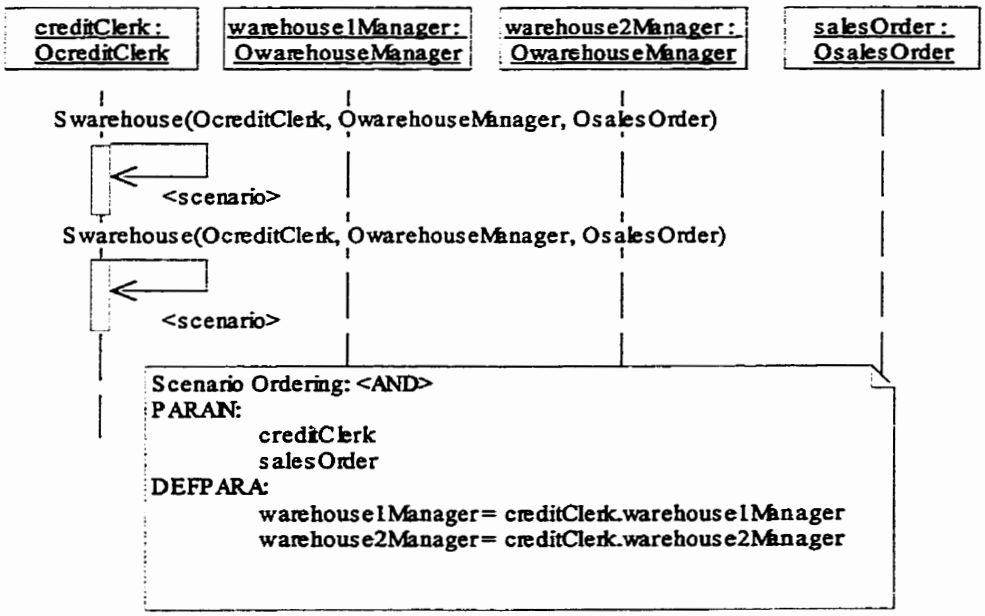

FIGURE 5.14. Scenario Type SgoodsRemoval

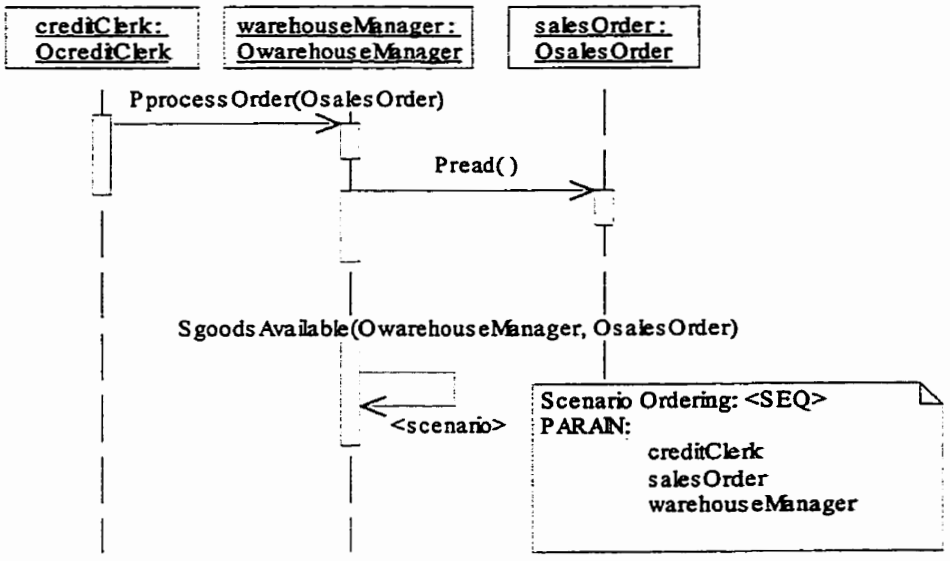

FIGURE 5.15. Scenario Type Swarehouse

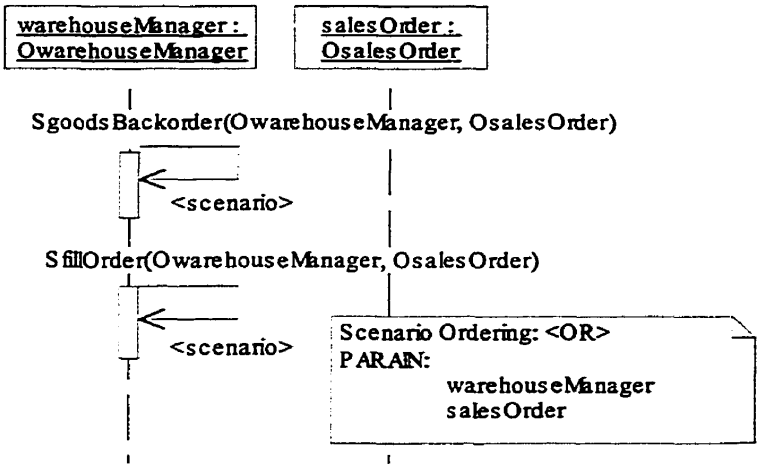

FIGURE 5.16. Scenario Type SgoodsAvailable

inventory of the warehouse and fill the order. This authorization-step is modeled by scenarios of type *SgoodsAvailable*, Figure 5.16. Scenario type *SgoodsAvailable* has an *or* scenario ordering to account for the non-deterministic possibilities of goods being available to fill the order, and goods being temporarily unavailable to fill the order (backordered). The warehouse manager is authorized to fill the order or place the order on backorder, but cannot perform both actions. The associated child scenarios are *SfillOrder* and *SgoodsBackorder*, Figures 5.17 and 5.18. Scenarios of type *SfillOrder* authorize the warehouse manager to unstock the goods from the warehouse inventory, and annotate the sales order as having been filled. The last child specified is a non-primitive scenario of type *SshippingTerms*. This is an authorization for the next task, shipping-terms. Scenarios of type *SgoodsBackorder* authorize the warehouse manager to annotate the sales order as having backordered goods. At some point the goods will become available and a scenario of type *SfillOrder* will be created. Both the case that goods are immediately available and the case that goods are backordered lead eventually to the creation of a *SfillOrder* scenario, which terminates with a scenario authorizing the next task, shipping-terms.

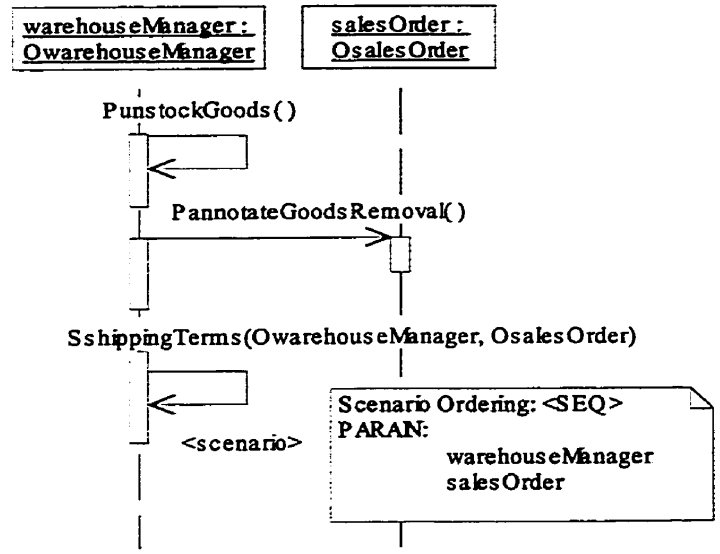

FIGURE 5.17. Scenario Type SfillOrder

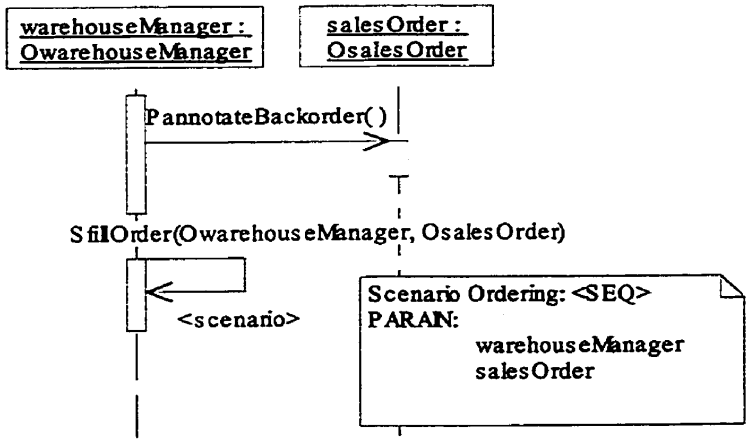

FIGURE 5.18. Scenario Type SgoodsBackorder

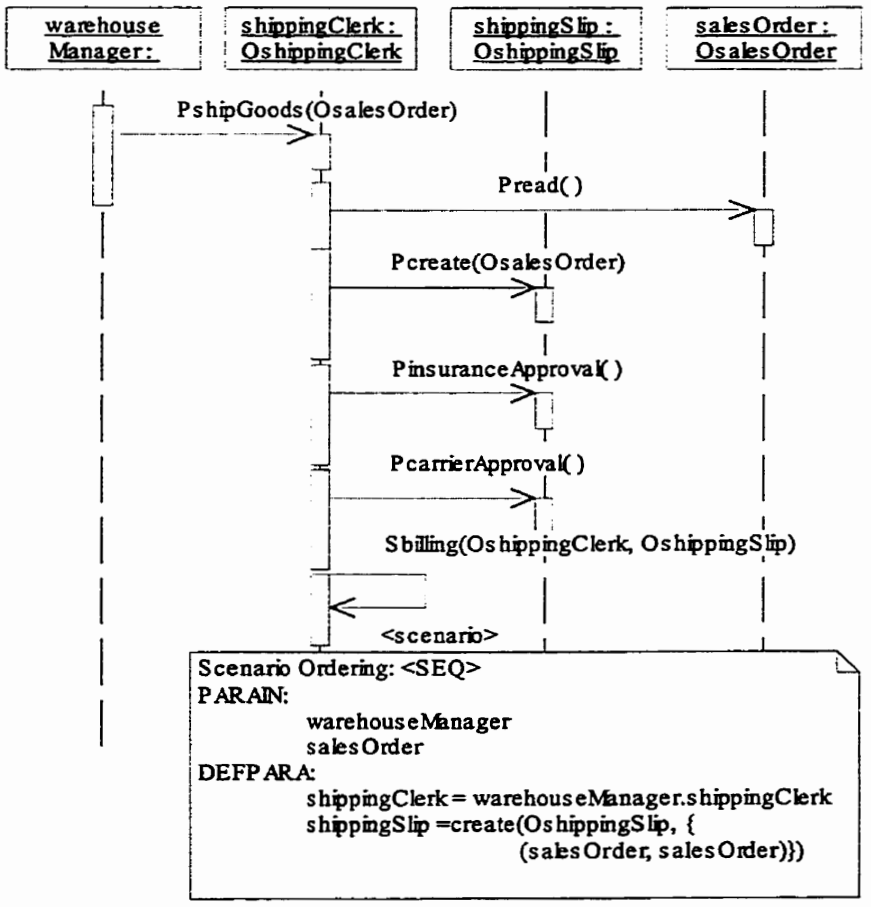

FIGURE 5.19. Scenario Type SshippingTerms

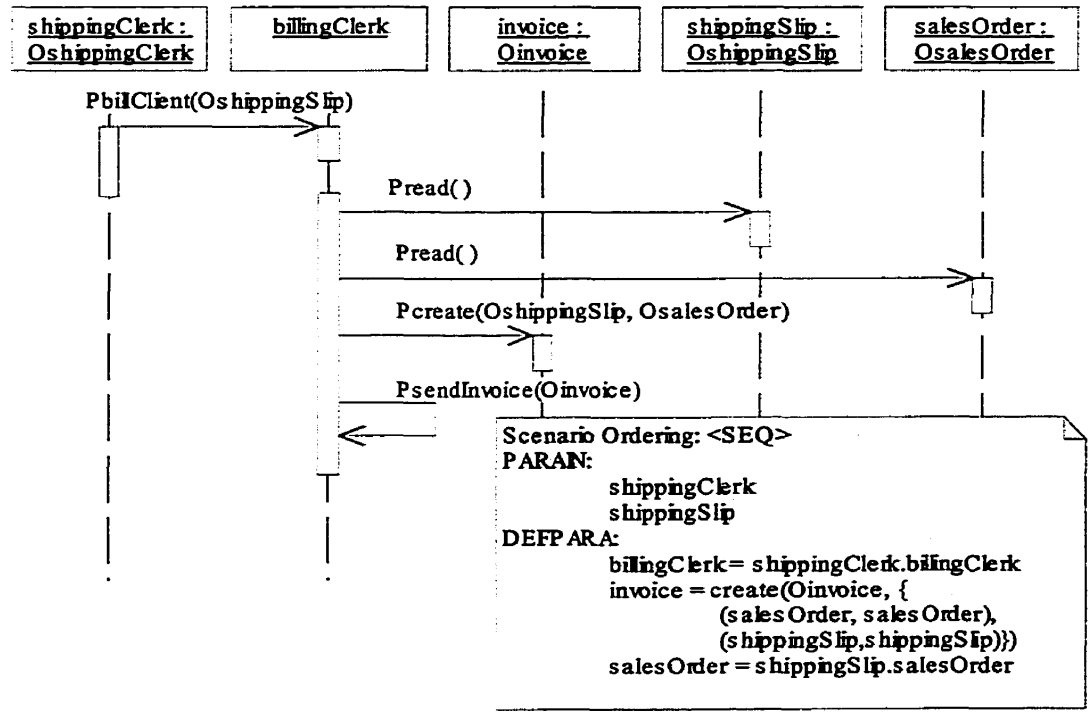

FIGURE 5.20. Scenario Type Sbilling

The task shipping-terms is modeled by the scenario type *SshippingTerms*, Figure 5.19. This task is a straight forward set of authorization-steps that allow the warehouse manager to forward a sales order for which goods have been allocated to a shipping clerk. The shipping clerk can read the sales order, create a shipping slip for this portion of the order, and authorize insurance approval and carrier approval. These authorizations are to be carried out in this order. The last child specified is a non-primitive scenario of type *Sbilling*. This is an authorization for the next and last task, billing.

Billing is modeled by the scenario type *Sbilling*, Figure 5.20. Again this is a straightforward task. The first authorization-step allows the shipping clerk to forward the shipping slip to the billing clerk. The shipping slip provides visibility to the

associated sales order. The billing clerk is authorized to read both and then to create an invoice corresponding to the shipment. The billing clerk is then authorized to send the invoice to the customer. This is the last step in this task. The scenario type does not authorize any other tasks so it terminates. This causes its parents to terminate as well, backing back up the scenario tree to the original *SsaleTerms* scenario. This is because there are no more children to create in accordance with their scenario descriptor sets and scenario orderings. This effectively revokes all authorizations for the sales order. As the model is presented, the sales order is not able to participate in any further scenarios. This represents the completion of a successful sales order in this example.

This sales processing system provides an example of a task-based authorization scheme modeled using SBAC. The example illustrates control over the order of authorizations, control over the type of user and specific user that can authorize a step in the sales process, and branching exceptional case processing (*e.g.* failed credit terms, backorders). Composite authorizations are also supported by nested scenarios. The example exhibits all the fundamental requirements for $TBAC_1$.

5.5 BLP Example

The Bell and LaPadula model will only be treated briefly here. SBAC was designed primarily to provide a modeling scheme for legitimate use but is essentially policy neutral. BLP was designed primarily to provide a modeling scheme for confidentiality. BLP inherently models one way information flow on a classification-clearance lattice. However, it is possible to model BLP style lattice-based policies using SBAC. Detailed models will not be presented here, but brief sketches of how BLP can be realized by SBAC will be illustrated.

BLP with tranquility is defined in the literature to be BLP where classifications

of objects and clearances of users cannot be changed once they have been assigned. *I.e.,* security labels (MAC labels) are fixed. There is a strong correlation between tranquility and the strong typing defined for SBAC. The relationship between strong typing and MAC labeling has been previously explored in [SG94]. The strategy is to provide a type associated with each security level. Objects have interface operations corresponding to the basic BLP rights, *i.e.* read, write, append, and execute. Scenarios can be defined that provide authorization for senders (BLP subjects) of the appropriate clearance level (as defined by the lattice) to invoke one of the basic interface operations of a receiver (BLP object). Where an access is prohibited by the lattice, no scenario is defined. That is, scenarios are only defined such that the simple security property and *-property hold. Accesses not defined by a scenario are prohibited by SBAC.

Alternatively, Sandu shows in [San98] that RBAC is general enough to implement lattice-based models, such as BLP. RBAC can be used to implement static policies based on confidentiality lattices, policies based on Biba style integrity lattices, and Brewer and Nash style Chinese wall policies. It has been demonstrated that SBAC can be used to model RBAC, which is general enough to support these policies.

5.6 Battlefield Information System Example

The last example presented in this chapter is an SBAC adaptation of an object-oriented analysis for a military message system. The original non-SBAC model ([Cos98]) was done by a graduate student as part of the requirement for completion of a course in object-oriented analysis and design, which was taught by the author. The system requirement for the Battlefield Information System (BIS) is to support the electronic dissemination of tactical military message traffic. This traffic is currently handled primarily by voice radio networks and transcription to and from a standard-

ized set of paper message forms. The system will permit units to compose and send messages to other units, view messages, and to receive messages. Messages will be in the standard format specified in Canadian Army publications. The graduate student completed a requirement specification for the system.

This model provides an example of SBAC being used in conjunction with contemporary software engineering analysis and design techniques. The student used object-oriented analysis methods and captured his design using UML [RJB98] as an OO modeling language. This model was then modified by the author of this dissertation to include the additional information required to complete an SBAC model of the system requirement specification.

The BIS provides an example of a model of a larger system and an example of how the safety analysis scheme performs with a larger size model. By 'larger' it is meant here that the Battlefield Information System example is larger than the pedagogical examples presented earlier in the chapter. It represents a simple requirement specification for a system with a real requirement. The BIS model contains 90 primitive scenarios and 58 complex scenarios, so its security policy, $\mathcal{P}$, is about an order of magnitude larger than the policy for the Document Release example.

The scope of the Battlefield Information System can be briefly illustrated by presenting its use cases. Use cases are used to model the interactions or dialog a user has with a system. They define the functional capabilities a system should have, from the viewpoint of the user. The following list describes the use cases for BIS.

Initialize system The system is initialized on startup. All messages, codes, users, and logs are retrieved from backup storage and stored in the BIS system data structures.

Login-logout Users must log in to start using the message system. A record of valid users and passwords is kept, and the User/Administrator/Security Officer

provided login name and password is verified against this list. Users may logout when they are finished. This does not shut the system down but only means that a valid user must login before the system may be used again. Login attempts, successful and not, are recorded in the system log. Logouts are recorded in the system log. Before completing any requested action from any user the BIS must first verify the type of user that is logged in to see if it is a valid request.

Compose new message A user is able to compose a message and save it for later transmission. Message format will be chosen from a message in the prototype message list. A new message is created from the prototype and placed in the draft message list. Default values may be preset for some fields of the prototype message by the system administrator.

Send message A user is able to send one or more of the current messages in the list of draft messages. All mandatory fields will be filled in before the message is sent. The message will be tagged as transmitted. All messages must be encrypted before being transmitted.

Receive message A message arrives on the network and becomes a system object. The received message is added to the list of received messages.

View message A user is able to view any message in the system currently held at his node, *i.e.* a message from the received message list, sent message list, draft message list, or the prototype message list. The user will not be able to make changes to a message while it is being viewed.

Edit message A user is able to edit any message in the draft message list. At the end of each edit, when complete, a message remains as a draft message on the draft message list.

View message log A user is able to view the message log. A user is not allowed to make any changes to the log.

View system log The administrator is able to view the system log but may not make any changes to it.

Set message defaults The administrator is able to edit any message in the prototype message list. When complete the message remains as a prototype message on the prototype message list.

Add-remove user The administrator is able to add new users to the system, and remove users from the system.

View user The administrator is able to view a list of the users permitted to use the system.

Add-remove code The Security officer may add and delete message encryption codes.

View codes The security officer may view the list of message encryption codes.

Backup system The administrator can backup all system messages, codes, users, and logs to backup storage. On shutdown the system automatically performs a backup.

To separate the capabilities of the various kinds of user, the general strategy for elaborating the Login-logout use case is to use roles. The mechanism used in this example to separate roles is similar to that presented for the Project Management example in Section 5.3. In this example a user logging into the system must select a single role (corresponding to User, Administrator, or Security Officer) which will be active for the login session. At the time of role selection the user would be challenged

for an ID and password. In the role-based access-control specification for the Project Management example, the roles authorized primitive scenarios, which were used to invoke specialized operations on data objects. This provided an implementation for RBAC style permissions. In this example, the roles authorize a specific set of non-primitive scenarios. These scenarios are more than specialized operations on data objects. They define complex interactions between a number of system objects. The scenarios authorized by a role correspond to the use cases permitted for that role. The use cases are tasks that involve a specific set of object interactions, which must be processed in a specific order. The scenario type that defines the role usually specifies the authorization of a number of child scenarios. The scenario types of these child scenarios specify sub-tasks associated with the use case. Together, this set of scenario types elaborates a use case. For example, the View message use case is composed of five scenario types that permit the user to select one of the message lists and one of the messages on that list for viewing. The Login-logout use case itself is composed of fifteen scenario types that permit the user to select a role, select a task from those permitted for a role, regenerate the set of task permissions when a task is completed, and allow the user to logout from the system.

Most of the original object-oriented design for the BIS is preserved when SBAC modeling is applied to the system. The decomposition of the functional requirement into its use cases is essentially unchanged. The logical flow of the main scenarios that elaborate the use cases is for the most part unchanged. The original scenarios have been decomposed in many cases to provide more rigor in the description of the ordering of messages. Formal detail describing the relationship between scenarios has been added as it is not a part of the standard object-oriented analysis method.

There are some specific places were changes are required to accommodate the SBAC modeling. The original design uses polymorphism and object sub-typing relationships. This is to be expected in OO analysis, but object sub-typing relationships

are not supported by the current version of SBAC. Objects in the SBAC modeling scheme described require that an object has one fixed object type. The parameter binding mechanisms do not allow sub-type substitutions. Similarly changes were needed to accommodate parameterized types (sometimes called templates or generics). In such cases a scenario mechanism has been repeated for the individual types of objects in the sub-type relationship. In some cases the analysis has been made more abstract to submerge the detail requiring sub-types in the original.

Modeling this larger system exposed some areas where the expressiveness of the SBAC modeling technique seems weak. OO analysis and design models often make use of indirection in handling object visibility. The relationships between objects are allowed to change during the execution of the system. For example, in some scenarios a single message viewer object can be made to display different messages by changing a viewer object visibility role from one message to another. In the SBAC modeling scheme, object visibilities are defined at the time of creation and are fixed. This leads to some difficulty in expressing object visibility through indirection. The same situation is encountered when using lists of objects, where the contents of the list may grow or change. In some cases, a work around is possible by regenerating a new scenario in which one of the roles is played by a different object. In other cases, a new object is created which acts as a surrogate for the original, except that one of the object visibility roles is played by a different object.

Analysis for this larger model was tractable and the results are presented in the next section.

5.7 SBAC Model Capture and Analysis Tool

To support the investigation of scenario-based access-control modeling, a model capture and analysis tool has been constructed. The tool is a feasibility demonstrator

used to support the research. The tool can be used in conjunction with an object-oriented analysis or design process to capture an SBAC security model.

The tool provides a menu driven, multi-window interface that can be used to capture all elements defined in the modeling scheme for modeling a static security policy, $\mathcal{P}$, and an initial state. Therefore, a complete model captured by the tool models a system, Σ.

The tool also produces an analysis tree for the modeled system. The creation of an analysis tree provides metric information about the topology of the tree. There is also a rudimentary tree browser that provides the user with a visual representation of the tree. The analysis portion of the tool was developed to provide metric information about the size of analysis trees. The purpose of the metrics is to provide some empirical investigation of the tractability of generating analysis trees. Presently, the tool does not capture the parameters of a safety question nor does it perform a search on an analysis tree to answer a safety question. These are straightforward modifications to the tool.

The user can use the tool in a standalone mode to develop and analyze SBAC models. The tool can also be used in conjunction with the Rational Rose [Rat98] modeling tool. The user can provide additional SBAC annotations to a UML model as he is using Rose. The resulting Rose model file can be imported by the SBAC tool. A complete static security policy, $\mathcal{P}$, can be captured this way. The SBAC tool can then be used to define an initial state and generate an analysis tree. An imported model can also be displayed and modified using the SBAC tool interface.

5.7.1 Specification and Implementation of the SBAC Tool

The scope of the SBAC model capture and analysis tool can be briefly illustrated by presenting its use cases. Use cases are used to model the interactions or dialog

a user has with a system. They define the functional capabilities a system should have, from the viewpoint of the user. The following list describes the use cases for the SBAC tool. Each use case specifies a specific requirement for the tool from the user's perspective. These use cases were used to drive the development of the SBAC model capture and analysis tool.

Specify Security Policy To specify the system security policy the user shall be able to provide the modeling information for each of the following model components:

- object types
- object visibilities
- scenario types
- scenario parameter type bindings, including:
  - $parain_{ST}$
  - $paraout_{ST}$
  - $definedpara_{ST}$
  - $childpara_{ST}$
  - $definedpara_{S}$
- scenario descriptors

The user shall interact with a set of window-based forms to provide the modeling information. When data is present in the system model it shall be displayed to the user as a set of choices in a list or pop-up box (*e.g.* object types, scenario types, etc.).

Parse Rose Model The user shall be able to populate the security model by specifying a Rose model file for parsing. The tool shall read and parse the file to

identify and load the model components. This is an alternative method for specifying the system security policy.

Browse Security Policy The user shall be able to pick a scenario type from the list of scenario types in the model. This scenario type and all of its associated components (*e.g.* parameters, scenario descriptors, ordering, etc.) shall be displayed. The user shall be able to pick an object type from the list of object types in the model. This object type and all of its associated components (eg. visibilities, etc.) shall be displayed. While browsing a scenario type or object type the user shall be able to select the name of another scenario type or object type being presented. The user shall then be able to request that the specification for the selected model component be displayed concurrently with the first specification.

Specify Initial State To specify the system initial state the user will provide the modeling information for each of the following model components:

- initial objects and their types
- initial object visibilities
- the initial scenario and its type
- the parent-acquired parameter bindings for the initial scenario

The user shall interact with a set of window-based forms to provide the modeling information. When data is already in the system model it shall be presented to the user as a set of choices in a list or pop-up box (*e.g.* object types, scenario types, etc.).

Browse Initial State The user shall be able to choose to display the initial scenario. The scenario and all of its associated components (*e.g.* para, order, etc.) shall

be displayed. The user shall be able to pick an object from the list of objects in the model. This object and all of its associated components (eg. visibilities, etc.) shall be displayed. While browsing the initial scenario or an initial object the user shall be able to select the name of another scenario type, object type, or object being presented. The user shall then be able to request that the specification for the selected model component be displayed.

Generate Analysis Tree When requested, the tool shall generate the fully unfolded state from the current system security policy and initial state. The SBAC unfolding algorithm shall be used. A visual indication (heartbeat) shall be displayed to the user to indicate that the program is working and has not halted. This heartbeat might take the form of a running count of the number of scenarios generated.

Browse Analysis Tree The tool shall provide a graphical representation of the analysis tree. The user shall be able to expand or collapse branches of the tree by selecting a tree node with the mouse pointer.

The tool interface windows present existing model components in lists, and provide visual representations of the specific model components and their parameters. The user can edit the model using text fields, action buttons, pick lists, and menu selections.

The tool is implemented in Java 1.1.7 and Swing 1.0.3. The implementation is decomposed into 191 classes. There is a Rational Rose UML model for the implemented design. The tool implementation includes approximately 16 K-lines of source code.

5.7.2 Results of Model Analysis

The SBAC tool was used to capture security models for the examples presented earlier in the chapter. Specifically, the tool was used with the document release, project management, sales-order processing, and battlefield information system examples. The Rational Rose tool [Rat98] was used with the SBAC tool to develop the examples. The message sequence diagrams included in this chapter and in Chapter 2 are copied from the Rose models. The specially annotated Rose models provide the complete security policy for the examples. The examples were imported into the SBAC tool and initial states were defined in order to specify a complete system model for analysis.

The initial states for the example systems were chosen such that the resulting analysis tree exercised all of the scenario types specified for the example. In the case of the BIS example two different initial states are presented. In the first, BIS #1, the system is allowed to initialize itself. In this case the model creates all of the system objects, *i.e.* there are no O^0 objects. This models the system reinitializing itself from stored state information. In this case the current state of all the message lists, codes, etc. comes from recovering their state from secondary storage. The modeled system in BIS #1 is set up to recover two messages in each message list, two codes in the code list, and two users in the user list. The analysis of this model exercises the state recovery mechanism, which is useful, but it does not provide much room for formulating a safety question based on O^0 objects. The second BIS model, BIS #2, provides an initial state in which O^0 objects are specified for each of the lists (three objects per list). This provides a basis for formulating safety questions. In this case the initial scenario specified for the model is a scenario that occurs after the system initialization. *I.e.,* the model begins analysis in the middle of system execution. This choice of initial scenario allows the security modeler to isolate a portion of the system for detailed analysis.

TABLE 5.1. SBAC Analysis Results

Example Name	$\|ST_p\|$	$\|ST_c\|$	$\|O^0\|$	$\|S^u\|$	h_m	h_a	k_m	k_a	$\|S^u\|/\|ST\|$
Doc. Release	9	8	3	88	12	8.1	4	1.9	5.2
Proj. Mgmt.	8	9	11	38	7	5.2	4	2.5	2.2
Sales-order	23	12	6	86	11	8.5	6	3.5	2.5
BIS #1	90	58	0	1325	12	9.4	13	2.7	8.9
BIS #2	90	58	47	581	13	8.8	11	3.3	3.9

Table 5.1 presents the results of the modeling and analysis of the example systems. For each of the systems, the number of primitive scenario types ($|ST_p|$), and the number of complex scenario types ($|ST_c|$) in the static policy are presented in the table in order to provide an indication of the relative size and complexity of the model. The table also presents the number of objects specified by the initial state ($|O^0|$). The analysis results presented include the size of the analysis tree ($|S^u|$), the maximum and average path depth in the tree (h_m and h_a), and the maximum and average fan-out of the tree nodes (k_m and k_a). A ratio of analysis tree size ($|S^u|$) to the number of scenario types specified for the model ($|ST|$) is also presented in the table. This ratio is included to provided an indication of the effect of model size and complexity on the size of the resulting analysis tree.

This small set of examples provides some encouragement for the tractability of SBAC analysis. The size of the analysis trees is quite manageable in all of the examples presented here. This is much better than the theoretical worst case complexity. As discussed in Section 4.4.5, the worst case complexity for generation of the analysis tree is in the order $O(2^{n^2})$, where n is $|ST|$. The time complexity for generation of the tree has a linear relationship to tree size. With this complexity, even the smallest of the examples here is intractable in the worst case and would produce an unmanageable tree. The actual results here exhibit tree sizes that are quite manageable, and therefore also have efficient running times for tree generation. This result might be expected considering that the human designers that fashion a system use

object-oriented decomposition of the system requirements as a mechanism to control the complexity of the design. The standard approach is to capture these ideas informally or semi-formally in an object-oriented design notation such as UML. The object-oriented decomposition and object interaction scenarios the designers produce are organized with the purpose of restricting the complexity of the system design so that the designers can understand the system, communicate the design to others, and develop a understanding of the execution properties of the system. The designers tend to keep the relationships between their object interaction scenarios sparse in order to restrict the complexity of the design. SBAC formalizes these relationships by statically defining scenario-type-to-scenario-type relationships using scenario descriptors in $child_{ST}$. The sparse inter-scenario type relationships greatly reduce the analysis tree growth when compared to the worst case.

These empirical results do not provide a proof of tractability for SBAC modeling in general, but it is expected that there is a large class of systems for which analysis based on the unfolding algorithm is tractable.

Chapter 6

SUMMARY AND CONCLUSIONS

6.1 Introduction

The first part of this chapter presents some brief comparisons between the SBAC modeling scheme and other models in the literature. A comparison with the Typed Access Matrix approach [San92] and the Transformation Models [SG94] is presented because the techniques used in these models provide a foundation and inspiration for SBAC. There are also comparisons made between Clark-Wilson [CW87], RBAC and TBAC style modeling and SBAC. These comparisons are made because, like SBAC, these modeling schemes also address commercial integrity and legitimate-use policies. The similarities between SBAC scenarios and message sequence charts is also briefly discussed. By design, SBAC is closely related to OO modeling techniques. Message sequence charts and message sequence diagrams have been used by the OO community to capture object interactions. The discussion addresses why these are not felt to be adequate by themselves for expression of scenario specifications in SBAC.

The next section of the chapter discusses the advantages of SBAC modeling and its relevance to security modeling and to information system modeling in general. This is followed by an examination of some areas for future work, including direct extensions of the work presented and applications to related areas.

The final section of the chapter presents a summary of the research work and the conclusions.

6.2 Comparisons

6.2.1 A Comparison to TAM and TRM

The Typed Access Matrix approach [San92] and the Transformation Models [SG94] use strong typing (protection types) to specify static security policies for a system. This is a form of mandatory access control (MAC). Protection types are specified for subjects and objects. The commands that are able to modify the security state of the system require that type checking be done on all parameters. The immutability of protection types once subjects and object have been created, and the type checking on commands provide enforcement of the static policy. SBAC makes no differentiation between subjects and objects. Object types are used for the same purpose in SBAC as they are in the earlier modeling schemes, *i.e.* to specify static security policies for a system. Security state in a system changes as scenarios are authorized and terminated. Object and scenario types in SBAC are immutable. Type checking is applied to scenario parameters to provide enforcement of the static policy. The main difference between the earlier models and SBAC is that SBAC also includes an aspect of history checking to influence changes in security state. In TAM and TRM a change in security state is predicated upon the current set of access rights. In SBAC this is true also; an authorization is needed to perform a security relevant action. In SBAC however, an authorization is part of a scenario tree. The scenario tree provides a context for the authorization. The existence of an authorization by itself does not lead to any other authorization being possible. The authorization in the context of a scenario leads to other authorizations being possible. The same authorization can lead to different changes in security state in the context of different scenarios. Thus, history is important. To decide what is possible given a certain set of authorizations (access rights), the system must take into account how it arrived at the current state, *i.e.* what scenarios are currently active.

As are TAM models, SBAC models are expressive enough to specify multi-parent creation of objects. This is because the creation of a new object may have any number of other objects as parameters. Additionally, when a new object is created, other objects may immediately have permission to send it messages. The ability to express multi-parent creation allows SBAC to express polices like ORCON, which seem to require multi-parent creation.

One of the most important contrasts to make between SBAC and TAM is in their respective analysis schemes. Monotonic-TAM and its predecessor, the Schematic Protection Model (SPM), use an unfolding algorithm to construct a maximal state. The unfolding technique in these models is applied to subject creation and restrictions are applied to the subject-type-to-subject-type creation function to ensure that the resulting graph size is tractable. Subjects in the graph act as surrogates for all other subjects of the same type that may be in similar circumstances with respect to the protection state. The strategy in SBAC is similar but the unfolding algorithm does not create representative subjects; instead, it creates representative scenarios. Scenarios that are not currently active are past history, but are not destroyed. Scenario authorization is monotonic over time. The set of currently authorized scenarios is non-monotonic over time. This change in focus allows the use of an analysis strategy similar to that used with TAM and SPM to be used with systems with non-monotonic security policies.

6.2.2 A Comparison to Clark-Wilson

Clark-Wilson security models [CW87] identify certain objects as Constrained Data Items (CDIs). Access to these objects is provided solely through Transformation Procedures (TPs). SBAC is object-based. Objects are instances of abstract data types. As such all objects can be thought to be CDIs, and interface operations for

the objects to be TPs. In Clark-Wilson, TPs are defined to operate on a specific set of CDI types. In SBAC, interface operations are specified for an object type.

SBAC differs from Clark-Wilson in that Clark-Wilson does not constrain the type of the subject accessing a TP. SBAC does provide such type constraints. Also, Clark-Wilson does not specify when TPs become permitted for a subject and in what order they can become permitted. Such an ordering specification is explicit in SBAC models and is considered to be an essential aspect of modeling legitimate use.

6.2.3 A Comparison to RBAC

Recall from Section 2.2.18 that a role is a semantic construct around which access-control policy is formulated. In an RBAC policy, users are assigned to roles and permissions/rights are assigned to roles. Role authorizations are granted to users, or groups of users, based on the activities they are allowed to perform on system data. The Project Management Example presented in Section 5.3 is an example of SBAC being used to express a role-based security policy. A role-based policy also underlies the security policy of the Battlefield Information System Example, Section 5.6. These examples exhibit the basic requirements of models of type $RBAC_0$. They provide support for user to RBAC role mappings, for RBAC role to permission mappings, and for user sessions. The Project Management Example also illustrates how hierarchical RBAC roles can be modeled using SBAC. Thus, SBAC can directly express $RBAC_1$ security policies.

The permissions allocated by RBAC can have a high degree of data abstraction. The permissions typically allow a subject to execute a specific program on a specific type of data item. SBAC provides this kind of data abstraction and also provides a richer kind of procedural abstraction. The RBAC style roles modeled in SBAC authorize scenarios. A primitive scenario is similar to an RBAC permission. *I.e.*, it

can authorize a specific sender object to invoke an interface operation on a specific receiver object. An RBAC style role modeled in SBAC can also authorize a more complex scenario. This means the role can authorize that a specific set of tasks may be performed by users filling a specific set of roles. *I.e.,* where an RBAC permission can authorize that a specific task be done by a specific user filling a specific role, an SBAC permission can authorize that a specific set of tasks be done by a specific set of users filling a specific set of roles in a specific order.

$RBAC_2$ and $RBAC_3$ add constraints, which impose restrictions on acceptable configurations of the RBAC models. SBAC does not provide direct support for constraints, but in some cases the safety analysis scheme might be used to provide assurance that a constraint holds for a specific system model.

6.2.4 A Comparison to TBAC

Recall from Section 2.2.19 that TBAC provides a framework for active security models and enforcement from the perspective of activities and tasks. Permissions are constantly monitored, and activated and deactivated in accordance with emerging context associated with the progress of the tasks being performed. Permissions arise just in time for their use in the context of some authorized task. The fundamental abstraction is an authorization-step. An authorization-step represents a primitive authorization processing step and is the analog of a single act of granting a signature in a paper-based system. Permissions are associated with exactly one instance of an authorization-step. An instance of an authorization-step is associated with exactly one instance of a task. Each authorization-step maintains a protection state that is the set of permissions currently valid for that authorization-step. The concept of ordering of permissions is explicit in TBAC (through the specification of dependencies). All of these components (the components of $TBAC_0$) and the concept of

hierarchical tasks ($TBAC_1$) have explicit representations in SBAC. The relationship between TBAC and SBAC model components is described in the presentation of the Sales-order Processing Example in Section 5.4.

As with the $RBAC_2$ and $RBAC_3$ models, $TBAC_2$ and $TBAC_3$ add constraints that impose restrictions on acceptable configurations of the TBAC models. Again, SBAC does not provide direct support for constraints, but in some cases the safety analysis scheme might be used to provide assurance that a constraint holds for a specific system model.

The similarity between the SBAC and TBAC models derives from their similar goals, *i.e.* the modeling of legitimate use. Although developed separately from the framework of models described by TBAC, SBAC can be considered to be a modeling scheme that fits within this framework. SBAC additionally provides model representations that are complementary to contemporary object-oriented analysis and design methods and also provides a safety analysis scheme. An SBAC model is a detailed instance of a modeling scheme that fits within the TBAC framework.

6.2.5 A Comparison to MSCs

Object-oriented analysis and design methods have borrowed the notation of message sequence charts (MSCs) from the telecommunications protocol design community. The OO community uses a variant of MSCs, in some cases called message sequence diagrams (MSDs), to specify object interactions, or scenarios. SBAC augments the concept of scenarios in OO by adding rigor to the scenario-scenario relationships. This is a semantic issue that involves the interpretation of scenarios. Therefore, it is possible that some of the semantic issues related to MSCs also have some relevance to scenario-based access control. There are four semantic issues raised with respect to MSCs in Section 2.3.2.

One of the semantic issues raised is whether systems represented by MSCs have some finite set of global states with respect to message passing behaviour. There is unbounded creation of objects possible in SBAC. Therefore there are an unbounded number of global control states. However, by the arguments presented in Chapter 4 there are a finite number of *protection states*; *i.e.* there are maximal states for the pre-order on PA. The purpose of the semantic analysis of the security models presented here is not to provide an analysis of system evolution in general (global control state), but to provide an analysis of the evolution of system protection state. This is finite-state.

Another of the issues raised with respect to MSCs is that the structure of individual MSCs or the structure resulting from the composition of MSCs can result in non-local choices. That is, an MSC process may be required to execute a behaviour (or chose between behaviours) as a result of the occurrence of a message pass it could not itself observe. The structure of SBAC scenarios have the same properties leading to the possibility of non-local choice. The result is the same; non-local choices require either unbounded history variables to keep track of control choices (non-finite-state control) or MSCs which lead to non-local choices must be considered to be ill-formed. However, this is again a global control-state issue. It does not effect the safety analysis of the system.

The issue of messages being received in a different order than that in which they are sent is also presented as a difficulty with the semantics of MSCs. Such message crossings are not possible by the rules for construction of SBAC scenarios. This is not an issue in SBAC.

The last issue raised with respect to the semantics of MSCs was related to the completeness of the information available in MSCs to specify liveness properties. The authors of [LL95a] argue that liveness properties are difficult to specify with MSCs alone and in many cases such properties are better specified by temporal logic

formulae provided in addition to the MSCs. This work focuses primarily on safety properties and does not address liveness properties directly; however, the analysis tree may provide some assurance with respect to liveness properties. An analysis tree is a statement about possible histories of a system. One may test whether a temporal logic formula is satisfied by an analysis tree.

SBAC scenarios are similar to MSCs but there are significant differences. Scenarios allow messages to self, which are not allowed by MSCs. The most significant difference is in the respective composition mechanisms. The MSC standard allows modular design via sub-MSCs and decomposed process instances. This decomposition is process based. SBAC scenarios are object based and not process based. Modular design is denoted in the context of SBAC modeling by using child scenarios, which in MSCs would appear to be decomposed message events. This mechanism is more expressive than MSC conditions, which are another mechanism for composition of MSC diagrams. Specifically, the child scenario authorization mechanism allows more control with respect to scenario parameterization and ordering.

6.3 Discussion

The access-control modeling scheme proposed by this work provides a rich, expressive modeling capability that can be used to capture a broad range of useful security policies. SBAC supports policies that include least privilege, separation of duties, fine grained data abstraction, lattice-based MAC policies, role-based policies, and workflow policies.

SBAC also provides a safety analysis method. The analysis scheme can be used to characterize properties of the protection states that are possible for a specific model instance. The types of security policy to which the analysis method can be applied include non-monotonic policies.

The model comparisons earlier in this chapter emphasize that a major difference between SBAC and other modeling schemes is its support for considering the ordering of permissions when formulating a safety question. For a system, a safety question in SBAC considers not only if some subject may gain an access right to some object, but also the order of such an occurrence with respect to the occurrence of other access rights. It is a tenet of this work that specification and analysis of the order in which access rights occur is an essential aspect of modeling legitimate use. This is really a refinement of the principle of least privilege. The least privilege properties of SBAC models are more fine grained than those expressed by Clark-Wilson and role-based access-control polices. In this case more fine grained means that not only is a subject restricted to invoking a specific kind of operation on a specific kind of data type, but also that the permission to invoke the operation only arises at the moment it is required in the context of a statically defined scenario or task. This adds a just-in-time aspect to the concept of least privilege.

The theoretical worst case complexity for the analysis method indicates an intractable analysis even for small systems. However, the running time for analysis of an actual model instance is very sensitive to the scenario-type-to-scenario-type relationships defined by $child_{ST}$. It is expected that for a broad class of useful systems the connectivity between scenario types will be very much less than the complete connectivity implied by the worst case. The examples presented in Chapter 5 are representative of some interesting classes of system for which analysis is tractable.

The sparse connectivity between scenario types should be expected. Object-oriented decomposition and object interaction scenarios are organized with the purpose of restricting the complexity of system design. The standard approach is to capture scenarios informally or semi-formally using an object-oriented design notation. A contribution of this dissertation is the formalization of these relationships.

Security modeling using SBAC can be a complementary component of object-

oriented analysis and design. The proposed model makes extensive use of data that is already collected by commercial object-oriented analysis and design tools. The motivation is the productivity gain that may be realized through reducing the effort required in the maintenance of separate security and design models and in ensuring there is consistency between security models and other system models. A security-modeling tool that works with familiar system design tools also increases the likelihood that such modeling will be done. The SBAC analysis tool described in Section 4.4.5 is integrated with the Rational Rose [Rat98] OO modeling tool. A complete static SBAC security policy can be captured using the Rose tool. Much of the data needed for the security policy specification is provided by the OO analysis or design itself. Additional SBAC specific information is added as Rose documentation mark-ups. The resulting Rose model file can be imported by the SBAC tool. The SBAC tool can then be used to define an initial state and generate an analysis tree. An imported model can also be displayed and modified using the SBAC tool interface. The SBAC tool can also be used in a stand alone mode to develop and analyze SBAC models.

Another motivation for the relationship between SBAC modeling and OO analysis and design is that OO techniques model systems using abstractions and interactions closely related to the actual problem domain. Scenario-based descriptions of tasks and workflows provide an abstract and an intuitive way of specifying the access permissions required to complete a business process. Scenario-based models can be used at different levels of abstraction as the development of a system progresses from analysis to design to implementation. Although SBAC can be used at various levels of abstraction, model correspondence between the levels is a difficult problem that is not addressed by this dissertation.

6.4 Future Work

This section proposes directions for future research involving scenario-based access control. Several areas of research will be briefly discussed, including direct extensions of the work presented and applications to related areas.

An obvious opportunity for research is the continued development of the SBAC modeling and analysis tool. Presently the analysis tool has been used in support of gathering empirical data on the size of analysis trees for specific model instances. The tool does not provide support for formulating a safety question nor does it perform a search on the analysis tree to answer a safety question. Rudimentary support for these operations would be very easy to introduce into the tool. However, this kind of support should be based on an investigation of how a security engineer would use the tool to perform safety analysis. The selection of interesting, safety critical initial states for a system model is important. How the security engineer should be guided in defining a set of models that provide assurance of safety in a system is not clear. There needs to be a better understanding of what models and what safety questions provide a sufficient level of assurance that a system is safe.

It becomes apparent when discussing the degree of safety in a system or a level of assurance, that the behaviour of the system is measured against some security meta-policy. Investigation of a language for the expression of such policies is another area of future research. The security behaviour of an SBAC model could be analyzed with respect to the soundness of security axioms expressed by the meta-policy. For example, the designer of a system may want to express that for any initial state of the system it is not possible for someone to cash a cheque before it has been authorized for payment. It seems that a combination of deontic[1] and temporal logic would be useful to capture both what actions should be permitted by a system and in what

[1]Deontic logic is the logic of norms or morals. The logic provides expression for concepts such as 'what ought to be,' 'what is permitted,' 'what is obligated,' etc. For reference see [GMP92]

order those actions should be permitted.

When modeling the examples in Chapter 5 one finds that there is some difficulty in modeling certain aspects of object-oriented designs when using the current version of SBAC. One of the weaknesses is in the inability of SBAC to account for the behaviours associated with the type hierarchies present in the examples. The use of type hierarchies allows the reuse of scenarios to capture object interactions across families of related object types. Support for type hierarchies was not included in the SBAC modeling scheme in order to reduce the complexity of the modeling scheme during the initial stages of research. In keeping with the goal of providing a security modeling scheme that is complementary to contemporary analysis and design techniques, SBAC should be enhanced to handle type hierarchies.

Another place where there is some difficulty in modeling certain aspects of object-oriented designs is where design models make use of indirection in handling object visibility. It is common for the relationships between objects to change during the evolution of a system. The current version of SBAC requires that object visibilities be assigned at object creation time, that only existing objects can be visible to a newly created object, and that object visibilities cannot be changed after object creation. These restrictions simplify the computation of scenario equivalence class, and the complexity of the analysis tree. A weaker set of restrictions would allow the modeling scheme to be more expressive and is an interesting avenue for future research.

Another direction for future work is in the simplification of the analysis tree. Presently the analysis tree graph does not allow cycles. By design, the analysis tree graph contains redundant information. This allows the analysis tree to model the permission ordering relationships arising from multiple occurrences of equivalent scenarios. Simplifications and efficiencies are possible in the analysis graph by allowing cycles. The current graph is finite and tractable but a broader class of systems may be candidates for analysis if greater efficiencies are found. In simplifying the analysis

graph by allowing cycles, one would have to be careful to preserve the authorization properties of the original analysis tree.

6.5 Conclusions

The development of scenario-based access control is driven by two main goals. The first goal is to provide a scheme that will provide efficient safety analysis for systems modeling legitimate-use policies. This implies efficient analysis of non-monotonic systems. This is because legitimate-use policies that employ just-in-time availability of access-control permissions are inherently non-monotonic. The second goal is to provide a modeling scheme that complements contemporary software engineering modeling techniques. The objective is to leverage the information that is already being captured by such techniques and to provide security modeling as a extension to existing software engineering methods. This eliminates duplication of effort in security modeling and may serve to encourage the wider use of security modeling. These goals are met by the scenario-based access-control scheme presented by the dissertation. The contributions of this dissertation are:

- a scheme for the modeling of legitimate-use in information system security policies,
- a scheme for the safety analysis of inherently non-monotonic, legitimate-use-based security policy models,
- formal rigor applied to scenario diagrams of the relationships between interacting objects, and
- a security modeling scheme that is complementary to contemporary analysis and design techniques.

Chapter 2 provides a literature review and the contextual information necessary to provide background for the area of research, to provide foundations and inspirations for the work, and to provide examples for comparison. Chapter 3 describes the modeling of scenarios of interacting objects. The models provide rigor to the relationships between types of scenarios and for the relationships between scenario instances. Chapter 4 defines the concepts of security policy and system. This chapter presents a scheme for safety analysis of scenario-based access-control models. The analysis scheme has an intractable running time in the worst case. However, it is expected that for a broad class of useful systems the analysis is tractable. Chapter 5 presents a series of worked examples. The examples presented are representative of some interesting classes of system for which analysis is tractable.

REFERENCES

[Ada95] C. Adams. DoD information security: Near-term dissonance, long-term promise. *Military and Aerospace Electronics*, pages 14–20, August 1995.

[And72] J.P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. 1, USAF, 1972.

[AS90] P.E. Ammann and R.S. Sandhu. Extending the creation operation in the schematic protection model. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 340–348. IEEE, 1990.

[AS94] P.E. Ammann and R.S. Sandhu. One-representative safety analysis in the non-monotonic transform model. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 138–149, Franconia NH, June 1994. IEEE.

[BC92] P. Bieber and F. Cuppens. A logical view of secure dependencies. *Journal of Computer Security*, 1:99–129, 1992.

[BCC94] N. Boulahia-Cuppens and F. Cuppens. Asynchronous composition and required security conditions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 68–78, Oakland CA, May 1994. IEEE.

[Bel74] D.E. Bell. Secure computer systems: A refinement of the mathematical model. Technical Report MTR-2547, Vol. 3, MITRE, April 1974. also ESD-TR-73-278, Vol. 3.

[Bib77] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153 Rev. 1, MITRE, April 1977.

[BL73a] D.E. Bell and L.J. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE, November 1973. also ESD-TR-73-278, Vol. 2.

[BL73b] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE, November 1973. also ESD-TR-73-278, Vol. 1.

[BL75] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE, November 1975. also ESD-TR-75-306.

[BN89] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, Oakland CA, May 1989. IEEE.

[Boo94] G. Booch. *Object-oriented Analysis and Design with Applications.* Benjamin-Cummings, Redwood City CA, 1994.

[BSS+95] L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker, and S.A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 66–77, Oakland CA, May 1995. IEEE.

[BY95] W.R. Bevier and W.D. Young. A state-based approach to noninterference. *Journal of Computer Security*, 3:55–70, 1995.

[Com88] Communications Security Establishment. *The Canadian Trusted Computer Product Evaluation Criteria, Version 3.0e.* Government of Canada, January 1988.

[Cos98] J. Costello. Battlefield information system: An application of design patterns in object oriented software. Term paper in graduate course, EE573-Royal Military College of Canada, 1998.

[CW87] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, Oakland CA, April 1987. IEEE.

[Den76] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Den82] D.E.R. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, 1982.

[Dep85] Department of Defence. *Department of Defence Trusted Computer System Evaluation Criteria (Orange Book), DOD5200.28-STD.* Government of the U.S., December 1985.

[FLR77] R.J. Feiertag, K.N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 57–65, New York, November 1977. ACM.

[Gas88] M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold, New York, 1988.

[GM82] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland CA, April 1982. IEEE.

[GM84] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 75–86, Oakland CA, April 1984. IEEE.

[GMP92] J.I. Glasgow, G.H. MacEwen, and P. Panangaden. A logic for reasoning about security. *ACM Transactions on Computer Systems*, 10(3):226–264, August 1992.

[HR78] M.A. Harrison and W.L. Ruzzo. Monotonic protection systems. In *Foundations of Secure Computation*, pages 337–365. Academic Press, New York, 1978.

[HRU76] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. In *Communications of the ACM*, pages 461–471. IEEE, August 1976.

[Int88] International Telecommunications Union - Telecommunication Standardization Sector. *ITU-TS Recommendation Z.100: Specification Description Language (SDL)*. ITU-TS, Geneva, 1988.

[Int94] International Telecommunications Union - Telecommunication Standardization Sector. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1994.

[Jac88] J. Jacob. Security specifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 14–23, Oakland CA, April 1988. IEEE.

[Lam71] B.W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton NJ, March 1971. reprinted in Operating Systems Review, 8, 1, ACM, January 1974, 18-24.

[Lee88] T.M.P. Lee. Using mandatory integrity to enforce "commercial" security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 140–146, Oakland CA, April 1988. IEEE.

[Lip82] S.B. Lipner. Non-discretionary controls for commercial applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–10, Oakland CA, April 1982. IEEE.

[LL94] P.B. Ladkin and S. Leue. What do message sequence charts mean? In R.L. Tenney, P.D. Amer, and M.U. Uyar, editors, *Formal Description Techniques VI, IFIP Transactions C, Proc. of the 6th International Conference on Formal Description Techniques*, pages 301 – 316. North-Holland, 1994.

[LL95a] P.B. Ladkin and S. Leue. Four issues concerning the semantics of message flow graphs. In *Formal Description Techniques VII, Proc. of the 7th IFIP International Conference on Formal Description Techniques FORTE'94*. Chapman and Hall, 1995.

[LL95b] P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, September 1995.

[LS78] R.J. Lipton and L. Snyder. On synchronization and security. In R.A. DeMillo, D.P. Dobkin, A.K. Jones, and R.J. Lipton, editors, *Foundations of Secure Computation*, pages 367–385. Academic Press, New York, 1978.

[Mau96] S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996.

[McC87] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, Oakland CA, April 1987. IEEE.

[McC88] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–186, Oakland CA, April 1988. IEEE.

[McL87] J. McLean. Reasoning about security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 123–131, Oakland CA, April 1987. IEEE.

[McL90] J. McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–187, Oakland CA, April 1990. IEEE.

[Min78] N. Minsky. On synchronization and security. In R.A. DeMillo, D.P. Dobkin, A.K. Jones, and R.J. Lipton, editors, *Foundations of Secure Computation*, pages 255–276. Academic Press, New York, 1978.

[Min84] N. Minsky. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.*, 6(4):573–602, October 1984.

[MS88] J.D. Moffett and M.S. Sloman. The source of authority for commercial access control. *Computer*, pages 59–69, February 1988.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1053–1058, December 1972.

[R+91] J. Rambaugh et al. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[Rat98] Rational Software Corporation. Rational Rose 98 Modeler Edition, 1998.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[Ros95] A.W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–127, Oakland CA, May 1995. IEEE.

[San88] R.S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, pages 404–432, April 1988.

[San89] R.S. Sandhu. Transformation of access rights. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 259–268, Oakland CA, April 1989. IEEE.

[San92] R.S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, Oakland CA, May 1992. IEEE.

[San93] R.S. Sandhu. Lattice-based access control models. *IEEE Computer*, pages 9–19, November 1993.

[San96] R.S. Sandhu. Access control: The neglected frontier. In *Proceedings of the 1st Australasian Conference on Information Security and Privacy*, Wollongong, Australia, June 1996.

[San98] R.S. Sandhu. Role-based access control. In *Advances in Computers, Vol. 46.* Academic Press, New York, 1998.

[SCFY94] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control: A multi-dimensional view. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 54–62, Orlando FL, December 1994. IEEE.

[SG94] R.S. Sandhu and S. Ganta. On the minimality of testing for rights in transformation models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 230–241, Oakland CA, May 1994. IEEE.

[Smi93] J. Smith. Privacy policies and practices: Inside the organizational maze. *Communications of the ACM*, 36(12):105–122, December 1993.

[SS92] R.S. Sandhu and G.S. Suri. Non-monotonic transformation of access rights. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 148–161, Oakland CA, May 1992. IEEE.

[SS94a] R.S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, pages 40–48, September 1994.

[SS94b] R.S. Sandhu and G. Srinivas. On the expressive power of the unary transformation model. In *ESORICS 94: Third European Symposium on Research in Computer Security*, pages 301–318, Brighton U.K., November 1994. Springer.

[TS94] R.K. Thomas and R.S. Sandhu. Conceptual foundations for a model of task-based authorizations. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 66–79, Franconia NH, June 1994.

[TS97] R.K. Thomas and R.S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization mangement. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, Lake Tahoe CA, August 1997.

[VC94] V. Varadharajan and C. Calvelli. Tickets and authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 213–229, Oakland CA, May 1994. IEEE.