A MODEL FOR A SECURE PROGRAMMING ENVIRONMENT

MARTIN L. KERSTEN

Promotor	: Prof. Dr. R.P. van de Riet
Copromotor	: Prof. Dr. A.I. Wasserman
Referent	: Prof. Dr. I.S. Herschberg

to Fernande and Joost

TABLE OF CONTENTS

SUMMARY

In this thesis a model for a secure programming environment (SPE) is developed to study access control concepts in high-level programming languages, the compilation and linkage of such programs, and their embedding in a supportive operating system. The motivation for this research originated in the design of a high-level programming language for the construction of interactive information systems where a secure implementation is a necessity for guaranteeing privacy to the users. In particular, one should ensure that a system contains both accurate information and that system usage is in accordance with an access control (flow) policy. The SPE model developed in this thesis addresses both the formal aspects of the protection philosophy for such an environment and its consequences for a verifiable implementation.

The SPE model components encompass the entity classes *users*, *objects*, and *regions*; a collection of protection state mappings; and a mapping composition and abstraction facility. The *users* represent the persons or their agents responsible for triggering protection state modifications. The *objects* represent the entities manipulated by the (automated) information processing system. Their semantics are largely ignored during the formalization of SPE model. The *regions* represent protection domains, a grouping mechanism for objects and users considered equivalent under a given access control policy.

The access control policy associated with the SPE model is administered by five relations:

- A relation between users and regions representing ownership;
- A relation between objects and regions representing object definition;
- A relation between regions representing access flow constraints;
- A relation between regions and objects representing acquisition of access rights;
- A relation between regions and objects representing granting of access rights.

Instances of the entity classes and relations define three overlapping sets of protection states; the *consistent* states, the *acceptable* states, and the *valid* states. A state is considered consistent when the elements used in the construction of the

five relations mentioned above are declared as entities. The acceptable states satisfy an elementary protection scheme; each region is associated with a user held responsible for it; each user owns at least on region; and each object is defined in precisely one region. Valid states model proper sharing of access rights on objects, that is, those in accordance with a given flow constraint.

In Chapter 4 it is shown that these state constraints can be used as invariants to characterize secure state transformations. The analysis shows that the algorithmic cost of guaranteeing access control security for state transformations extending the protection state is negligible. The corresponding cost for state transformations reducing the protection state is limited by the size of the state representation.

In Chapter 5 a set of state transformations and an authorization policy formally define an SPE instruction set, which is shown to be *well-defined*, i.e. it is minimal, each secure state is derivable, and the effects of each instruction can be undone. The protection range obtained by this instruction set is analyzed, leading to conditions for sharing and stealing access rights. Moreover, the model and its instruction set are used to simulate two existing theoretical protection models to illustrate and to emphasize the differences in approach taken. It shows that SPE is equally powerful in answering the security questions posed by these models, but differs by providing access flow constraints and a general abstraction mechanism to construct multi-level protection systems.

In Chapter 6 the SPE model as a unifying protection model for secure systems implementations is illustrated. First, a sketch of a secure distributed computer system architecture is given where the central role of SPE lends itself to a kernelized implementation and verification. Second, the knowledge obtained from the theoretical analysis is used to evaluate protection issues in the programming language PLAIN. Third, the SPE model indicates a secure infrastructure for a project development environment, thereby providing a formal basis for existing module definition languages. Finally, static and dynamic aspects of PLAIN programs are revisited to sketch a verifiable secure implementation.

As a result the SPE model provides both a formal framework for studying access control and an implementation direction for a secure programming environment.

SAMENVATTING

Dit proefschrift beschijft een model voor een veilige programmeeromgeving (SPE) waarmee het gebruik van objecten in hogere programmeertalen, de vertaling en bundeling van programmas, en de integratie met het bedrijfssysteem kunnen worden beheerd en geanalyseerd. Het SPE model is een uitvloeisel van het ontwerp van een hogere programmeertaal voor het bouwen van interactieve informatiesystemen, waarin de privacy van de gebruikers controleerbaar moet kunnen worden gewaarborgd. In het bijzonder moet de toegang en het gebruik van gegevens in een informatiesystemen kunnen worden gereglementeerd. In dit proefschrift wordt zowel aandacht geschonken aan de theoretische aspecten van de beveiliging in een programmeeromgeving als aan de implicaties van het SPE model op de realisatie van zo'n omgeving.
Het SPE model omvat de verzamelingen entiteiten *gebruikers*, *objecten* en *regionen*; een verzameling toestandsovergangen; en een compositie mechanisme voor toestandsovergangen. De *gebruikers* corresponderen met personen, of hun representanten, verantwoordelijk voor het wijzigen van de gebruiksrechten. De *objecten* representeren de entiteiten die door het informatiesysteem kunnen worden gemanipuleerd. Tijdens de formalisering van het beveiligingsmodel is geen rekening gehouden met de semantische eigenschappen van deze objecten. De *regionen* modelleren beveiligings-domeinen, waarin gebruikers en objecten aan elkaar worden gerelateerd wanneer zij vanuit het oogpunt van beveiliging een identieke rol vervullen. De distributie van gebruiksrechten wordt in het SPE model beschreven door vijf relaties:

- Een relatie tussen gebruikers en regionen om het begrip eigenaar te modelleren;
- Een relatie tussen objecten en regionen om objectdefinitie te modelleren;
- Een relatie tussen regionen onderling om transport van gebruiksrechten te modelleren;
- Een relatie tussen regionen en objecten om acquisitie van gebruiksrechten te modelleren;
- Een relatie tussen regionen en objecten om verlening van gebruiksrechten te modelleren.

De mogelijke beveiligingstoestanden vormen drie overlappende verzamelingen; de *consistente* toestanden, de *acceptabele* toestanden, en de *valide* toestanden. Een toestand is consistent als de entiteiten genoemd in de relaties ook als zodanig zijn gedefinieerd. De acceptabele toestanden voldoen aan het volgende criteria; bij elke regio hoort een gebruiker die verantwoordelijk is voor de beveiliging; omgekeerd heeft elke gebruiker tenminste een regio; en elk object is in precies een regio gedefinieerd. Valide toestanden modelleren het veilig verspreiden en verkrijgen van de gebruiksrechten.

In hoofdstuk 4 wordt aangetoond dat de eigenschappen van SPE toestandsbeschrijvingen kunnen worden gebruikt als invarianten voor veilige toestandsovergangen. De algorithmische kosten voor het garanderen van veilige transport van gebruiksrechten is verwaarloosbaar, terwijl de algorithmische kosten bij toestandsovergangen die gegevens uit de SPE toestandsbeschrijving verwijderen beperkt blijft tot de omvang van de toestandsbeschrijving.

Een verzameling toestandsovergangen en een autorisatiebeleid definieren tezamen een SPE instructieset, een voorbeeld hiervan wordt in hoofstuk 5 ingevoerd. Er wordt aangetoond dat deze instructieset *welgedefinieerd* is, dat is, minimaal in aantal instructies, elke veilige toestand is afleidbaar, en de effecten van de toestandsovergangen kunnen ongedaan gemaakt worden. Vervolgens worden aan de hand van deze instructie set een aantal beveiligingsvraagstukken besproken, zoals onder welke voorwaarden gebruiksrechten kunnen worden gestolen. Het SPE model wordt vergeleken met twee bestaande theoretische modellen, het Take-Grant model en het Harrison-Ruzzo-Ullman model. Beide modellen kunnen met SPE nagebootst worden, waardoor het SPE model tenminste de beveiligings mogelijkheden van beide biedt. Het is echter niet mogelijk het SPE model na te bootsen in beide modellen, omdat SPE meer mogelijkheden biedt voor het construeren van gelaagde beveiligingssystemen.

In hoofdstuk 6 wordt aangetoond dat het SPE model als overkoepelend theoretisch model fungeert voor verschillende praktische toepassingen. Dit wordt op vier manieren geillustreerd. Allereerst wordt een schets gegeven van een veilig gespreid bedrijfssysteem waarin het SPE model de basis vormt voor de nucleus en voor het bewijzen van een correcte implementatie. Ten tweede, de theoretische analyse wordt gebruikt om een aantal tekortkomingen van de programmeertaal PLAIN te belichten en op te lossen. Vervolgens wordt het SPE model gebruikt als basis voor een indicatie van een veilige PLAIN programmeeromgeving. Ten slotte worden de SPE machine- en programmeertaal PLAIN in samenhang bekeken, waarbij de statische en dynamische aspecten in termen van SPE worden gedefinieerd.

Hiermee is aangetoond dat het SPE model dienst kan doen als theoretisch kader voor beveiligingsanalyses en voor de constructie van veilige prommeeromgevingen.

ACKNOWLEDGEMENTS

The research reported in this thesis reflects my involvement in the area of computer security over the last six years under the direction of prof. R.P. van de Riet. He pointed out the need of a formal model to unify access control protection in information systems and he put a lot of effort in keeping me focussed on that target. Prof. A.I. Wasserman stimulated my research in the area of information systems, especially the design and implementation of tools for their contruction, such as PLAIN, Troll/USE, TBE/USE and Trump which are reported elsewhere. Under his guidance my English prose and the applicability of the SPE model have been improved considerable.

Prof. Dr. I.S. Herschberg owes my gratitude for the time spent on reading preliminary versions of this manuscript and for the many constructive comments which improved its readability and the consistent use of terminology. Drs. S. Mullender was instrumental in preparing the material for the phototypesetter.

The present work has been made possible by a stipendium granted by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) from September 1979 up to August 1983. I am endebted to the Centrum voor Wiskunde en Informatica in Amsterdam for using their text processing facilities.

CURRICULUM VITAE

Naam:	Kersten, Martin L.
Geboren:	25 oktober 1953 te Amsterdam
1972:	Diploma HBS-B, Pascal Scholengemeenschap te Amsterdam
1976:	Kandidaatsexamen Wiskunde, Vrije Universiteit te Amsterdam
1979:	Doctoraalexamen Wiskunde, Vrije Universiteit te Amsterdam
1979-1983:	Promotie medewerker, ZWO
1983-heden:	Wetenschappelijk medewerker, Vrije Universiteit te Amsterdam

Current address of the author:

Subfaculteit Wiskunde en Informatica
Vrije Universiteit
De Boelelaan 1081
1081 HV Amsterdam

1

INTRODUCTION

Many computer systems currently in use contain and manipulate large quantities of information concerning individuals and organizations. Although the storage and processing of this information can be traced back many centuries, its potential impact on one's daily life is becoming more apparent. Its interference with our lives imparts a feeling of danger, triggered by the seemingly unrestricted capabilities of modern information processing systems. In the past we could rely on both the shortcomings of communication and information storage media, as on the fragmentation and distribution of information such that "Nobody knows enough of me to significantly influence my daily life". We are now faced by the technical feasibility of its integration to complete the picture on one's behavior, i.e. "Big brother is watching you" is a real threat.

Similarly, organizations nowadays heavily rely on the proper working of their information processing systems. The last decade has shown that such a reliance may turn out to be costly if not based on proper certification of the system software [Herschberg84]. Many cases of computer fraud, to cover up embezzlement or theft of information, have been reported in the literature [Parker76, Chambers78, Krauss79].

These effects have triggered much research centered around the topic of 'privacy and security in computer systems' over the last decades. Privacy in this context is understood as the right of individuals, groups, or organizations to control the collection, use, and dissemination of information about them. As such, its issues are primarily social-political, rather than technological. The question of how far to go in computer-based record keeping on people is a political-social question in which the rights, needs, and interests of the individual should be weighted against that of the institutions (companies, organizations, government). Since the early 1970's, privacy protection laws and regulations

have been enacted in several industrialized countries to control the collection, use and (transborder) dissemination of personal data about individuals [Turn80]. Yet, the continual change in hardware and software systems, the impact of computerized communications, the widespread distribution of inexpensive home computers and communication equipment, and changes in attitude towards information processing complicates the further development and enforcement of such laws [Kamer82, Kuitenbrouwer79, Leerkamp82, Turn80].

Although the threat to privacy primarily requires legal and regulatory remedies, technology is required to enforce them. This is where the term computer security comes into play. Computer security is the protection of computer resources against the unauthorized use, access, and modification. This pertains to both data files and software. Due to the variety of computer systems, hardware devices, software systems, programming languages, etc., the security problem is a multi-facetted problem. Solutions are required for each facet and they must be combined in an effective way. One important facet of computer security, called access control, is the focus of this thesis.

1.1. Concepts and terminology

A basic aspect of all secure systems is the regulation of activities of persons on computer systems. This poses the problem how a system recognizes or identifies a person, i.e. *authenticates* its users. A fairly standard mechanism used for this purpose is some kind of a password scheme, which requires the user to enter a user name, or account number, and a password. If the user name is valid and the password matches the one associated with the user name, the user is given access to the computer system. The major threat to security in this context is *masquerading*, an intruder gains access to the system by using another user's account number and password. Numerous recent newspaper stories illustrate these threats.

Password schemes are not only used to authenticate users on a single computer system, but they can also be used to authenticate users from other computers linked together in a computer network. If there were no authentication in a communication network, it might be possible for a person to interlope on the communication line and thereby steal the permissions granted to the legitimate user. An illustration of the vulnerabilities of computer networks is reproduced from [Hoffman77] in Figure 1.1.

Ample techniques have been designed over the last decades to reduce the risk associated with masquerading and interloping [Hoffman77] and therefore these topics are not further analyzed in this thesis. We assume that a mechanism exists such that a legitimate user can gain access to the (distributed) system.

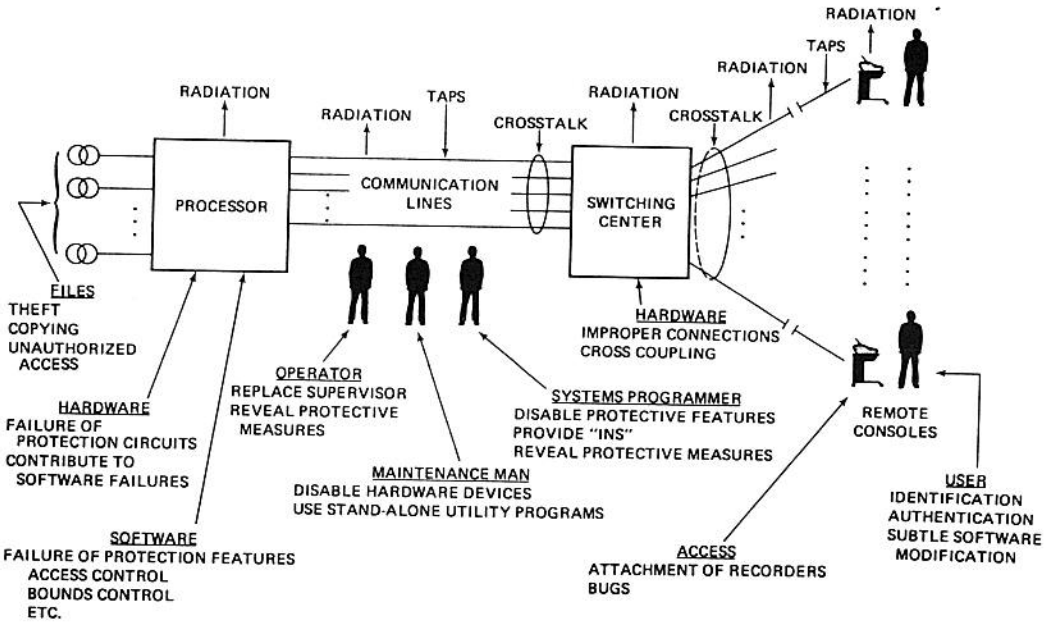


Figure 1.1 Computer network vulnerabilities

The different aspects of computer security shown in Figure 1.1 can be further classified according to their nature into *physical*, *data*, *operational* and *organizational* security. *Physical security* deals with the physical threats to computer system hardware and environment, such as fire, attacks, malicious entrance, bombing etc. A classic work on physical security is [Martin73].

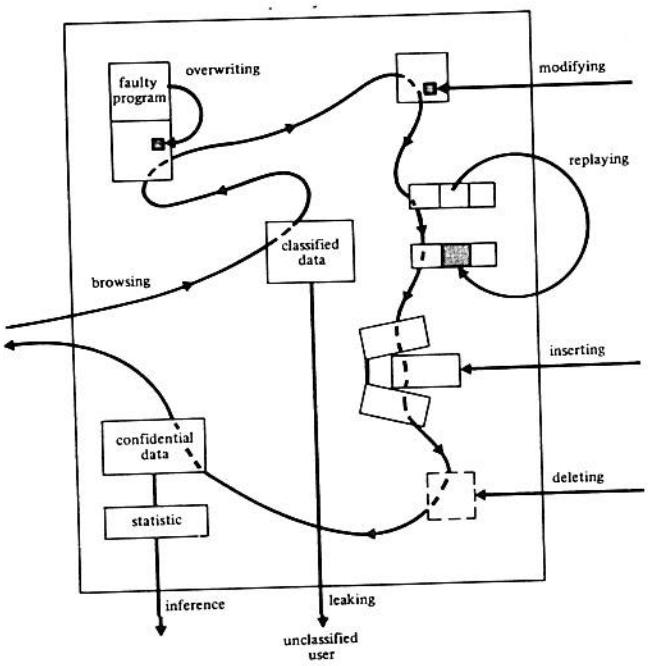


Figure 1.2 Threats to data stored in computer systems

Data security deals with the protection of data stored within a computer system and communicated between systems. Its major threats are illustrated in Figure 1.2 after [Denning82]. Browsing refers to searching (confidential, proprietary) data within a system. Programs may threaten to overwrite data by accident (faulty programs). Leakage refers to the transmission of data to unauthorized users by processes having legitimate access. Inference refers to the deduction of confidential information about an individual by correlating released statistical data. Operations such as insertion, deletion, and modification by unauthorized users threaten the integrity of the data stored.

Operational security deals with the organizational policies and techniques to limit and control access to the computer systems. User authentication and communication networks fall into this category and they are dealt with to avoid masquerading, piggy-backing, and interloping. Techniques for proper operational use of the computer systems can be found in textbooks such as [Martin73, Hoffman77]

Organizational security deals with the way computerized data is acquired, stored, manipulated, and flows in the organization. It differs from the previous security dimensions by emphasizing the organizational structure rather than the computer system. Its methods for security provisions are mostly taken from the auditing world, dealing with personnel practices and policies, physical access regulations, documentation standards, change control for applications and

system software, etc. A thorough discussion of these topics is given in [Krauss79].

Although no computer system can be considered secure without providing methods and techniques in each of the four major security problem areas, it is data security that can be considered the nucleus of computer system protection. The basis for data security is formed by an access control model and an authorization scheme. The former describes how data objects and relations between data objects are administered in a computer system and how they may change over time. The latter controls the use of access rights by users or processes, that is, it regulates the use of the computer system.

1.2. Subject of this thesis

Despite the large body of literature dealing with access control mechanisms and authorization schemes, very little of this work abstracts from the semantics of the underlying objects, machine architecture, and system software layer. We feel that more work is needed on access control, emphasizing the secure, manageable flow of access rights, for we consider such a prerequisite for the implementation of specific authorization policies.

In this thesis, we define an access control model, focusing primarily on interactive information systems, an important application domain. They provide users of a computer system interactive access to data files and databases, possibly using customized programs to ensure database integrity and implementing domain specific data manipulations functions.

The data security problems are manifested differently in an information system, because they make use of the protection properties of the hardware, the operating system, and the programming language. In particular, user authentication and access control as provided by the operating system are often insufficient to guard information systems against data security breaches. For one thing, data security depends as much on the value of the data in a specific environment as on the name or location of its representation

A second important protection objective in information systems is selective sharing of access rights. That is, access rights are granted and revoked by users. The dominant problem is how to guarantee proper access usage and foresee the impact of one's access decisions.

A third protection objective in information systems is to safeguard confidential or derivable information from data being stored. Although this problem can not completely be solved by an access control policy, such a policy is a necessary prerequisite. The access control model defined in this thesis shows that the problems sketched can be formalized and thereby become attainable for theoretical analysis. Using a formal model rather than English text serves the following purposes.

- A formal model provides a framework for discussion and improved understanding by concentrating on the salient features of a problem without considering the details involved in specific implementations.
- A formal model provides a basis for theoretical studies of security and allows security properties to be proved.
- A formal model abstracts from the peculiarities of specific machines and programming environments and therefore can be used as a standard or reference model for actual implementations.
- A formal model makes it easier to compare different secure system implementations.

A formal model by itself is not a panacea for underlying real world security problems. A formal model has shortcomings as well:

- The formalism used for definitions, theorems and proofs is often hard to read, while the proofs are not necessarily error free.
- A small fraction of the protection problems can be effectively translated into a formal model.
- A formal model mostly does not guide an efficient implementation.

1.3. Outline of thesis

Chapter two introduces the various approaches described in the literature dealing with access control. A framework is presented to illustrate the approaches taken over the last two decades. In chapter three we informally introduce the SPE model, expand on the motives for its choice, and indicate the results of a formal analysis. The SPE model is formally defined in chapter four, where the behavior of a secure abstract machine is cast into a series of theorems and conditions for a well-defined protection system instruction set. Chapter five starts with the formal definition of a particular instruction set, which is used thereafter to analyze alternative granting/revocation policies. Moreover, the instruction set in combination with an abstraction mechanism is used to compare our model with some existing theoretical access control models. Chapter six interprets the model to direct the design of a distributed operating system, programming language, and programming environment. Chapter seven concludes with a summary and indication of future research issues related to the SPE model, both regarding its formal specification and how the results may direct future investigations in this area.

2

A TAXONOMY OF ACCESS CONTROL MODELS

2.1. Historical overview

Protection in computer systems has been studied during the last two decades from many angles and a staggering amount of literature exists covering the aspects of access control, authorization policies and confidentiality. Example surveys are [Hoffman69, Anderson79, Linden76, Landwehr81, Landwehr83, Cheheyl81]. Maturing of this field is observed from the lecture books currently available, both surveying the field and providing general background [Hoffman77, Martin73, Hsiao79, Fernandez81, Denning82] as books on special topics, such as computer fraud and abuse [Parker76, Krauss79]. Conferences on these topics are organized on a regular basis by International Federation of Information Processing (IFIP/Sec) [Fak83] and Technical Committee on Security and Privacy of the IEEE Computer Society [Davida80], while a session is dedicated to the issue in many international conferences. Hardware support for secure systems is available in the form of general purpose capability-based processors [Pollack82], fault-tolerant systems [Borr81], and encryption/decryption hardware for secure communication [DES77].
Although research activities and influences cannot be pinpointed to exact dates and therefore a survey on these activities can not be complete nor unbiased, it is possible to indicate the shifting interest over the last two decades in this area of computer science.

Access control in the sixties was simplified by the batch oriented nature and single instruction stream in most computer systems. Files and programs resided

for a short period of time in the machine and access rights were determined primarily by handing the proper card decks to the desk clerk. Research activities concentrated on the architecture of multiprogramming systems to achieve a better system utilization. Segmented memory was invented to safeguard the programs concurrently occupying main store and memory tags were used to safeguard individual programs against unintended use of the memory locations [Iliffe62]. In combination with the introduction of multiple operating states of the machine, such as user state and supervisory state, it became possible better to control the multi-programmed system behavior. A generalization of the concept of supervisor state resulted in the MULTICS operating system [Graham68, Organick72]. These descriptor-based machines are forerunners of capability-based systems, where addresses of all objects are replaced by codewords which include an encoding of the access permissions. A conceptual design for a capability supervisor appeared in 1966 [Dennis66], while the first industrial capability hardware and software system appeared in 1969 [England72]. In the same period many of the basic threats to security in computers were discovered, classified, and cast into a system penetration methodology [Ware67].

The early seventies show increased activity in the formalization of protection problems and practices. The multi-level security in military environments led to the design of the Bell-LaPadula [Bell74] model (See also [McLean85] for a fundamental critique on this model), which was seminal for the development of many prototype implementations using a security kernel [Schroeder77, Berson79]. Capability-based protection and the access matrix model [Lampson69] led to the Take-Grant model [Jones76] and the operational model of Minsky [Minsky78b, Minsky77]. These are analytical models used to describe and investigate the protection system properties and do not impose an implementation technique. The theoretical limits of safety analysis were uncovered by Harrison-Ruzzo-Ullman [Harrison76] using the access matrix framework.

This period also shows increased awareness on the different aspects of protection. Access control is recognized as one dimension of the problem; equally important aspects are the authorization policy, information flow control, and inference control. The intricate role of information flow in relation to protection issues is first recognized and described by Lampson [Lampson73, Fenton74].

Access control within operating systems based on the access matrix is limited to data-independent access. By contrast, in database management systems access control takes into account the granularity of the objects being protected, the level at which information is manipulated, and the semantics of the information being stored. Moreover, an authorization policy to regulate the sharing of permissions is required. Formal approaches in that field include [Hartson76, Fernandez75], pragmatics are found in [Stonebraker74,

CODASYL78].

Access control is transferred to the programming language arena by Conway [Conway72a] , who addresses protection issues in information systems, and Morris [Morris73], who translates capability protection to programming language primitives. This work is extended by Liskov and Jones in [Liskov78] to provide full capability-based protection in typed high-level programming languages and analyzed formally on its information flow characteristics by D.Denning [Denning76].

During the late seventies, it was recognized that simple protection models alone are not sufficient to describe and analyze the behavior of complex systems. The design of a secure software system depends as much on the reliability of the code, i.e. absence of software bugs, as on the protection policy. Therefore, work in the area of operating system security resulted in increased interest in the practicality of formal program specification and verification techniques [Cheheyl81, Levitt79, Thompson81, Egge80]. This influence is notable in the design, verification and implementation of secure operating systems like KSOS [McCauley79] , PSOS [Feiertag79], and Secure Unix [Popek80] and a secure implementation of INGRES [Downs80].

The availability of large quantities of information on individuals in (statistical) databases increased concern for the threat to privacy. Unfortunately, it is relatively easy to disclose information about individuals from a statistical database using limited outside knowledge. Placing restrictions on the application of the statistical operators does not solve the problem, for a general scheme exists to breach privacy [Denning80, Jonge83]. The schemes ensuring some privacy are database partitioning and sampling [Denning82]. However, these methods can be used in very large databases only, because partitioning and sampling may lead to loss of statistical properties subject to investigation.

The focus of protection research in the eighties is shifting towards distributed systems. This makes access control even more involved, since a distributed system can not rely so much on hardware support as a centralized system. To alleviate these problems, either probabilistic protection [Tanenbaum81] or a combination of insecure sites with restrictions on inter-site transfers [Rushby83] is assumed. Moreover, the secure transfer of messages in distributed systems requires new solutions to the authentication problem [Lampson81] and encryption of messages [Jonge85]

The practice of access control and authentication is getting more substance now the information society is within reach. For example, lawyers search ways to transfer and integrate computer-based methods of access control with legislation, so as to protect the investments made in software development. International banks automate their transfer of monetary values, which require, due to the potential losses, extreme sensitivity of the computer security problems.

2.2. A framework for comparison

The design of a new access control model for an integrated secure programming environment is necessarily rooted in the past. Therefore, we first define a framework for these protection models and review the significant approaches taken. This framework for comparison is general and does not fully represent all existing models perfectly. Yet it focusses on the problems at hand and it aids in the comparison of the approaches discussed.
A protection model minimally consists of four components:

1) A model of the real world.
2) A finite set of operators.
3) A finite set of security constraints.
4) An abstraction mechanism.

The most important component of a protection model is its notion of the real world, such as what real world objects it considers and what relations exist between these objects. Although one can imagine protection systems, i.e. implementations of protection models, in which users do not play any role, nor interact with the system, such systems primarily regulate human actions. Therefore, each protection model recognizes *active* entities, as opposed to *passive* entities. Active entities can initiate actions, while passive entities are the object of manipulation only. This distinction is often used to associate active entities with the users of the system and to associate passive entities with files.

For regulatory purposes active entities are normally represented in a system just like passive entities. However, they differ by the way they are used. For example, when the passive entity represents a user, its representation contains information about the identity of the person. This information is used at log-on to distinguish users and to authorize individual requests. Authentication mechanisms are not dealt with in this thesis; it is assumed that proper mechanisms exist, such as a password scheme [Wood77, Morris79, Needham78] and physical identifiers [Martin73].
The access relations among the passive and active entities can be subdivided into five classes [Denning82] ;

- Data-independent access relations, where access authorization takes place without consideration of the objects' semantics.
- Data-dependent access relations, where the semantics of the object is used for authorization decisions.
- Data-flow relations, where data-flow access relations regulate the flow of information and the flow of access permissions.

- Time/history relations, where time relations regulate access on the basis of past events.
- Context relations, where access is regulated on the basis of the combined use of requests.

The objects and relations of a protection system can be considered as defining a state of an abstract machine. States can be either used solely for deriving authorization permissions, i.e. the protection state is static, or can be subject to change, i.e. they are dynamic. In both cases we need operators or state transformation rules. For a static state the set of operators return derived information only. The state itself is never changed after it has been set up. In case of dynamic states the set of operators can be considered the instruction set of the implied abstract machine. For simplicity we assume that each operator is atomic, i.e. non-interruptible by any other request, and transforms the protection state in some meaningful way.

A protection system is of limited use if the protection policy it implements is not spelled out precisely by a set of security constraints. One way to formalize these constraints, is to introduce predicates describing secure protection states and operator invariants. The former describe the properties of users, objects and their relationships administered by the protection system. Operator invariants describe the security policy enforced for state modifications, i.e. the integrity of the protection state.

An instruction set alone is not sufficient to support the construction of secure systems. There should be a means to integrate these protection instructions with normal data processing instructions. Moreover, there should be means to encapsulate such sequences into parameterized commands or programs. The advantage of such an abstraction mechanism is that it enables the construction of multi-level machines supporting alternative authorization policies using the protection properties inherited from the underlying model.

In the subsequent sections some of the more influential models and system approaches concerned with access control are reviewed. The models discussed are classified by their prime orientation into:

- System models, which emphasize the architectural structure of the protection system and its integration with the processing environment
- Data-independent models, which abstract from implementation aspects of the protection system.
- Data-dependent models, which emphasize the intertwining of access control with the object semantics.
- Data-flow models, which emphasize the flow of access rights and data.

2.3. System models

2.3.1. Multi-level security systems

The ADEPT-50 time-sharing systems developed at the System Development Corporation was one of the first systems to implement multi-level security by software controls [Weissman73]. In a multi-level security system the information is classified following the organizational structure of a military environment. That is, each object is assigned to a hierarchically ordered security level, i.e. Unclassified, Confidential, Secret and Top Secret, and is assigned to a category, like Nuclear and Windmills. Moreover, a list of users is attached to an object, called the franchise, together with the mode of access being permitted, like Read Data and Append Data. The franchise sets are used to implement discretionary need-to-know controls.

In the ADEPT-50 system four objects types are recognized: users, jobs, terminals, and files. Upon system log-on the job serving the user's terminal is given the minimum of the security levels assigned to the user and the terminal, the category is set to the intersection of the categories accessible to the user and terminal respectively. The security policy implemented is called a high-water-mark approach, because each object (file) created is assigned the security level and categories of the existing environment and session history. Moreover, actions on objects are permitted when the user's, job, and terminal security exceed the properties of the object. The system provides hooks for the enforcement of extended protection policies.

Although the system has been implemented and used at various sites, full scale use was impossible due to the authorized "downward" flow, i.e. a user can copy classified information into a (preexisting) file that is unclassified. Moreover, automatic classification of new files suffered from overclassification of data, since the history mechanism did not allow the high-water-mark to decrease during a single session.

2.3.2. The formulary model

The formulary model for protection has been introduced by Lance Hoffman in his Ph.D. thesis in 1970 [Hoffman70]. The model provides a general method for describing access control routines and enforcement procedures. The basic observation Hoffman made is that the simplicity and rather static properties of the protection mechanisms for file systems and the passwords schemes in use permit only a small number of specific access types to be associated with the data files. He notes that the assumption underlying the file protection schemes using passwords is false. The underlying assumption is that all components of a file have the same security properties. The real world does not conform to this assumption. Problems arise when parts of a file should be made available to users from different authorization groups. For example, password protection of

files is acceptable in limited cases only. There is a need for a more dynamic approach to access control.

The formulary model proposed provides both the freedom of choosing the access control policy and claims to be efficient by checking all accesses at run-time. The scheme basically works as follows. The user at the terminal talks to the machine in some language TALK. The language TALK can be anything from programming language to interactive or menu-driven conversation. The primary task of the TALK interpreter is to perform syntax checking and to relate the resource names provided by the user to internal names. The access control takes place in an access-control routine ACCESS, which uses a 'formulary' to authorize the access request. The formulary consists of a set of four procedures:

Control procedure
Virtual procedure
Scramble procedure
Unscramble procedure

Parameters to these procedures are the internal name of the datum requested along with the internal name of the user responsible for the request. The virtual procedure transforms the internal name of the datum into a virtual address. The control procedure contains the algorithm to decide for access and performs the requested action, which is either a fetch or store of the datum. The scramble and unscramble procedures are used for data encryption purposes. Upon entering the system a user is paired with a single formulary, of which the procedure entry points are copied to the user control block.

The most interesting feature of the formulary model is the simplicity of the approach coupled with the flexibility provided for access control. Both data-dependent and data-independent protection can be supported using the control procedure. The major shortcoming of the model is the lack of a formal model to describe the access decisions and to analyze the interrelationships between collections of formularies. Moreover, the notion of granting rights to others is not provided at a general level.

Relating the formulary model to our framework, the model recognizes users and words in a virtual memory as components of the real world. The relationships between the users and objects are described in the form of the formularies, which can be thought of as the primitive operators in the model states. The single general security constraint is that all access is funneled through the procedure ACCESS. No provisions are made for composition of formularies into parameterized commands.

2.3.3. The functional model

A generalization of the formulary model is given by Conway, Maxwell, and Morgan in [Conway72a] and called a functional model. They examine the potential of an access control matrix approach to support integral protection in programming languages, operating systems, and database systems using controlled read and write functions as operators. They claim that a cost-effective implementation scheme exists based on the combined use of a trusted compiler and secure run-time system.

The functional model's notion of the real world is untyped objects and users. No formalized relationships are maintained between objects. Instead, four basic security functions to reflect elements of the access matrix are presented to guarantee data security in a wide class of situations; F_t and S_t, the translation time fetch and store functions, and F_r and S_r, the run-time fetch and store functions. Each of these functions has two arguments: a user identification and a data item. When a fetch reference is made to a data item, i.e. in the right-hand side of an assignment statement or its appearance in an output statement, $F_t(u,d)$ interrogates the specified element of the matrix and either generate conventional code when the user is permitted data-independent access, aborts the compilation when the user is denied data-independent access, or generates a call to $F_r(u,d)$ to check access permission at run-time. Similarly, S_t and S_r are used for storage requests.

Guaranteeing security constraints as embedded in the four functions depends on a secure integration of the compiler and object libraries. It should be impossible to alter the object code generated by the compiler. Abstraction primitives are not considered in this model, but as the approach easily blends with high-level programming languages, multi-level security policies are attainable. For example, the access routines can be included in an abstract data type declaration. The cost effectiveness of this model is illustrated by comparison with existing approaches, like Hoffman's formularies, MULTICS [Graham68], and their own file management system ASAP [Conway72b].

2.3.4. Capability systems

Over the last several decades, the computer industry and universities have been searching for alternative architectures better to support selective information sharing and the construction of reliable and complex systems. An important class of said machine architectures use *capability-based* addressing [Fabry74]. Capabilities are protected addresses, which can be freely copied, passed as parameters, and transmitted from domain to domain, but cannot be forged or modified by user programs. Capabilities are context-independent and address the same object regardless of where the capability is used. One can think of capabilities as tickets containing an object identifier and access permissions to that object. A detailed ontology of the capability systems has

been written by Levy [Levy84], their history is illustrated in Figure 2.1.

System	Developer	Year	Attributes
Rice University Computer	Rice University	1959	segmented memory with "codeword" addressing
Burroughs B5000	Burroughs Corp.	1961	stack machine with descriptor addressing
Basic Language Machine	International Computers Ltd., U.K.	1964	high-level machine with codeword addressing
Dennis and Van Horn Supervisor	MIT	1966	conceptual design for capability supervisor
PDP-1 Time-sharing System	MIT	1967	capability supervisor
Multicomputer/ Magic Number Machine	University of Chicago Institute for Computer Research	1967	first capability hardware system design
CAL-TSS	U.C. Berkeley Computer Center	1968	capability operating system for CDC 6400
System 250	Plessey Corp., U.K.	1969	first industrial capability hardware and software system
CAP Computer	University of Cambridge, U.K.	1970	capability hardware with microcode support
Hydra	Carnegie-Mellon University	1971	object-based multi-processor O.S.
STAROS	Carnegie-Mellon University	1975	object-based multi-processor O.S.
System/38	IBM, Rochester, MN.	1978	first major commercial capability system, tagged capabilities
iAPX 432	Intel, Aloha, OR.	1981	highly-integrated object-based micro-processor system

Figure 2.1 [Levy 84] Major Descriptor and Capability Systems

Capability-based systems take a simplistic view of the real world. The entities considered are objects with read/write/exec permissions, users in the sense of the previous models are not recognized. Instead, processes with capability lists or assigned a protection domain are the equivalents of users.

A consequence is that capability systems use a simple authorization rule; a process can gain access to an object if it can supply a matching capability for it. Other primitive operations on capabilities are transport and copying. The latter may be controlled by an access right, the copy flag, which is part of the capability representation. Moreover, the access permissions in the derived capability form a subset of the source. The abstraction mechanism for capability systems lies in the environment of its use where abstract data types can be used to control the use of object classes.

2.3.5. Remarks on system models

The prime issue addressed by the system models is read/write protection. As such it safeguards the information stores against malicious use. However, the approaches fall short in providing operations to control the use of the access control information, i.e. selective granting and revocation of access rights. Most of the data-dependent access control decisions in the formulary and functional

approach are encoded in programs and thus allow a wide class of security policies to be implemented, but they rely on a proper programming environment. Moreover, enforcement of the policy should be proved secure using program-verification techniques. Capability-based protection differs from the other approaches by its lack of a policy; instead it is a technique to implement and subsequently enforce a wide range of protection policies, guaranteeing that only accesses permitted by the policy take place [Jones75]. In Chapter 6 we show how the philosophy of formularies or access functions can be beneficially used to supplement the access control schemes to arrive at a secure implementation of information systems.

2.4. Data-independent models

2.4.1. The Harrison-Ruzzo-Ullman model

In recent years models have been developed to formalize access control in a computer system and to analyze and predict their behavior [Harrison76, Jones76, Minsky78b]. Of these models the Harrison-Ruzzo-Ullman model (HRU) is of prime interest because of its simplicity and far reaching theoretical results. The model of the real world is an access matrix P containing elements from a set of generic rights R. The matrix is manipulated by the following primitive operations:

enter r **into** (X_s, X_o)
delete r **from** (X_s, X_o)
create subject X_s
destroy subject X_s
create object X_o
destroy object X_o

An abstraction mechanism forms an integral part of the protection model. All operations are assumed to be encapsulated in a finite set commands of the form

command $\alpha(X_{s1},..,X_{sm},X_{o1},..,X_{on})$
if r_1 **in** (X_{s1},X_{o1}) **and**
  r_2 **in** (X_{s2},X_{o2}) **and**
  ...
  r_m **in** (X_{sm},X_{om})
begin
  op_1
  ...
  op_k
end

The condition is optional and describes the required properties of the access matrix before the sequence of actions is applied. The elements X_i are the formal parameters of the command α. Each op_i is one of the primitive operations. Subjects are denoted by s_i, objects by o_i. A *configuration* of a protection system is defined as the triple (S,O,P), where S is the set of current subjects, O is the set of current objects (which include S), and P is an access matrix. Every subject is associated with a row in P and every object is associated with a column in P. An element of the matrix, P[s,o], contains a subset of the generic rights, like *read*, *write*, or *execute*. An sample of such a matrix is shown in Figure 2.2.

	S_{s1}	...	S_{sm}	O_{o1}	...	O_{sn}	...
S_{s1}				read			
...						read write	
S_{sm}				exec			

Figure 2.2 An access matrix

For this model the general safety question "Can a command pass a right to an element of the access matrix where it did not occur previously to the call?" is considered. They assume that the authorized or "trustworthy" subjects are removed from the access matrix, because in that situation passing a right can be considered as a safety breach. Unfortunately, the following theorem can be proved [Harrison76].

Theorem: It is undecidable whether a given configuration of a given protection system (access matrix, collection of commands and rights) is safe for a given generic right.

This result should not discourage the design of access-matrix based protection systems, because the result obtained does not rule out particular systems where the safety question is decidable at reasonable cost. For example, the Take-Grant model, to be discussed shortly, has a simple decision procedure. Moreover, within the HRU framework, placing restrictions on the form of the commands makes the safety problem tractable. For example, if the protection system is *mono-operational*, that is each command's interpretation is a single primitive operation, then the safety question can be answered for a generic right r and an initial state. Removal of the deletion and destroy primitives turns the protection system into a *monotonic* system, but for monotonic systems with at most two operations per command the safety question is also undecidable.
Alternatively one might conceive that placing constraints on the commands' *pre*-condition makes the safety question decidable. Again this is true when the protection system is both *mono-conditional*, that is, each commands' *pre*-condition

has at most one term, and *monotonic*, that is, deletion and destroy primitives are forbidden. It is unknown whether the safety problem is solvable for *mono-conditional* systems in general.

The HRU model is generally taken as a reference model. It has not been used as an implementation guideline. An extension of the HRU model with state transition flow constraints is discussed in [Kreissig80].

2.4.2. The Take-Grant model

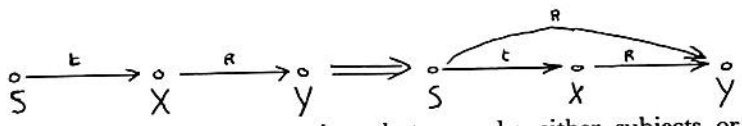
One of the formal protection models for which a safety question can be answered is the Take-Grant model [Jones76] , which, like HRU, is used to model capability-based protection systems. The central concept in this model is the description of the protection state as a labeled directed graph. Nodes come in two types, denoting subjects (active entities) and objects, respectively. The access type is used to label arcs, like read, write, execute. Two right labels play a special role: *take* (abbreviated *t*) and *grant* (abbreviated *g*). If a subject S has the right *t* for an object X, then it can take any of X's rights; if it has the right *g* for X, then it can share any of its rights with X. As such the Take-Grant model describes the transfer of authority and can be used to answer questions of the form "Can a given user obtain α access to a particular object."

The dynamics of the protection model, its primitive operations, are cast into four graph-rewriting rules.

Take Let S be a subject such that $t \in (S,X)$ (t belongs to the labels associated with the arc (S,X)) and $r \in (X,Y)$ for some right r and nodes X and Y. Then the command

S **take** *r* for Y from X

adds r to (S,Y). Graphically,

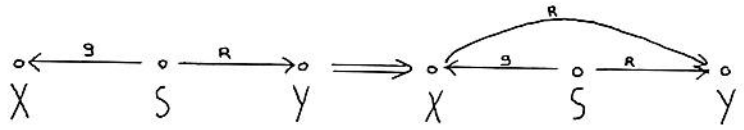


where the symbol o denotes vertices that may be either subjects or objects.

Grant Let S be a subject such that $g \in (S,X)$ and $r \in (S,Y)$ for some right r and nodes X and Y. Then the command

S **grant** r for Y to X

adds r to (X,Y). Graphically,



Create Let S be a subject and R a set of rights. The command

S **create** R for **new** {**subject**|**object**} X

adds a new node X and the set R. Graphically.

RemoveLet S be a subject and X a node. The command

S **remove** R **from** X

deletes R from (S,X). Graphically.

The safety problem is formulated as follows. Let G be a graph and S, X, and P nodes of G such that $r \in (S,X)$ and $r \notin (P,X)$ then G is called safe for the right r for X if and only if $r \notin (P,X)$ in every graph derivable from G. This property can be analyzed both from the standpoint of cooperative users, denoted by the predicate *can.share*, and by considering stealing the right by users, denoted by *can.steal*. For both cases it has been shown that the existence of an undirected path in the graph with each edge labeled with either *take* or *grant* suffices to decide safety in linear time in the size of the protection graph [Snyder81]. A variation of *can.steal* is counting the number of conspirators needed to acquire the right. Upper bounds on this number can be derived with algorithms of the same time complexity as the safety question [Budd77].

The Take-Grant model does not provide abstraction primitives for the construction of multi-level protection systems. This led Jones in [Jones78] to extend the model with a procedure mechanism. In her extension, 'property sets' are associated with both subjects and procedures and serve as templates for subject creation. That is, the execution of a procedure results in a subject with

rights defined by the template.

The Take-Grant model has been the focus of much research. The relationship between this model and HRU is described in [Snyder81], the complementary relationship is addressed in [Weyuker78]. She shows how to simulate the Take-Grant model in terms of the HRU model and vise versa. Consequently the HRU safety question turns out to be undecidable for the Take-Grant model. The role of the *take* and *grant* rights as opposed to *read* and *write* rights has been addressed by Bishop and Snyder [Bishop79] , who call them *de jure* and *de facto* rights, respectively. A hierarchical version of the model to simulate multi-level discretionary protection is described in [Bishop81]. The global flow of privileges is addressed in [Lockman81]. A commentary on the model and its applicability can be found in [Hilhorst83].

2.4.3. The grammatical models

The formal access control models discussed so far raise the following two general questions.

The safety question

Given a protection system G and two objects X and Y in that system, if we introduce the right of X to *alpha* Y, what other objects can thereby obtain the rights to *alpha* Y?

The extended safety question

Given a protection system G and two objects X and Y in that system, if we introduce the right of X to *alpha* Y, what potential changes will this produce in the entire system?

The HRU model indicates that for arbitrary protection systems these problems are undecidable. However, restricted classes, like the mono-operational systems, can be analyzed in polynomial time. For the Take-Grant model a linear time algorithm exists to answer both questions. In [Lipton78] it is shown that access control can be mapped into a language parsing problem. Consequently, the extended safety question translates to the problem of finding derivable sentences from a given grammar. For example, consider a protection system with transition rules of the following form

$$T_i \xrightarrow{\beta} T_j \xrightarrow{\gamma} T_k \Rightarrow T_i \overset{\alpha}{\frown} \xrightarrow{\beta} T_j \xrightarrow{\gamma} T_k$$

where T_i, T_j and T_k denote types drawn from the set T and α, β and γ indicate the rights from the set R. A grammar can be derived for such rewriting rules by

defining a production for each rule as follows. If they do not already exist, introduce three nonterminals, A,B, and C $\in$ T$\times$R$\times$T such that A corresponds to an arc labeled α between vertices T_i and T_k. Similarly, B and C are defined, which gives the production A $\rightarrow$ BC. A terminal counterpart 'a' is defined for each nonterminal A and the production A$\rightarrow$a (B$\rightarrow$b, C$\rightarrow$c) is added.

A protection system is called *grammatical* if for each right α $\in$R there is a grammar L and a start symbol S such that given two vertices X and Y, C can α Y iff X and Y are connected by a path such that the concatenation of the right symbols on that path form a word in L(S). For grammatical protection systems the following observations are proved.

Theorem Given two vertices P and Q of type T_p and T_q, respectively, P can α Q *iff* there exists a path between P and Q in L((T_t,α,T_q)).

Theorem The extended safety question can be answered for a general arc moving protection system in O($|V|^{2.81}$) where V is the set of vertices.

If it happens that the grammar derived for a protection system is regular, it is called a regular grammatical system, the safety question for which can be answered in linear time in the size of the protection graph. Another class of protection systems which seem to arise quite frequently are the *non-discriminating* grammars where all rules are of the form "If X and Y are connected by an arc with some right γ, and Y has any right to Z, then X can obtain that right to Z". The name implies that no distinction is made between the rights. Non-discriminating grammatical systems are proved to be regular too.

Not all protection systems can be classified as grammatical. For example the MULTICS protection and the Bell-LaPadula models are not, because they do not constrain the flow of privileges and are therefore considered "loose" protection systems. If for each right α there is some regular expression E_α such that a necessary condition for a vertex X to α a vertex Y is that they be connected by a path with word in E_α then the system is called *near-grammatical*. When the global conditions imposed by the protection system can be checked independently of the vertices involved in a transfer and they can be verified in constant time, then the safety question for near-grammatical systems can be answered in linear time in the size of the protection graph as well.

Variations on the Take-Grant model using the grammatical approach for safety analysis are discussed in [Budd80]. He shows that the safety questions for most of the extensions proposed in the literature can be answered using this framework. For example, limiting the power of Take-Grant by combining it with the right being transferred leads to a polynomial safety question. A similar result holds when multiple capabilities are required for transfer [Graham72]. The safety question for systems with unbounded create and non-monotonic rules, e.g. when rights are added and removed within a single rewrite operation, remain undecidable.

2.5. Data-dependent models

The data-independent models are characterized by an apparent lack of concern about practical use and efficient implementation. However, they are of interest in the analysis of the protection to be gained within a given context and highlight the semantics of access rights and the effects of the authorization policy chosen.

The class of data-dependent authorization models distinguishes itself from the previous approaches in addressing the problems from a database point of view. Although much of the work in this area is influenced by the access matrix in operating systems, database protection emphasizes different aspects. For example, there are often more objects in a database than in an operating system. Moreover, database security is concerned with different levels of granularity (like file, record, and field), the semantics of the data, as well as its physical representation.

2.5.1. The Hartson model

The multi-level security in ADEPT50 and the formularies of Hoffman are integrated and extended to obtain the five-dimensional security space of Hartson [Hartson75, Hartson76] The security space is defined as $A \times U \times E \times R \times S$, where A is the set of authorizers, U is the set of individual users, E is the set of operations, R the set of resources,and S the set of states. Each database access is considered a series of requests by an individual user u for an operation e on resources $r \subset R$ at a time when the system is in state s. Thus, each access request can be represented by a 4-tuple $q=(u,e,R,s)$ where both u and s are supplied by the system in an "non-forgeable" way.

The authorization process is split in two phases. The first phase is conventionally executed at system logon and establishes the users' access profile. In particular, the franchise for the user, operations, and resources are looked up, the intersection of which determines the access permissions. The second phase is executed upon each access request, where it is matched against the access profile to permit or deny access. Access requests denote a subspace of a four-dimensional projection of the security space and a request is authorized when it is contained within this projection.

The model does not provide an abstraction mechanism, but contains hooks to execute pre-decision, post-decision, and history procedures for additional protection enforcement and threat monitoring.

2.5.2. CODASYL based models

One of the most influential works on database standardization is done by the CODASYL Data Base Task Group (DBTG) [CODASYL78]. The CODASYL data model supports both the logical description of the database (the *schema*) and

derived descriptions (*subschemas*) to support user views. Each user can interact with the database through a subschema only and the schema forms the nucleus for information integration, administrative control, and the optimization of the entire database. This schema/subschema architecture has important data security ramifications since it can be used to conceal from the user information in the database which he is not allowed to access.

The model of the real world considered in the DBTG reports primarily deals with data, its relationships, and primitive data manipulations. The concept of users is not explicitly defined in the data model. Users are implicitly defined by the organizational structure and the knowledge of passwords for selective (sub)schema access. For details on the model we refer to the books on database models [Date77, Ullman80].

The languages specified in the DBTG reports [CODASYL78] contain facilities for data security such as the ability to define password protection and to invoke procedures defined by the database administrator (DBA) for security checking. The protection model is rather static, as it does not provide primitives for the dynamic dissemination and change of access information. The two abstraction mechanisms are the subschema and the database procedure. The subschema provides a mechanism for selective access to parts of the database. The database procedure is a piece of software triggered by the use of database manipulation primitives and specified in the ACCESS CONTROL clauses for data items, records, sets, areas, and (sub)schemas.

In [Manola75] it is pointed out that the required control on the internal use of data in applications places trust on the proper working of the language compiler and run-time environment. It is shown that little security is possible if an untyped, untrustworthy programming language is used for their construction. They suggest extension of the CODASYL primitives for data security along the following lines. First, the DBA should be able to administer, by declaration, more precisely the use of various operators defined for the subschema. In particular, there is a need for additional syntax to allow more precise declarations of security constraints for a subschema in terms of legal operations on the data defined in the schema. For example, the ability to declare the result of a compiler as trustworthy for certain constraints, that the DBMS is to do the checking, or that the DBA supplies a routine which takes care of the access constraints. Second, the ability to define more complex schema/subschema transformations is required, so that data objects and their operations tailored to the user's application may be defined.

2.5.3. The relational data model

2.5.3.1. Query modification for authorization

The relation between query processing in a relational database management system and access control is indicated by Stonebraker [Stonebraker74] and is applied in the relational system INGRES [INGRES]. The real world administered consists of the relations and attributes in the database. System data such as time of day, date, terminal, and user information are inherited from the operating system environment. Access control protection is provided through the view mechanism and two-level grouping. The users of a database are divided into the database owner and the rest. All relations defined by the database owner can be accessed by the other users, provided they have been given access permission. These access permissions are coupled with the primitive database operations, such as RETRIEVE, APPEND, etc. Tables defined by users other than the database administrator are considered local tables and cannot under any circumstances be accessed by other users, except the database administrator. To provide access control on table partitions, based on their content, table views can be defined to which access is given on a selective base as well. Data security is guaranteed by merging all queries with the access constraints for a user before database processing starts. The view mechanism can be considered the abstraction mechanism provided within this system.

2.5.3.2. The Griffiths-Wade model

A more conventional translation of the access matrix into database access control is found in the relational database management system developed in San Jose [Astrahan76]. The protection model defined by Griffiths-Wade [Griffiths76] controls access of users to objects stored in the database only. User identities are obtained from the environment, i.e. the operating system. The protected data objects in System R consist of relations, divided into base relations, a physical table stored in memory, and views, which are a logical subset, summary, or join of other relations. The access rights associated with relations are *read, insert, delete, update* and *drop*. The protection system primitives allow the creation of new entries and the transfer of right subsets when such transfer is permitted by the presence of the right to copy. The creator of a table receives all rights, including the right to grant these rights to others.
A system relation AUTHSYS is used to administer the permissions of users with respect to the relations and whether these rights can be granted to other users. A time stamp is included to infer the dependencies among the grants during revocation. That is, the authorization relation is an encoding of a directed graph in which the nodes represents users and the arcs are labeled by object being granted access, mode and time the grant was issued. This graph is used to drive the revocation algorithm, which removes all grants solely depending on the grant being revoked. That is, revocation may result in a domino effect, a series of grants are undone. Fagin improves the algorithm and proves its correctness

in [Fagin77]

This protection model does not provide abstraction facilities beyond the view mechanism, which makes the construction of alternative access policies and access rights infeasible.

2.5.3.3. The Wood-Fernandez-Summers model

In [Wood79] it is pointed out that the concept of ownership for access control is not always viable in a database environment. With many shared databases it is difficult to identify the owner uniquely and once identified the ownership may be valid for a short period only. Therefore, Wood proposes to place access control in the hands of one or more database administrators, such that transparency of administrative control to the users of the database is provided. That is, unlike the previous schemes, a change in administrative control does not result in a cascade of revocations.

The model recognizes users and database object types, also called data classes. A subclass is defined as a subset of the occurrences in a data class satisfying a given predicate. Classes (and subclasses) are the units of delegation. Authorization is described in this model by the 5-tuple (s,O,t,p,f), which states that subject s has access right t to those occurrences of data class O for which the predicate p is true. A user can grant this right (or its derivations) to others when the copy parameter f is true only. The access rights are separated into two groups, one denoting the normal database manipulation operations, and the second controlling the dissemination of the former to users. However, the difference is visible from the value of the copy parameter only. Revocation in this model is simplified by the rule that a class is delegated once and by propagating all grants based on the revoked action to the revoker.

2.5.3.4. The Bussolati-Martella model

In [Bussolati80, Bussolati81a, Bussolati81b] protection is modeled along the lines introduced by Wood-Summers-Fernandez in combination with the three level ANSI-SPARC database architecture. In particular, security is described at three levels: the conceptual security schema, the external security schema and the internal security schema. The prime advantage of this separation is independence of the users' security requirements from constraints on system resource security requirements.

The real-world model of this approach consists of binary relations and applications. The set of operators consist of read, delete, update, insert, which are used in three different modes: as access right, as administration (delegation) right, and as property right. The access rights permit a user to perform the corresponding action on the object associated with the right. Administration rights permit granting and revocation of access rights. Property rights denote ownership. Moreover, each right can be extended by a predicate to control

data-dependent access.
Security in this model has two flavors. First, it is guaranteed that each user access satisfies the external security schema. Second, definition of each external schema is checked against the conceptual schema for inconsistencies.

The abstraction capabilities within the model are restricted by the constraints placed in the conceptual security model; subset constraints can be used to model more restrictive policies. The link provided with an application programming language provides the options to implement multi-level protection policies. The application of this model to a distributed database environment is presented in [Bussolati80]. A variation on this theme in a specific application environment is given in [Ardity78].

2.5.4. Remarks on data-dependent models

The data-dependent access models discussed all aim at providing data security in a data base environment. The starting point is protection practice in operating systems, i.e. the access matrix, which is merged with the data model supported by the system. However, it is currently understood that most data models are too weak to support an application environment. The same holds for incorporating access protection. A better approach is to strive for an integration of access control with the application programming language semantics. For example, the database is described by a complex of abstract data types, using part of a formulary approach to enforce data security. Necessarily, such an approach requires integration and consistency with the access control model in the operating system and programming language.

2.6. Data-flow models

2.6.1. Multi-level security

The access control models discussed so far focuses on access regulations to objects. However, access control covers only part of the data security problem, since information flow as a result of exercising one's rights may threaten security as well [Lampson73]. One of the earliest models to describe and analyze the threat of information leakage was developed by Bell and LaPadula in the design of a secure operating system kernel [Bell74].

The entities considered in the Bell-LaPadula model are subjects and objects in a multi-level security environment. That is, each object is assigned a security level (secret, confidential,public) and a category. Each subject has a clearance, which enables him to access objects at a certain security level and category. For a system to be secure, two properties must hold

(1) the *simple security property*: no subject has read access to any object that has a classification greater than the clearance of the subject; and

(2) the **-property*: no subject has append-access to an object whose security level is not at least the current security level of the subject; no subject has read-write access to an object whose level is not equal to the current security level; and no subject has read access to an object whose security level is not at most the current security level of the subject.

Additional properties to be satisfied are the *tranquility principle* and nonaccessibility of inactive objects. The former means that a subject can not change the security level of an object. The latter means that a subject cannot read the contents of inactive objects and each newly activated object has an initial state independent of any previous activation.

The usual operators for object creation and modification are supported. The operators governing the transition from one state to another are formally defined and proved to satisfy the security constraints. Abstraction to hierarchical objects and integrity has been studied by [Goldsmith81]. A fundamental critique on aspects of the Bell-LaPadula model is given in [McLean85]

A variation of this model is developed by Feiertag et al. [Feiertag79], where each function reference and state variable is assigned a security level, so that the security level of each data item in a specification can be compared to the level of the function reference, which makes it more amenable to automated proofs of security. This version of the model has been the basis of the PSOS [Feiertag79] and KSOS [McCauley79] design efforts of a provably secure operating system.

2.6.2. The lattice model of secure information flow

The Bell-LaPadula model controls the flow of information by a series of conditions and properties to be maintained when a state transformation takes place. Denning [Denning76, Denning77] has shown that information-flow in multilevel protection systems can be treated in a more general way as follows. The information-flow model of the real world is defined by five components:

- a set of objects
- a set of processes
- a set of security classes
- a class combining operator
- a flow relation.

The class and the object categories in the previous model are combined to form security classes. A process can be thought of as the interpretation of a program. The class-combining operator $\oplus$ specifies the class of the result of any operation triggered by the process. The flow relation specifies the permissible information flow among security classes. The three components of the model (classes, $\oplus$, flow

relation) form a lattice structure and play an important role in multi-level systems. Maintaining secure information flow in the modeled system corresponds with ensuring that actual information flows between objects do not violate the specified flow relation.

Extension of the model to axiomatize the information-flow semantics of assignment, composition, modification, and procedure invocation has been developed by Andrews and Reitman [Andrews80]. An extensive treatment of the flow model is given in [Denning82]

2.6.3. Language-based protection

The general information-flow model of Denning has a counterpart for capability-based protection in high-level programming languages. In [Liskov76] an access control technique is introduced based on an augmented concept of data type. Entities in that model consist of reference variables and their type in high-level programming languages like Pascal. They require the declarative specification of access restrictions as soon as variables are declared. In essence, a variable declaration is extended with a list of access rights which represents operations defined for the type. The resulting program is considered *access-correct* if the accesses actually used do not go beyond those stated in the declarations. The mechanism is a compiler-based check of the access correctness, which, according to the authors, results in enhanced software reliability.

The flow relation, i.e. the flow of access rights, is associated with the assignment statement and with procedure invocation. An assignment is allowed when the rights available to the recipient of the object are a subset of those of the source object. Right amplification is permitted to occur at only one point: at the entry to a procedure implementing a primitive operation of the type. An abstraction mechanism is defined in [Liskov78] to handle the access right intricacies of hierarchically structured objects. The abstraction mechanism can be considered a generic procedure-declaration mechanism, the access rights and constraints on variables control the procedure instance. An extension to accommodate a larger class of access-correct programs to be written is given in [Stepoway81]

Similar to the capability-based approach to security, this model provides a protection mechanism to implement a class of protection policies. The orientation on programming language constructs and compile-time access enforcement is claimed to be sufficient; "the effect of a program execution should be clear when the program text is considered only."

The models discussed in this section primarily aim at the regulation of information flow. As such they deal with the reading and writing of information, while access control is considered a secondary issue.

2.7. Goals for the Secure Programming Environment

The taxonomy presented in the previous sections shows that security objectives and primitives are highly influenced by the perception level of the protection problems considered. In an information system we can distinguish at least four levels:

User view of protection;

Programming language view of protection;

Database management system view of protection;

Operating system view of protection;

At the top level we find the user view of the information system where the system is considered as an information source and manipulation system. As far as the implementation is concerned, messages are received from the user and scrutinized by protection filters like the ones proposed by Hartson [Hartson76]. The prime protection requirement is the ability to exercise one's access rights and to share one's rights with other users.

This orientation becomes more significant in the context of database management systems, where protection concentrates on the value-dependent access of information and the dissemination of access rights among users of the system. Specific models for access control have been proposed by [Griffiths76, Fernandez75, Stonebraker74, Manola75, Bussolati81b]. Verifiable implementation of security constraints in a DBMS is addressed by Downs [Downs77] relying on the secure Unix implementation of UCLA [Popek80].

The protection concepts at the programming language level are related to the construction of reliable, portable and maintainable software. Within this area we can distinguish two classes of protection tools, those related to programming language constructs and those to maintain large collections of application programs. The former include typing, scopes, and procedural abstraction, which provide the programmer protection against his own mistakes. The latter is centered around program modules, which hide the module implementation details from its environment and control the generation of system versions. Sample systems in this area are the module specification method of DeRemer [DeRemer76] , module definition facility of Tichy [Tichy79] and the Module Control System [Riet81]. A survey covering these aspects in detail is [Pietro-Diaz83].

In the operating system view of protection, the semantics of the objects are limited and are directly related to the objects defined by the operating system kernel, like files and sequences of bytes in main memory. The rights to perform

elementary operations (read, write or execute) on an object are represented by capabilities, also called tickets, and are best supplied by the hardware or operating system kernel as protected data objects [Fabry74].

The major shortcoming of the protection tools provided at each level is the apparent lack of integration with other levels. Protection concepts available at the programming language level are not related to the users running the programs or using object classes. Instead protection is mostly handled by the underlying operating system. For example, the notion of (closed) scopes as a protection mechanism within programming languages is not reflected in the operating system protection mechanisms. The notion of a protected memory segment in an operating system, with (read,write,execute) capabilities, as a protection domain, does not carry all the protection features of scopes in a programming language. Furthermore, the protection provided by strong typing in programming languages does not provide the necessary protection required at the user level. The typing mechanism provides static protection only. Often it does not allow alternative views to be defined on the same set of objects, the latter being a prerequisite in a database system and the design of an information system. At the other end of the spectrum, no tools are provided at the O/S level for the description of the relationships among the memory segments.

These statements do less than justice to the work being done by others in attempts to integrate the protection issues of the four layers. The problems pointed out were already described by Minsky in [Minsky76] , who not only suggests that data security requires the ability to form abstractions and to define appropriate operators on them, but stipulates that data security cannot be enforced without the ability to control the internal working of the user's program, i.e. integration of DBMS and application language is required. Capability-based protection is applied to programming languages by Jones and Liskov [Liskov78]. In their paper protection specifications are associated with the object type, the procedure parameters, and the assignment statement. Part of the protection enforcement can be done at compile time, the other part requires run-time support (parameter matching). The Hartson model provides many primitives for bridging the gap between the user level view of protection and the operating level view, but its formal basis is narrow. Most protection objects are introduced informally and no results are known, which predict the behavior of the model under a specific class of operations.

The Secure Programming Environment system developed in this thesis is an attempt to provide the basis for a unified approach to protection. In this respect it concentrates on access control problems in each view, largely ignoring the semantics of the objects involved. First, an access control model is defined, in which the notion of access to objects and the propagation of access is formally defined. Second, the model is used as a basis for defining a distributed machine architecture implementing the protection model. It is shown that the protection provided at the machine level carries over to the programming language

environment. In particular, scope and visibility protection is inherited from the underlying machine architecture. The gap between programming view and user view of protection is bridged using Abstract Data Types enhanced with user oriented protection rules, details of which are published in [Riet81, Wasserman82a, Kersten81a].

We conclude this introduction with a list of wishes for protection models being defined, so that when the task is finished it is possible to make some assessment of the validity of the design and implementation. These rules and suggestions are based on the work of Saltzer and Schroeder [Saltzer75]

- **Least privilege.** Every user and process should have the least set of access rights necessary to perform its task.
- **Dynamic system.** The protection system should provide primitives for the introduction, removal, and activation of the objects and users, as well as for the relationships between users and objects.
- **Fail-safe defaults.** Rights should be acquired by explicit permission.
- **Completeness of mediation.** Every access should be checked for authorization.
- **Economy of mechanism.** The protection system should be simple, efficient and must involve a limited amount of software for its proper functioning.
- **Support for distribution.** The protection system should support decentralized authorization and authentication.
- **Minimal semantics.** Access control protection should be described as much as possible independently of the semantics of the objects and users involved.
- **Openness.** The protection system should provide primitives for the analysis of the protection state, which should be available to all users.
- **Integration requirement.** The model should reflect the scope and control flow protection found in high-level programming languages.
- **Formality.** The model should be formalized such that it lends itself for sound mathematical reasoning.

3

A MODEL FOR A SECURE PROGRAMMING ENVIRONMENT

3.1. Introduction of the SPE model

The framework for a secure programming environment (SPE) developed in this thesis consists of four components: the SPE model, a set of security constraints, an abstract machine, and an abstraction mechanism. The SPE model defines the entity world considered for protection. The security constraints define the protection properties to be guaranteed for the state instances of the SPE model. The SPE abstract machine provides the procedural interpretation of the protection model and forms the basis for actual machine architectures. The SPE abstraction mechanism facilitates program definition and interpretation to support different protection policies.

In this chapter these topics are addressed informally, aimed at providing an overall picture of the model, its interpretation, its assumptions, and its capabilities. Detailed and more rigorous definitions are presented in the subsequent chapters. We start with a description of the SPE model.

The SPE model is designed to model access control in a programming environment, in which *active entities*, the set of users U, *passive entities*, the set of objects O, and *mediator entities*, the set of regions R, are distinguished. An *active entity* differs from a *passive entity* in the ability to initiate a state change under a procedural interpretation of the model.

For the moment, only the relationships between the entities are considered. The semantic properties of *passive* objects, like file read- and write- semantics, are ignored. The only right associated with an object relevant for the definition and

analysis of the protection model is its *visibility*: "a user u sees the object o within the region r". Visibility models the right to access an object and it is a prerequisite for a user to manipulate the object using any of the type specific functions. The extension of the objects with semantics is deferred to Chapter 6 where an architecture of a SPE machine and programming environment is given.

Similarly we assume that no information, such as name and password, is associated with *active* entities. The *active* entity represents a real-world person only, presumably in the form of a process running on an SPE machine. Binding the person with a process is performed at some point in time through an authentication mechanism, examples of which are described in the context of the SPE induced machine architecture in Section 6.1.7.

The *mediator* entities are artificial objects, introduced to represent protection domains; they associates a set of *active* entities with a set of *passive* entities. Users and objects are bound to a mediator, called a region, when they fall under the same security requirements or exhibit the same protection relations. They can be compared to the elements of an access matrix [Harrison76].

The entity types participate in five relationships and define an SPE protection state by the tuple:

$$SPE=<U,R,O,OWN,STR,DEF,IMP,EXP>$$

with

$$U,R,O \subset NAMES$$
$$OWN,STR,DEF,IMP,EXP \subset NAMES \times NAMES$$

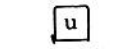
U,R, and O are finite subsets of the unbounded name space NAMES. The component U denotes the set of "active entities", R the set of "mediators" and O the set of "passive objects". The DEF relation describes the association between an object and the region in which it is defined. The component OWN describes the relation between users and regions and models the co-owner relationship.

The STR, IMP and EXP relations administer and regulate the transfer of access rights between regions. STR is an asymmetric relation between regions and defines a partial order among regions. Granting and obtaining access to objects is expressed by the relations EXP and IMP. IMP administers the fact that a particular object available within the environment of a region is made accessible within the region itself. EXP administers the complementary fact, making an object accessible within a region available to the regions in its environment.

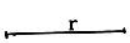
The security concepts such as owners of regions and objects, visibility of objects, granting and obtaining access rights, structure of an environment, can now be given more precisely in terms of the SPE model.

- Each user is represented within the SPE model by an element $u \in U$.
- A user u is called an *owner* of a region r iff $(u,r) \in OWN$.
- An object o is *defined* in a region r iff $(o,r) \in DEF$.
- A user u is called an *owner* of an object o iff there exists a region r such that: $(u,r) \in OWN$ and $(o,r) \in DEF$
- An object o is *exported* from region r to a region s iff $(o,r) \in EXP$ and $(r,s) \in STR$.
- An object o is *imported* from a region s into region r iff $(o,r) \in IMP$ and $(r,s) \in STR$.
- An object o is *accessible* in a region r iff either
  - o is defined in region r, or
  - o is accessible in a region s and o is imported from s into r, or
  - o is accessible in a region s and o is exported from s to r.
- An object o is called *visible* to a user u iff there exists a region r such that: $(u,r) \in OWN$ and o is accessible in r.
- The *environment* of the region r is the set of regions $\{s:(r,s) \in STR\}$
- The *contents* of the region r is the set of regions $\{s:(r,s) \in STR\}$
- A user u *trusts* v in the context of region r iff there are regions r and s such that $(u,r) \in OWN$ and $(v,s) \in OWN$ and $(s,r) \in STR$.
- User u and v are *co-owners* of region r iff both $(u,r) \in OWN$ and $(v,r) \in OWN$.
- A region r is shared if $|\{u: (u,r) \in OWN\}| > 1$.
  An object r is shared if $|\{r: o \text{ is accessible in } r\}| > 1$.

A pictorial representation is used to visualize and highlight aspects of SPE protection states. An SPE state is graphically displayed as a labeled directed graph with the following graphical conventions being used:



stands for the user u



stands for the region r

stands for the object o

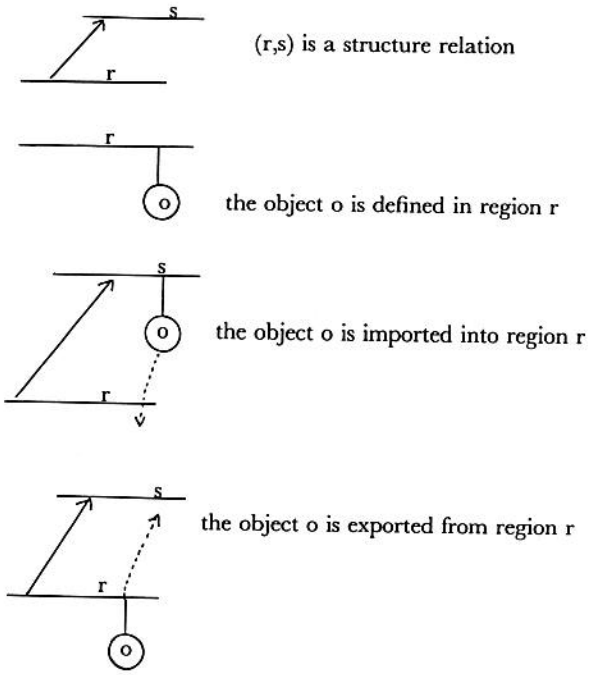


Figure 3.1 Graphical representation primitives

Graphs and algorithms on graphs play an important role in the formal analysis of SPE. Therefore, a few basic graph concepts are introduced here for convenience.

A digraph G=(V,E) consists of a finite, nonempty set of *vertices* (nodes) V and a set of *edges* E. The edges are ordered pairs (v,w), where v is called the tail and w the head of the edge. A path between vertices A and B in the graph G is a set of edges $(r_i, r_{i+1}) \in E$ (i=0..n-1) such that r_0 = A and r_n = B. The length of a path is the number of different edges making up the path. A cycle is a path between a vertex and itself. A digraph G is called a Directed Acyclic Graph (DAG) if the graph does not contain cycles. A textbook providing additional information on the concepts is [Robinson80].

3.2. The SPE abstract machine

The real world modeled by SPE is dynamic. Users and objects are created and destroyed, access permissions are granted and revoked. These dynamics are not represented by the SPE protection state. Rather it describes the relationships between the entities at a given time only. To incorporate and analyze the protection dynamics, the SPE tuples can be considered state representations of an abstract machine, similar to the approach taken by Popek and Farber [Popek78] for data-secure operating systems. Following their approach the SPE machine would be defined by:

$$SPE_{machine} = <\Sigma,\sigma_0,I,T>$$

with Σ the set of SPE states; the machine is in one state at any given time, represented by a single SPE protection state; and starts in the state σ_0. The instruction set I causes the machine to move from one state to another, as expressed by the partial function:

$$T:I\times\Sigma\rightarrow\Sigma$$

The instructions are not always total functions, because security properties will prohibit the class of insecure states transformations, to be characterized shortly. The set of instructions is further divided into two categories: the state-analysis instructions and the state-modifying instructions.

3.2.1. The state-analysis instructions

The state-analysis instructions extract information from the SPE protection state and present it to the caller, the user, or they are used as parameters to a program. Applying an analysis instruction never changes the protection state. It behaves like a no-op operation in the abstract machine. The set of state-analysis instructions used in this thesis is shown in Figure 3.2. The instructions in the first group return a set of SPE names. The instructions in the second group are protection predicates yielding a Boolean result. The latter are extensively used in the description and analysis of the SPE instruction set. They are also being used as Boolean functions in the description of SPE programs, where the protection state indicator σ is omitted and the function is written in the Roman font.

usr_reg(r)	users associated with the region r
reg_usr(u)	regions associated with the user u
reg_own(r)	some owner of the region r
obj_reg(r)	objects defined within the region r
reg_obj(o)	region where the object o is defined
reg_exp(r)	objects exported from the region r
reg_imp(r)	objects imported into region r
reg_con(r)	regions in the context of region r
reg_env(r)	regions in the environment of region r
Owner(σ,u,r)	u is the owner of the region r
Defines(σ,o,r)	o is defined in the region r
Access(σ,o,r)	o is accessible in region r
Visible(σ,o,u)	o is visible to user u

Figure 3.2 SPE state-analysis instructions.

3.2.2. The state-modifying instructions

The state-modifying instructions either extend or remove entities and relationships. Figure 3.3 shows the list of state modifiers used. They are separated into incremental and decremental instructions. Unlike the analysis instructions, each invocation of a state modifier is associated with an active entity, called its *activator*. The activator is held responsible for the call and is the focus of the security constraints to be discussed shortly. A region which participates in the change of the protection state is called an *affected* region. For example, the regions r and t in the list below are *affected* regions. The user u is an *activator*.

add_region(u,r)	u introduces a new region s and becomes its owner
add_object(u,r,o)	u creates a new object o within the region r
add_owner(u,r,v)	u introduces user v as a co-owner of r
add_struct(u,r,s)	u introduces a structure between regions r and s
add_import(u,r,o)	u imports the object o into the region r
add_export(u,r,o)	u exports the object o from the region r
del_region(u,r)	undoes add_region(u,r)
del_object(u,r,o)	undoes add_object(u,r,o)
del_owner(u,r,v)	undoes add_owner(u,r,v)
del_struct(u,r,s)	undoes add_str(u,r,s)
del_import(u,r,o)	undoes add_import(u,r,o)
del_export(u,r,o)	undoes add_export(u,r,o)

Figure 3.3 SPE state-modifying instructions.

3.3. Security axioms and properties

The SPE model as presented above allows a rich class of complex SPE instruction sets to be defined, of which the instruction set introduced so far is one As the SPE model is aimed at the description and regulation of security problems in programming environments, the definition of instruction sets is restricted by the following seven axioms. The first four axioms should be guaranteed by any protection system. The last three are specific to the SPE model.

AXIOM 1 The initial state of the machine is secure.

AXIOM 2 Access control security is a state transformation invariant property.

Given a secure initial state of the abstract machine, security violation can only occur as an effect of applying an SPE state-modifying instruction. Therefore, association of security with the actions provides a more flexible means to extend or alter the protection policy. When security is associated solely with properties of the protection state, for example by using an access matrix to represent the relation between users and objects, the authorization policy is too tightly coupled with the representation. Moreover, in general, it is easier to inspect the changes made by an action for undesirable effects, than to check the full protection state for violation of the security properties. Therefore, emphasis is placed on guaranteeing proper behavior of these instructions (Sec 4.3).

AXIOM 3 The use of state-analysis instructions is unrestricted.

This axiom embodies the worst-case assumption necessary to reveal the scope of protection provided by the protection system. This way each user of a protection system is able to analyze the impact of his access control decisions. Therefore, in the SPE model the contents of a protection state can be inspected by all users.

Note that this axiom may in practice reveal confidential information on the access rights available to individual users, who thereby may become the target of bribery or blackmail. However, the usage and control of that information is outside the scope of the protection model and its implementation.

AXIOM 4 All co-owners are considered equally powerful.

Users with the same security properties can be grouped together within a single region. Unlike in some older models and methods, objects created by any member of the group can be deleted by any member of the group. Moreover, users may introduce co-owners at any time and remove any of the co-owners as well.

AXIOM 5 Each user owns at least one region.

Each user recognized within the protection state should be associated with a protection domain and all actions on the protection state involve manipulation of a protection domain. Therefore, making provisions for users without a protection domain is considered an insecure state of affairs.

AXIOM 6 Each region has at least one owner.

Users (or processes) are the only entities to bring about a change in the protection state. Therefore, they should be related with those parts of the protection state, i.e. regions, for which they are held responsible. This way multiple users can be associated with a single region; this models the common situation where a group of users share resources under the same protection requirements.

AXIOM 7 Each object is defined in exactly one region.

This axiom implies that object sharing is established through co-ownership and access permission grants only. It avoids the situation where an SPE implementation should take into account duplicate or remote information during the protection analysis and should guarantee consistency between the copies. Together with the former axiom it ensures that at least one user is responsible for each object during its life.

The axioms for the protection model can be refined to security properties for a particular SPE instruction set. They differ from axioms by specifying the behavior, or security policy, of a particular implementation. The policy considered in this thesis for SPE implementations is expressed by the following security properties (SP):

SP-1 A state of the SPE machine should be *consistent* and *acceptable.*

Consistency is a weak constraint on the information represented by the protection state. In particular, consistency requires that each region used within the relations OWN, DEF, STR, IMP, and EXP is a member of R, each object used within DEF, IMP, and EXP is a member of O, and each user mentioned in OWN is a member of U. A protection state is considered acceptable when it obeys the axioms A5 to A7.

SP-2 A state of the SPE machine should be *valid.*

A state of SPE is considered valid if the accessibility of objects, as modeled by DEF, STR, IMP and EXP, is consistent. This means that an object is granted to (or obtained from) a region only if it is accessible in the region from which it is granted (or obtained).

SP-3 A state transformation of the SPE machine is *permissible* if the activator of the transformation is an owner of the affected regions.

Intuitively this means that the state of SPE can only change by an action of an activator, i.e. an active entity. Moreover, an activator can only change those parts of the protection state for which he is held responsible.

SP-4 The flow of access rights in the SPE machine is bounded by the existence of a path in the undirected graph induced by STR.

The undirected graph induced by the STR relation plays the role of a map. Information can flow along paths defined by this map in either direction through proper use of import and export instructions. In fact, the STR relation describes the permissible flow of access to objects within the model, while instances of IMP and EXP represent the actual flow.

The constraints SP-1 to SP-4 impose restrictions on the states and state transformations. SP-1 and SP-2 are static constraints that define the notion of a secure state, while SP-3 and SP-4 are dynamic constraints that restrict the class of state changes. The axioms and properties can now be used to define security for an SPE implementation more precisely as follows:

Definition 3.1

An SPE abstract machine is *secure* if each instruction can be proved to guarantee axioms 1 to 7 and security properties SP-1 to SP-4.

The instruction set shown in Figure 3.2 and 3.3 is one of many possible sets defined with these axioms and properties in mind. The question arises as to what in general should be considered a good instruction set, that is *well-defined.* In the context of protection systems an instruction set can be considered well-defined if it satisfies the following constraints:

- For each sequence of state-modifying instructions applied to a state, a sequence can be given which undoes all the changes made,

- An instruction can not be simulated by a sequence of other instructions taken from the same set, and
- All secure states can be generated from a given initial secure state.

Well-definedness of SPE instruction sets is discussed in Section 4.5, while the instruction set introduced above is shown to exhibit this property in Section 5.2.

3.4. SPE security analysis

3.4.1. Revocation

The sample SPE instruction set is used to study various related protection issues in Chapter 5. First, we consider the problem of *revocation*, i.e. under what conditions an action once performed can be undone. This problem is studied using sample approaches found in the literature, which show that there exist semantic limitations on undoing actions in general. For example, incremental instructions can be undone only and in that case the role of the revocation activator is a limiting factor.

A formal definition of two SPE revocation policies is given: *chronological* revocation and *goal-seeking* revocation. Chronological revocation uses the sequence of actions applied to a protection state to undo an action. The technique applied is basically a simulation scheme and of limited use in practice. However, its analysis results in a better understanding of the dependencies among the SPE instructions. The goal-seeking scheme takes as a target the successful execution of the decremental counterpart of the instruction being revoked. Unsuccessful execution of the counterpart is followed by an analysis and revocation of the obstructing instructions first. Thereafter the target action is tried again. The prime advantage over the previous method is that the algorithm does not require exclusive access to the protection state.

3.4.2. Derivable secure states

To bridge the gap between initial and desirable protection states, part of the instruction set should be available to the user and the desired state should be derivable. To find out if this gap can be bridged, the SPE predicates indicated below are introduced and analyzed in Section 5.4.

Can.connect(σ,r,s,*P*)
Can.share(σ,r,s,o,*P*)
Can.obtain(σ,u,o,*P*)
Can.steal(σ,r,s,*P*,*Csp*)

These predicates give a handle on protection properties in an early stage of the design of a protection system. The predicate *Can.connect* is true when the two

regions r and s can be connected by a structure path using the instruction set *P*. *Can.share* tells if it is possible to pass access to the object o from region r to s. *Can.obtain* extends the latter to incorporate users, can the user obtain access to the object? The last predicate, *Can.steal*, is useful for the analysis of access stealing. It requires both an instruction set and a set of potential conspirators. In particular, the analysis shows that the two most dangerous SPE instructions for stealing access rights are add_owner and add_import. These instructions, therefore, should be precluded from general use in practice. Moreover, it is shown that answering *Can.steal* does not require the construction of the class of reachable states. Simple properties of the initial state and knowledge of instructions available suffice to obtain the answer at marginal cost.

3.4.3. SPE programs

To improve the usability of the SPE model as a framework for application-specific access control, commands or SPE programs are introduced in Section 5.2. A set of commands defines a new abstract machine with extended constraints on the applicability of the SPE machine instructions. Such a derived machine is called an SPE configuration P(σ,C) and is characterized by an initial state σ of the SPE machine and a (finite) set of commands C, defined in the following form:

command alpha (X_1 , .., X_n)
pre state condition
begin
        SPE instructions,
        SPE commands, or
        general-purpose instructions
end
post state condition

The command body consists of a sequence of SPE instructions such as were defined in Figure 3.2 and Figure 3.3. The parameters of the SPE instructions are either constants or the command parameters X_i. Recursion of commands is allowed, making the construction of realistic programs more attainable. The general-purpose instructions denote instructions provided by the machine hardware, such as arithmetic and flow-of-control instructions. These instructions are ignored, because their impact on the security properties requires a precise specification of their semantics, as in [Feiertag79]. In this thesis we restrict ourselves primarily to access control and assume that general-processing instructions do not interfere with the protection model.

The *pre* (*post*) condition prescribes the validity of the SPE machine state upon command invocation (termination). That is, a command is executed when the

pre-condition holds, all statements can be executed successfully, and the final state has the properties implied by the *post*-condition. Naturally the *post*-condition is a consequence of the *pre*-condition and statement semantics when one assumes that there exists only one instruction stream. That is, commands are not executed concurrently, which we assume in this thesis. Therefore, when the result of the command can be derived from the *pre*-condition and the behavior of the command body we leave out the *post*-condition. The command structure resembles the definition given in [Harrison76] but deviates from their definition by allowing recursive SPE command calls. Furthermore, the *pre*-condition will be used to augment the authorization constraints defined for the model.

Commands are represented in the SPE protection state as SPE objects, making them subject to the SPE protection rules. However, assigning commands to SPE objects adds semantics. In particular, the SPE machine should be able to interpret such an object. The SPE instructions associated with commands are shown in Figure 3.4. The most important instruction is exec(), which takes the name of an SPE object and a parameter list and interprets the command body with the actual parameters replacing the formal parameters. In addition to the analysis instructions of Figure 3.2, the process enquiry instruction usr() and reg() can be used within a command to construct a dynamic protection system. They return information on the run-time environment of the call. The usr() operation returns the user responsible for the activation of the command, i.e. the command activator. The reg() operator returns the name of the region in which the command was activated. Note that the instructions reg() and usr() are merely a convenience for two mandatory command parameters.

exec(o,parm_1,..,parm_n)	execute a command
reg()	region where the command was activated
usr()	user who activated the command

Figure 3.4 SPE command instructions.

The SPE configuration and the SPE instruction set can be considered as defining a two-level machine. The SPE instructions form the kernel of the machine, while the collection of commands defines the user interface to the machine. This separation between the kernel and user interface does not limit the modeling abilities of SPE, because one can construct a user interface where each SPE instruction is represented by a single command. However, this separation of levels allows for the definition of SPE configurations in which the security policy to be enforced differs from the policy of the security kernel. For example, the invocation of a command can be restricted to a particular user or user group, while the command object itself is globally accessible.

The generalization of SPE security to include configurations gives:

Definition 3.2

An SPE configuration $P(\sigma,C)$ is *secure* if each command in C can be proved to guarantee the protection axioms 1 to 7 and constraints SP-1 to SP-4 given an initial secure state σ.

3.4.4. Comparison with existing models

The SPE model and its instruction set are one in a series of well-documented theoretical models on access control. Two other well-known models are the Take-Grant and the Harrison-Ruzzo-Ullman (HRU) models. The SPE command mechanism is used in Chapter 5 to simulate both models. The prime conclusion from this exercise is that both models can be simulated without loss of semantics. This means that the protection properties of an SPE implementation can provide a secure basis for both other approaches. Moreover, the simulation shows that indeed the semantics of operations such as *read* and *write* form a second level of protection problems, mappable into access control.

The reverse process has been applied as well. It is shown that the SPE model and instruction set can be simulated using the HRU model, provided that the formalism of the latter is extended to simplify the mapping. For the Take-Grant model it is not possible to simulate the SPE model, because the former model is devoid of a proper abstraction mechanism.

3.4.5. Architecture of an SPE machine

In Chapter 6 the SPE model is applied to the design of a distributed computer system, a sketch of such a machine based on loosely coupled processors is given. In that context we are not aiming at presenting a totally new approach to distributed systems architectures, but concentrate on the issue how the SPE model fits into approaches published elsewhere. One of the outcomes is that the SPE model provides a formal framework for a machine architecture centered around abstract data types. Moreover, the security kernel, necessary to implement a secure distributed system, can be associated with the SPE regions, because they are the focal point of access decisions. It is also shown that to control the flow of access rights in a distributed systems a description of the possible flows is required, which is naturally covered by the structure relationship in the model.

3.4.6. Protection in a programming environment

In Section 6.2 it is shown that visibility rules in high level programming language can be considered special cases of access control. The visibility rules in the programming language PLAIN are used to illustrate this. Application of the SPE concepts show some weak points in the language definition. In particular, the specification of exception identifiers and visibility in nested module

definitions need some adjustment.

The access protection found in high-level programming languages have a natural extension to include the environment of the language. A good example of such an approach is the development of Ada* and the User Software Engineering environment [Wasserman79a]. In Section 6.3 the relation between protection model and structure of such an environment is exemplified. In particular, we illustrate how module interconnection relates to access control. Finally, the SPE machine architecture and PLAIN language semantics are combined to indicate the architecture of a secure environment in Section 6.4.

3.5. Example use of the model

3.5.1. SPE states and sequences

In this section the functioning of the SPE protection model and the command facility are illustrated by a few examples. We start with an initial state of two users u and v, the owners of two regions r and s, respectively. Within the region s an object o is defined. Thus v is the owner of s and o. The situation is illustrated in Figure 3.5.

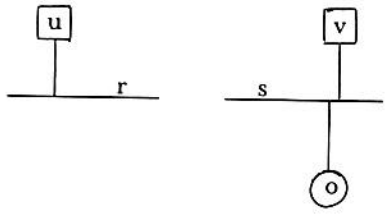


Figure 3.5 Initial situation.

This picture shows that the state satisfies both security properties SP-1 and SP-2, that is, the state is consistent, acceptable, and valid. The predicates defined in Figure 3.2 evaluate to:

Owner(σ,u,r) = true,
Defines(σ,o,s) = true,
Access(σ,o,r) = false,
Access(σ,o,s) = true;

Owner and *Defines* obtain their value from the state as shown. *Access*(σ,o,s) is true, because the object is defined in region s only. *Can.connect*(σ,r,s,*P*) is true when *P* includes the operator add_owner only. The same is true for

* Ada is a trademark of US Department of Defense.

Can.share(σ,r,s,*P*). *Can.steal*(σ,u,o,*P*,*Csp*) is true when v can act as a conspirator (v∈*Csp*) and access can be shared between regions r and s. More details on these properties are given in Section 5.4.

Consider the case where v wants to share access to the object o with user u. One solution is to add u as a co-owner of the region s with the operation:

add_owner(v,r,u)

which satisfies property SP-3. In the context of the SPE model this is a bad solution, because a side effect is that it makes u and v co-owners of the region r, which allows u to remove v as owner of s and thus remove v as the owner of the object. This effect is a result of axiom 4, which states that all users associated with a region are considered equally powerful, both with respect to the objects defined within the region and with respect to the addition and removal of co-owners of the region.

An alternative solution is to construct an STR relation between r and s as shown in Figure 3.6. This situation is interpreted in SPE as follows. The region r is defined in the context of region s. When access rights are imported, it is checked that the object is already accessible in the environment of the affected region. If this is the case then the import action succeeds and access is obtained. In this example u can gain access to the object by issuing the instruction:

add_import(u,r,o)

The drawback of this approach is that all objects defined within s can be accessed in r through an import command.

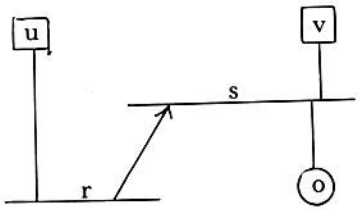


Figure 3.6 Second situation.

In case of suspicion, that is v does not want this to happen, a dummy region, like a mailbox, can be used. First user v creates a new region, say t, builds a structure relation from s to t, exports accessibility to the object, and introduces u as co-owner of t with the sequence of instructions:

add_region(v,t)
add_struct(v,s,t)
add_export(v,o,s)
add_owner(v,t,u)

The result is illustrated in Figure 3.7. In that situation u can connect to the region t (he owns it too) and use import operations to make the object o

accessible within region r using the sequence:

add_struct(u,t,r)
add_import(u,o,r)

Note that in the final situation user u can gain access to those objects exported from region s by v only. Moreover, it illustrates the use of structure relations to bind and limit flow of access rights.

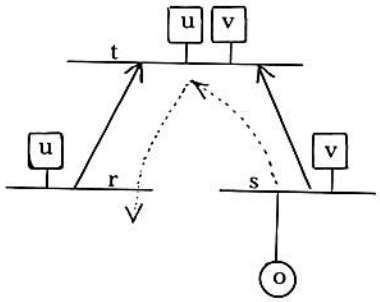


Figure 3.7 Final state

The introduction of user u as a co-owner of t is not dangerous any more, because v keeps control of the object and export operation. In particular, v can remove (s,t) from STR and, as long as v is still an owner of t, remove (r,t) from STR. Breaking the structure path implies revoking access to o within r. More details on this topic are given in Section 5.3.

3.5.2. A project management environment

The examples in the previous section demonstrate the modeling capabilities of SPE without concern about the real-world environment being modeled. A translation of SPE and its operations to a real world example is illustrated by modeling a project management organization. Therefore, assume that a system is needed in which a single system manager, called Boss, is allowed to introduce new projects and to designate its project leaders. Each newly introduced project should be allowed to use a number of system resources, such as a filing system (F) and a compiler (C). Moreover, each user should be given some privacy in manipulating objects. Access to objects may be passed freely between the members of a project team, while objects can be passed between projects with consent of the project leaders or Boss only.

Such an environment can be described in SPE using a system region, Office, containing the system resources F and C. The Agenda should be inaccessible to anyone except its owner, Boss, and therefore is stored within a separate region Boss_private. The creation of projects, programmers, and object sharing is

specified by three commands: newproject (P), newprogrammer (N), and share (S). Newproject allows Boss to define a new project at the same time designating a project leader. The project region is owned by both Boss and the project leader. The project leader (and Boss) can assign programmers to the project using newprogrammer. All system resources are imported into the program's region, thereby establishing its working environment. The protection state is constructed such that free sharing of objects and resources among the project team members within the same project is possible using the import and export facility in SPE. One command, Share, is specified to illustrate the transfer of an object to the project leader. Interproject sharing is controlled by the leaders (using Office region as a mailbox).

```
command newproject(pjname, leader: NAMES)
pre      usr() = Boss
begin
     add_region(Boss, pjname);
     add_struct( Boss, pjname, Office);
     add_owner( Boss, pjname, leader);
     add_import(Boss, pjname, C);
     add_import(Boss, pjname, F);
     add_import(Boss, pjname, N);
     add_import(Boss, pjname, S);
end

command newprogrammer(directory,friend: NAMES)
pre Owner(usr(),reg()) ∧ (reg(),Office) ∈ STR
begin
     add_region(usr(), directory);
     add_struct( usr(), directory, reg());
     add_owner( usr(), directory, friend);
     add_import(usr(), directory, C);
     add_import(usr(), directory, F);
     add_import(usr(), directory, S)
end
command share(object: NAMES)
pre Owner(usr(),reg()) ∧ Access(object,reg())
begin
          add_export(usr(),reg(),object)
end
```

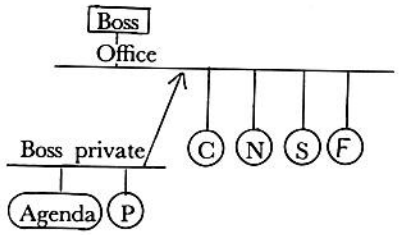


Figure 3.8 Initial situation

Figure 3.9 shows the situation after the definition of the database project team headed by Jones. Two programmers Smith and Black are assigned to this team and have been given access to the file system and compiler. As soon as an object is exported from the programmer's private region to the project region it can be accessed by the other members using add_import. Note that not all access flow is shown. Figure 3.10 shows the situation after the definition of the second project B. When transfer of objects is restricted to the existing hierarchical structure relation, project teams can only share work through cooperation of the project leaders or Boss.

In practice more commands are required to model the envisaged project organization. For example, one would make a provision to administer the last approved version and provide access to this version of the system only. In addition one might consider keeping a log of the actions for management control and accounting.

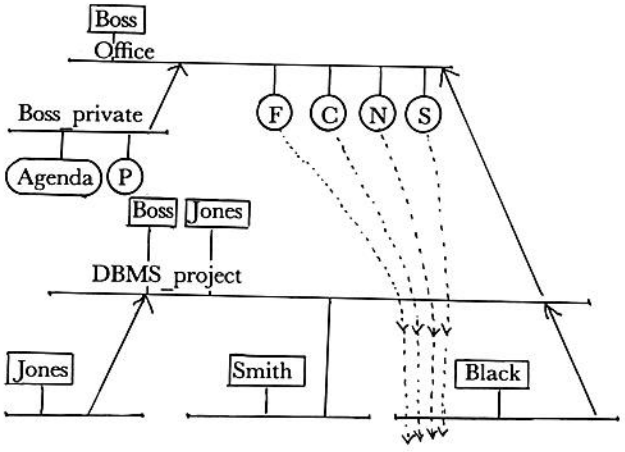


Figure 3.9 Example project organization

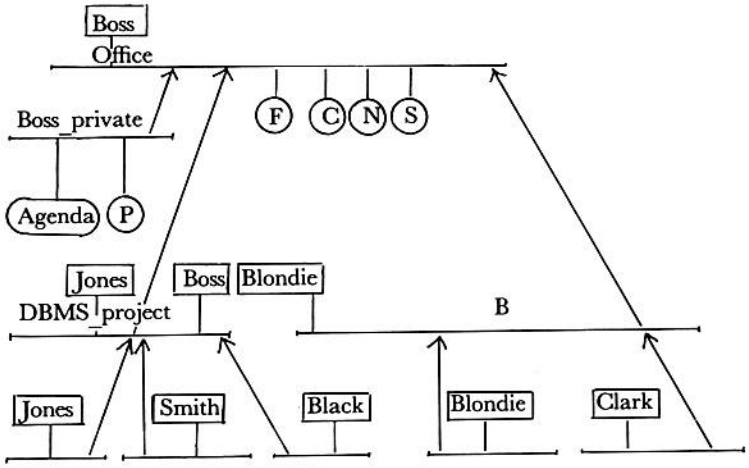


Figure 3.10 Example project organization

3.6. Variations on a theme

As mentioned before, the SPE model components and properties are influenced by the envisioned application domain: a programming environment for a high-level language which supports the construction of interactive information systems. Variations of the SPE model are possible, but are, as we will demonstrate, unattractive under this objective. For example, removing the STR relation yields a model in which all regions can be considered as boxes within a universe U. Each region contains the object definitions and has owners associated with it, as in the SPE model. The notion of export in the derived model implies making an object accessible in the universe U, while import implies obtaining an object accessible within U. The major restriction of this model is the unbounded flow of access control information.

A second model can be obtained by dropping either the IMP or EXP relationship. First, consider the removal of EXP, then each export operation can be simulated by an appropriate sequence of primitive operations. In Figure 3.11 an export of o is shown in the SPE model, in Figure 3.12 shows the simulation of this situation without using EXP. This sequence can be encapsulated in a command definition to handle any export operation. Observe the complexity of the simulation, as opposed to the situation of Figure 3.11 where v is actively involved in the transfer. Moreover, the region t' is needed to shield the existence of p. (Such a region is called a filter region)

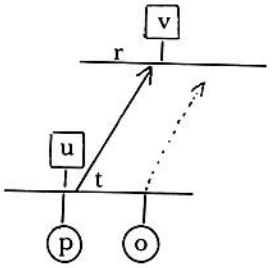


Figure 3.11 Export of o.

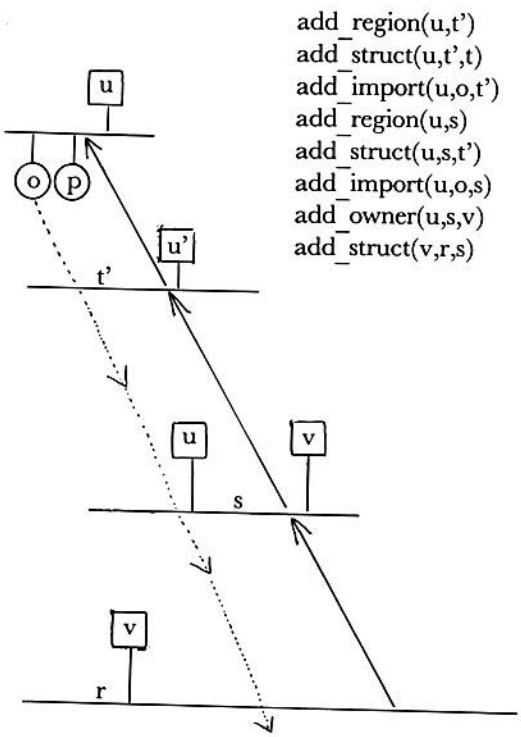


Figure 3.12 Simulation of export.

A third variation of the SPE model is obtained by merging the IMP and EXP information into one visibility relation VIS $\subset O \times R$. That is, in VIS we administer all the regions in which the corresponding object is accessible. Upon an import or export operation on an object o, the regions defining the environment, say $E \subset R$, are collected and VIS is extended with $\{o\}^*E$. The

contents of VIS is based on the actual environment at the time of the activation of the operation. Thus, changing the structure of the protection state will imply changes to VIS too, but from VIS we can not determine the import/export operations performed. This problem also applies to the revocation of the access rights. For example, consider the system in Figure 3.13 in which user u has obtained access to o via both user v and w. In the third model these two grants cannot be distinguished and revocation by either user v or w of their export operation results in the withdrawal of the visibility of o. In SPE revocation by w (or v) does not affect the visibility of o, because revocation does not affect the grant of v (or w).

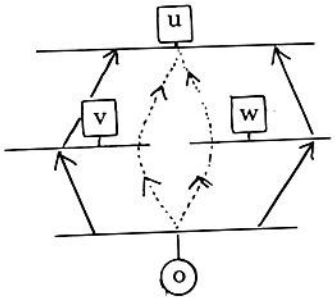


Figure 3.13 Multiple grants for o.

3.7. Summary

In this chapter we have given an overview of the SPE protection model. The motives for choosing the SPE 8-tuple and the semantics of the protection system were influenced heavily by the systems and models discussed in Chapter 2 and may be summarized by the following principles:

- *SPE is protection domain oriented.* Each access control decision is taken in a local environment, the affected region. This agrees with the approach taken in capability-based machine architectures [England72, Wulf74, Pollack82].
- *SPE supports restricted flow of access rights.* Unlike the capability-based systems, the flow of access rights is bounded by the requirement of STR relationships between the protection domains involved. This feature was borrowed from Denning's approach in restricting the flow of information within programs [Denning77].
- *SPE models scopes in programming languages.* The role played by the regions, structure, import and export relations closely resembles the structure of scopes in modern high-level programming languages. This feature was influenced by the early work of Conway in considering access protection to

be a software problem [Conway72a].

- *SPE assumes non-symmetric trust.* Non-symmetric trust better represents reality. A user must not necessarily accept the access rights given, nor should a user be forced to remain responsible for access rights once obtained and therefore should be able to revoke his actions. A similar observation has been made by Minsky for the Take-Grant model [Minsky81].
- *An SPE implementation is an extensible system.* The command construction facility allows complex security policies to be defined and enforced.
- *Security is an invariant property of a dynamic system.* This principle means that in the design of any secure system we should not stop considering the security problems after authorization of a request has taken place, but that the effects of the actions are equally important.
- *A protection system should supply an abstraction mechanism.* This principle means that in general one can not define the semantics of all objects involved in advance. Rather, one should be able to parameterize sequences and this way build multi-level protection systems. An example how this may affect the construction of interactive systems is given in [Riet83, Wasserman82a].

The combination of these principles into a new protection model sheds some light on the potentials and limitations of access control in actual computer systems, especially the intermingling of this in machine architecture, operating system, programming language, and program development environment. However, the resulting SPE model is not the final answer to all security problems, since, as will be stated repeatedly, many real-world policy decisions shape the architecture and behavior of such systems.

4

FORMALIZATION OF SPE

In this chapter the SPE model is given a formal basis. A formal description of the model provides the means to study and analyze the behavior of the model under different circumstances. Moreover, a formal and precise definition provides a sound basis for contemplated implementation of the model.

The formalisms applied in this chapter are the theory of sets, graphs, and functions, which we assume to be more comprehensible than any of the formalisms designed during the last decade in the area of programming semantics such as [Stoy77].

This chapter is divided into three parts. First, the states of the SPE abstract machine are characterized formally and the class of secure states is defined. Second, the transformations on SPE states are defined by their mapping characteristics, which results in a characterization of potential SPE machine instructions. Finally, we formalize general security requirements for SPE instruction sets. The concepts introduced are used in the next chapter to design a particular instruction set and to study its behavior.

4.1. The SPE states

Within the SPE model we consider three finite entity sets: the users called USERS, the objects called OBJECTS, and the regions called REGIONS. These sets are abstract representations of corresponding real-world entities with the elements drawn from an unbounded name space NAMES. For convenience, we assume that the three sets are mutually disjoint and not empty. Thus, the type of the entity can be derived from its name. In particular, we will use the following notational convention:

u,v,w,u',v',w',.. ∈ USERS
o,p,q,o',p',q',.. ∈ OBJECTS
r,s,t,r',s',t',.. ∈ REGIONS

The entity sets define five binary relations in the protection model:

- OWN for ownership,
- DEF for defining context,
- STR for environment structure,
- IMP for object importation and
- EXP for object exportation.

These sets model part of the relationships between entities found in reality, that is, subject to the problem area addressed within this thesis. Using the SPE model as a framework for the description of an abstract machine leads to the notion of SPE machine states, which are characterized by instances of the entity sets and the binary relations. Thus, an SPE (machine) state is a set of names and relations between names, which refer to entities and relations in the real world at a particular moment.

Definition 4.3

A state σ of the abstract SPE machine is a tuple

$$<U,R,O,OWN,STR,DEF,IMP,EXP>$$

such that:

$$U \subset USERS$$
$$R \subset REGIONS$$
$$O \subset OBJECTS$$
$$OWN \subset USERS \times REGIONS$$
$$STR \subset REGIONS \times REGIONS$$
$$DEF \subset OBJECTS \times REGIONS$$
$$IMP \subset OBJECTS \times REGIONS$$
$$EXP \subset OBJECTS \times REGIONS$$

where USERS, REGIONS, and OBJECTS are mutually disjoint subsets of the name space NAMES. Σ is the set of SPE machines states.

4.1.1. Consistent states

We are not interested in all possible SPE machine states, but rather in subclasses with properties relevant for access control. A first subclass of SPE states comes to mind immediately, namely those states which are at least internally consistent. That is, the binary relations OWN, DEF, STR, IMP and EXP are defined between elements of the entity sets U, R and O in the same state. For example, a state where an object o is defined in region r, i.e. $(o,r) \in DEF$, while either $o \notin O$ or $r \notin R$ is not considered consistent. This requirement was introduced in section 3.3 as security assertion SP-1 and is formally defined by:

Definition 4.4

A state $\sigma \in \Sigma$ is a *consistent* state, denoted by *Consistent*(σ), if the following properties hold

c_1) OWN $\subseteq U \times R$
c_2) STR $\subseteq R \times R$
c_3) DEF $\subseteq O \times R$
c_4) IMP $\subseteq O \times R$
c_5) EXP $\subseteq O \times R$

The class of consistent states is denoted by C and is a proper subset of the universe Σ. The finiteness of the state components ensures that state consistency can be checked using any of the well-known set representations and set membership algorithms. As a notational convention, a state component is subscripted with the name of the state when it is not clear from the context which state is meant, e.q. OWN_σ.

4.1.2. Acceptable states

The second subclass of the universe Σ considered within the SPE model is the class of acceptable states. The acceptability property of a state models the ownership policy. Informally, in an acceptable state each user is associated with at least one region and each region is associated with at least one user, who is considered the owner of the region. Furthermore, we restrict the class of acceptable states to those where the objects are defined within a single region only. These constraints prohibit sharing of objects through the DEF, i.e. definition, relation. Sharing of objects is provided for within the model using a different strategy, the import-export mechanism defined in the next section.

Definition 4.5

A state $\sigma \in \Sigma$ is called *acceptable*, denoted by *Accept*(σ), if it has the following properties:

a_1) $\forall\, u \in U: \exists\, r \in R: (u,r) \in OWN$
a_2) $\forall\, r \in R: \exists\, u \in U: (u,r) \in OWN$ *
a_3) $\forall\, o \in O: \exists!\, r \in R: (o,r) \in DEF$

The class of acceptable states $A = \{\sigma \in \Sigma : Accept(\sigma)\}$ is a proper subset of the universe Σ. Acceptable states can be thought of as modeling the concept of an access matrix [Harrison76] using the three entity sets and the relations OWN and DEF. In a sense, regions can be considered as representations of the non-empty elements of the access matrix. The constraints enforce that no matrix element exists without a corresponding row indicator, the user, and that for each object at most one column exists. Thus, acceptable states model the situation where multiple users are associated with the same row in an access matrix

* $\exists!$ indicates "there exists a unique."

model.
In the sequel, two predicates *Owner*(σ,u,r) and *Defines*(σ,o,r) are used, which are true in the state σ when the user u is an owner of region r and the object o is defined in the region r, respectively (See Section 3.1).

4.1.3. Valid states

In the definition of an acceptable state we formulated constraints on the relations OWN and DEF. Together, these constraints relate users with objects using the region as an intermediate. This leads us to the situation that we can define the concept of object accessibility for the SPE model.

Access control is the basic policy used in computer systems to regulate the use of objects by users. Moreover, a system is defined secure if one can guarantee that all usages of objects are authorized, i.e. covered by access control information. Translated to the concepts defined so far, an object is said to be accessible to a user if the user is owner of the region where the object is defined. The authorization question, or access control policy, in this context is translated to the assurance that object usage is limited to its owners.

Protection models aimed at the regulation of the relationship between objects and their owners only are of little interest. The model should also include mechanisms for the selective sharing of access rights among users. Within the SPE model this is realized using the three binary relations STR, IMP, and EXP, which model the concept of an access control flow graph. The structure relation defines the permissible paths along which the access rights can flow, while the import and export relations describe the actual access control flow at some point in time.

Two state analysis predicates, *Access*(σ,o,r) and *Visible*(σ,o,u), are introduced to define the notions of accessibility and visibility more precisely. The state predicate *Access* is defined over an object o and a region r denoting the ability to "use o within the region r" or "to perform an operation on o within the protection domain defined by region r". The predicate *Visible* extends the predicate *Access* to users. That is, as a first definition, an object o is visible to a user u if the object is accessible within a region owned by u. Thus, given the predicate *Defines* and *Owner*,

$$Defines(\sigma,o,r) \rightarrow Access(\sigma,o,r)$$
$$(Defines(\sigma,o,r) \wedge Owner(\sigma,u,r)) \rightarrow Visible(\sigma,o,u)$$

The general definition of the predicates *Access* and *Visible* require the introduction of two additional predicates to represent (part of) the actual access control flow, namely *Exported* and *Imported*. The predicate *Exported*(σ,o,s,r) says that o is exported from s to r. The predicate *Imported* models the reverse flow. *Imported*(σ,o,r,s) is true when the object o is imported from s into r.

Definition 4.6

$$\textit{Exported}(\sigma,o,s,r) = (s,r)\in STR_\sigma \wedge (o,s)\in EXP_\sigma$$
$$\textit{Imported}(\sigma,o,r,s) = (r,s)\in STR_\sigma \wedge (o,r)\in IMP_\sigma$$

The terms grantor and grantee are used frequently in the literature in this context. Transferred to the SPE context, a grantee is a region (or one of its owners) receiving access right on an object. The region r mentioned in *Imported* and *Exported* is a grantee. A grantor is a region from which an access right is coming (region s in *Imported* and *Exported*). Grantees and grantors can be either active or passive. An active grantee in SPE is the region mentioned in an IMP tuple. That is, an import action is administered by the grantee. Contrary, a passive grantee is characterized by an EXP tuple associated with a region in the contents of the grantee. Note that access rights are given to the grantee without active participation of the latter's. The access permission is, so to say, enforced upon the grantee. A similar distinction between passive and active entities can be found in the Take-Grant model.

The predicates *Exported* and *Imported* alone are not sufficient to fully define accessibility of objects, because they describe local flow of access rights only. For example, when $\textit{Exported}(\sigma,o,s,r)$ is true, it does not necessarily mean that the object o was properly accessible in s. Similarly, if $\textit{Imported}(\sigma,o,r,s)$ is true we know that the environment of r is not empty. Whether the object can be rightfully accessed within the environment remains to be seen. An adequate definition of *Access* can now be given in terms of flow of access rights through a series of regions connected by structure relations, starting at the defining region of the object up to the region for which access permission is checked.

Definition 4.7

In a state $\sigma\in\Sigma$ an object o is *accessible* in a region r, denoted by $\textit{Access}(\sigma,o,r)$, if there exists regions r_i ($i=1..n$, $r_n=r$) such that $\textit{Defines}(\sigma,o,r_1)$ and

$$\textit{Exported}(\sigma,o,r_{i-1},r_i) \text{ or } \textit{Imported}(\sigma,o,r_i,r_{i-1})$$

The concept of visible objects as objects which are accessible in regions owned by a user is defined formally by:

Definition 4.8

In a state $\sigma\in\Sigma$ an object o is *visible* to a user u, denoted by $\textit{Visible}(\sigma,o,u)$, if $\exists r:(u,r)\in OWN \wedge \textit{Access}(\sigma,o,r)$

The actual flow of access rights as described by a state leads to the notion of valid and invalid states. Informally, a state is valid if the information about granted and obtained rights as implied by IMP and EXP is in accordance with the structure and definition relations, STR and DEF. Its formal definition is given in two steps. First, we define state validity for a single object, i.e. partial validity. Second, a state is defined valid when it is partially valid for all objects.

Definition 4.9

A state $\sigma \in \Sigma$ is *partially valid* with respect to object o, i.e. *Pvalid*(σ,o), if $\forall$ (o,r)$\in$IMP :$\exists$s (r,s)$\in$STR $\wedge$ *Access*(σ,o,s) and if $\forall$ (o,r)$\in$EXP : *Access*(σ,o,r).

Note that in this definition no problems are raised by the existence of cycles in the structure relation, nor by multiple paths of grants to the object definition, since the definition of *Access* requires that one structure path exist between the region under consideration and the region where the object is defined such that each structure relation is correctly used for import or export.

Definition 4.10

A state σ is valid, denoted by *Valid*(σ), if $\forall o \in O$:*Pvalid*(σ,o).

The predicate Valid defines a new subspace, the class of valid states $V=\{ \sigma \in \Sigma$: *Valid*(σ)$\}$.

4.1.4. Class relations

In this section we compare the state classes defined above and show that the class definitions are independent in the sense that little information is gained by knowing one class membership only. Thus, the three static security concepts, consistency, ownership, and flow of access rights, are reflected in the model in a clear, separable way. This provides a means to consider alternative definitions of each class so as to arrive at different variations of the SPE model. In this thesis, though, we concentrate on the definitions given and, in particular, concentrate on the class of secure states obtained by taking the intersection of the three classes. Before we investigate the class relationships a few sample SPE states are introduced.

Example 1. The *empty* state $E \in \Sigma$ is the SPE state $< \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing >$.

Example 2. The class of trivial states $T \subseteq \Sigma$ characterized by:

$$O = STR = DEF = IMP = EXP = \varnothing; \; OWN = U \times R.$$

A trivial state contains only relations between users and regions. In particular, each user is an owner of all the existing regions. This state typically reflects the situation found at the start of an actual computer system, in which a group of users is introduced and associated with yet empty file system directories. Observe that the empty state E is a trivial state too. The trivial states are, by definition, consistent, acceptable, and valid.

Example 3. The class of regular states $R \subseteq \Sigma$ is characterized by

$$\forall r \in R: \exists!\, u \in U \wedge (u,r) \in OWN$$

The regular states are characterized by regions with one owner only, that is, no co-owners are allowed. Regular states can be used to analyze systems where the co-ownership supplied by the SPE primitives is not restrictive enough (SPE considers all co-owners as equals). In particular, the regular states can be used to model situations found in file systems where each directory has a single owner attached to it.

Example 4. An SPE state is called a DAG state when the structure relation describes a directed acyclic graph. A subclass of the DAG states form the hierarchical states, where the structure relation describes a tree. DAG and hierarchical states can be used to describe and analyze the protection issues of hierarchical file systems, like the UNIX file system, and scope protection issues found in block structured languages, like PLAIN.

4.1.4.1. Consistent and acceptable states

The class of consistent states and the class of acceptable states are proper subsets of the SPE state universe Σ. Their intersection includes the empty and trivial states. However, consistent and acceptable states are not proper subsets of one another. One can perceive an acceptable state σ in which some user mentioned in the OWN relation is not reflected in the set U and thus the state is inconsistent. Conversely, the state may be consistent but not acceptable due to violation of the rule that each object is defined in one region only.

The fact that a state is consistent does not guarantee that it is acceptable as well. State consistency implies that all the binary relations are in accordance with the basic entity sets. Thus, for example, we know that the users mentioned in the OWN relation also belong to U. However, to be an acceptable state all users mentioned in U must be reflected in OWN by a tuple, which need not be the case. A similar argument holds for the rule that each region has a user attached to it. The last constraint requires checking the existence of a defining region for each object and the assurance that each object is defined in one region only. Note that, in general, knowledge about the consistency of the state does not reduce the work to check acceptability of the state significantly, for an exhaustive check is required.

Conversely, how should one check for consistency of the state with the knowledge that it is acceptable? State acceptability relates the OWN relation to the members of U and R. Thus, part of the consistency constraints are enforced, that is, for those user, regions, and objects mentioned. However, consistency requires more; it requires that all elements of OWN are based on U and R. As there may be a tuple (u,r) in OWN for which both u and r do not belong to the basic entity sets, we are forced to check all elements of OWN and DEF. Moreover, consistency requires that the information in IMP and EXP is consistent as well, which is not implied by any of the acceptability constraints. These must be checked separately. In summary, knowledge about the acceptability of a state does not reduce the work to check for consistency. Thus, the consistency and acceptability properties of a state are independent.

4.1.4.2. Consistent and valid states

The intersection of the class of valid states with the class of consistent states is not empty either, for the trivial states are both consistent and valid by definition.

Similarly to the analysis above, we can consider the question whether the concepts of consistency and validity are independent, and whether knowledge about one of the predicates reveals much information about the other. Obviously, a consistent state need not be valid, because consistency does not enforce any constraint on the flow of access rights as $Valid(\sigma)$ does. This means that we are forced to check the partial validity of all objects to ensure validity of the state.

Conversely, a valid state need not be consistent. For example, consider a valid state with a single region r not belonging to the basic entity set R; thus the state is inconsistent by definition. Then, if this region is not used to transfer access rights, the state remains valid. So, even when the state is valid, checking consistency requires all consistency constraints to be evaluated, which makes consistency and validity independent aspects of the SPE states.

4.1.4.3. Acceptable and valid states

The definition of acceptable states immediately shows that acceptability alone is not sufficient to guarantee validity of a state, because the acceptability predicates do not enforce any constraint on the flow of access rights. Validity of the state is not sufficient for acceptability either, because the constraints are associated with the flow of access rights only. If a region r is not used for this transfer and does not have any owner attached to it, then the state is not acceptable. Therefore, the properties consistency and validity describe a non-empty subclass of the SPE states, it includes the trivial states, and the properties are independent.

4.1.5. Secure states

So far we have introduced three subclasses of the SPE state universe Σ using the notion of acceptability, consistency and validity. The consistency predicate defines the intuitive notion of a consistent state, the acceptability predicate models the entity administration policy for the SPE model and, finally, the validity predicate defines what is meant by consistent administration of the access information in a state so as to enforce an access flow policy.

The pairwise intersections of the classes are not empty, since they include the empty state and the set of trivial states. The trivial states satisfy all three state properties; thus, the intersection of all classes is also non-empty. Furthermore, the analysis of the relationships between the three classes revealed that the predicates defining the three classes are independent. Knowledge about membership of a state in one particular class does not imply membership in any

of the others. As such, the predicates associated with the classes A, C and V are independent. This naturally leads to the definition of secure SPE states, i.e., states which are both acceptable, consistent, and valid.

Definition 4.11

The set of secure states, $S \subset \Sigma$, is defined by $S = A \cap C \cap V$

The set of secure states is used later on to define classes of secure state transformations, that is, operators defined on the domain of secure states and preserving the security properties. First, however, we derive alternative notations for secure states and use them to indicate the cost involved in checking the security property for any given state.

4.2. SPE induced graphs

Each of the binary relationships in the SPE tuple can be used to define a (directed) graph; the vertices are obtained from the entity sets (U,R,O); edges from the binary relations. Subsequently we will look at three different graphs: the ownership, the structure, and the import/export graphs. The construction of these graphs provides a handle on the algorithmic complexity of checking the state security properties.

4.2.1. The ownership graph

The ownership graph is obtained by taking the union of the sets U and R as vertices and by using OWN to define undirected dges. The ownership graph obtained for the trivial states forms a fully bipartite graph, with each vertex in U connected to all vertices in R and vice versa. For secure states we can observe the following behavior:

Theorem 4.1

Let σ be a secure state. Then for each vertex v in the ownership graph derived from the state σ, $1 <= d(v) <= \max(|U|,|R|)$, where d(v) is the degree of the vertex v.

Proof An acceptable secure state means that each user is associated with at least one region and each region with at least one user. This means that at least one edge emanates from each vertex, i.e. $d(v) >= 1$. Moreover, a user is associated with at most all regions and each region is owned at most by all users. □

Corollary If the ownership graph derived from the state σ contains isolated points then the state is not secure.

Proof The isolated points invalidate acceptability constraints a_1 and a_2 of Definition 4.3. □

In the informal introduction of the SPE instruction set, we indicated that the construction of new structure relations requires the invoker of the command to be owner of both regions involved. This requirement can be translated to the owners-graph as follows. Let N(v), called the neighbors function, be the set of vertices incident with v. Then a structure relation can be constructed between r1 and r2 iff $N(r1) \cap N(r2) \neq \emptyset$.

4.2.2. The structure graph

The second graph to be considered is the graph obtained by using the structure relation and the object definition relation. This graph is called the structure graph or SG(σ). A sample SPE state with its corresponding structure graph is illustrated in Figure 4.1a and 4.1b. This graph is formally defined by:

Definition 4.12

The *structure graph* $SG(\sigma)=(V,E)$ for a state $\sigma \in \Sigma$ is a directed graph defined by

$$V = \{r: \exists s\ (r,s) \in STR \vee (s,r) \in STR\}$$
$$E = STR$$

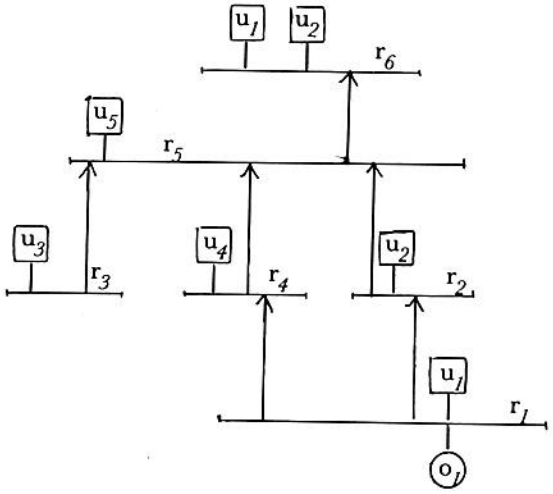


Figure 4.1a An SPE state.

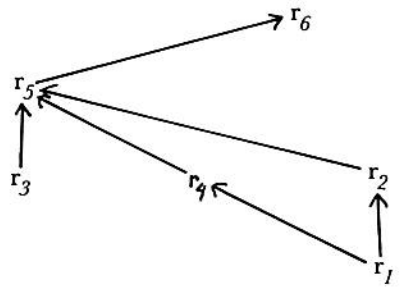


Figure 4.1b The structure graph.

The structure graph visualizes the concept of the environment and the contents of a region. Both are represented by state predicates for the following analysis. The predicates are defined on a structure relation (r,s) in the state σ. We say that r belongs to the contents of s or that region s belongs to the environment r.

Definition 4.13

Contents(σ,r,s) if (r,s) ∈ STR then true, false otherwise
Environment(σ,r,s) if (s,r) ∈ STR then true, false otherwise

The *Exported* and *Imported* predicates defined in a previous section relate to these predicates as follows:

Exported(σ,o,r,s) → *Contents*(σ,r,s)
Imported(σ,o,r,s) → *Environment*(σ,s,r)
Contents(σ,s,r) ←→ *Environment*(σ,r,s)

The definition of secure states allows arbitrary structure relations to be defined, which makes the structure graph a directed graph, which may contain cycles. A side effect is that it is possible to construct a protection state with a region s, which paradoxically belongs both to the environment and the contents of itself. Cycles in the structure graph do not affect the predicate *Access*, because in that case an acyclic path can be found for which *Access* holds too.
The DAG states introduced in section 4.1.4 have the property that the structure relation is acyclic. Thus for these states the resulting structure graph is acyclic as well and describes hierarchical access structures.

Viewing the structure graph as a map for the transfer of access rights implies that objects can be made accessible in the regions belonging to the connected component of this graph where only the object o is represented. Three particular structures to regulate the transfer access rights should be pointed out: a filter region, a mailbox region and an agent region. Examples are shown in Figure 4.2.

Definition 4.14

A region s is called a *filter region* between regions r and t, denoted by *Filter*(σ,s,r,t) when both (r,s) ∈ STR and (s,t)∈ STR.

Definition 4.15

A region s is called a *mailbox region* between regions r and t, denoted by Mailbox(σ,s,r,t), if (r,s)∈STR and (t,s)∈STR.

Definition 4.16

A region s is called an *agent region* between r and t, denoted by Agent(σ,s,r,t), if (s,r)∈STR and (s,t)∈STR.

The filter region is so named, because it can be used as a security barrier for access flow. A mailbox is characterized by the fact that access to an object can be transferred from region r to the region t without any information administered for this object in the intermediate region s. To transfer the object o from r to t it suffices to export it from r and import it into t. Of course, the object o becomes accessible in the mailbox region too. The agent s in Figure 4.2.c region plays a double role in the transfer of access from r to t. Both an import and an export action are associated with it.

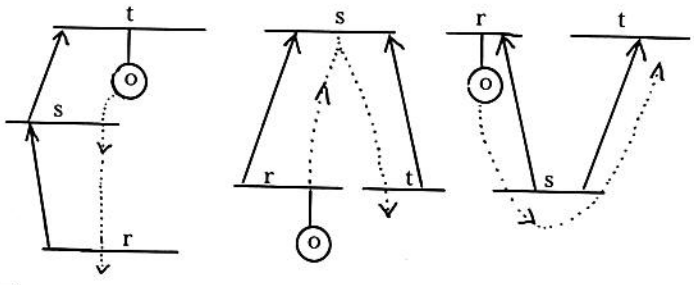


Figure 4.2 a) Filter region, b) mailbox region, c) agent region

4.2.3. The import/export graphs

The definition of accessibility in terms of structure relations and the existence of import/export actions can be made visible using the notion of an import/export graph. An import/export graph for the object o is obtained by taking the object o, its defining region and all regions used in the transport of access permissions as vertices and introduce edges whenever accessibility is transported between the two nodes. Thus, the import/export graph models the actual flow of access permissions for a particular object.

Definition 4.17

Let σ be an SPE state. An import/export graph for object o, denoted by $IE(\sigma,o)=(V,E)$ is a directed graph defined by:

$$V = \{r:(o,r)\in EXP\} \cup \{r:(o,r)\in IMP\} \cup \{o\} \cup \{r:(o,r)\in DEF\} \cup \{r:\exists s,t\in R \wedge Mailbox(\sigma,r,s,t) \wedge (o,t)\in IMP \wedge (o,s)\in EXP\}$$

$$E = \{(o,r): (o,r)\in DEF\} \cup \{(s,r): r,s\in V \wedge \textit{Exported}(\sigma,o,s,r)\} \cup \{(s,r): r,s\in V \wedge \textit{Imported}(\sigma,o,r,s)\}$$

The import/export graph visualizes the notion of valid states. The graph consists of all the regions potentially involved in the validation of the predicate *Access*. The direction of the edges reflect the transport of grants between the regions. For example, observe in Figure 4.3b that the STR direction between r2 and r5 is reversed in the import/export graph.

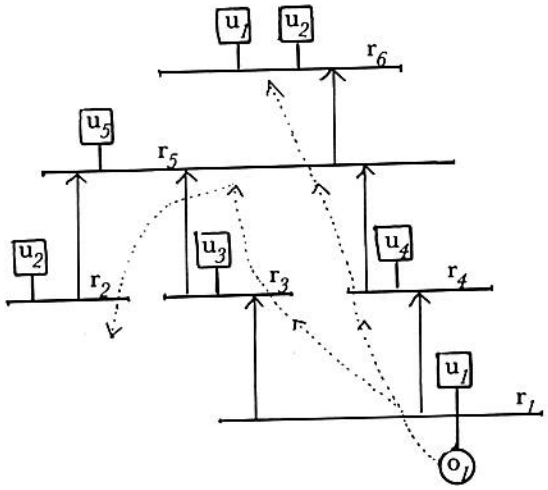


Figure 4.3a An SPE state with import/export actions

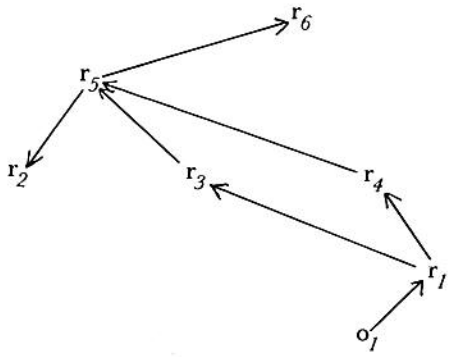


Figure 4.3b The import/export graph $IE(\sigma,o_1)$

In Figure 4.4 we have depicted different import/export graphs for the filter, mailbox, and agent regions shown in Figure 4.2. Observe that accessibility does not depend on the existence of a single specific path of grants from the owner of the object to the grantee, but on the existence of some path. Thus, no distinction is made between grantors, nor as to the order in which grantors grant access. Note too that the definition of accessibility is not affected by cycles in the structure relation, because each cyclic path implies the existence of a non-cyclic path in the structure graph as well, along which access permissions are transferred.

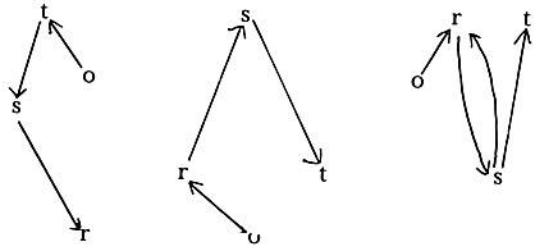


Figure 4.4 Filter, mailbox, and agent regions.

Definition 4.18

A *source graph* is a directed graph with a single source and multiple sinks, such that there exists a path from the source to each vertex.

A source graph has one vertex without incoming arcs and a (non-empty) set of vertices without an outgoing arc. A source graph has no isolated points nor isolated components. Removal of cycles from a source graph results a source graph. In fact, it turns the graph into a tree. Source graphs are related to the

import/export graph as follows:

Theorem 4.2

Let $\sigma \in \Sigma$ and $o \in O$. If *Pvalid*(σ,o) then IE(σ,o) is a source graph.

Proof Assume *Pvalid*(σ,o) and that IE(σ,o) is not a source graph. As σ is a partially valid state for the object o IE(σ,o) has at least two vertices, namely o and its defining region r. The assumption that IE(σ,o) is not a source graph requires two cases to be considered.

First, assume that a vertex $s \neq o$ belongs to IE(σ,o) with indegree zero, i.e. s forms a second source node. Then vertex s is a region, because each import/export graph depends on one object only. Since s has no incoming arcs it can not be a defining region for the object o, nor a mailbox. Therefore, we have to consider two cases, either $(o,s) \in$ IMP or $(o,s) \in$ EXP. Partial validity of the state ensures that for the case $(o,s) \in$ IMP there exists a region r such that $(s,r) \in$ STR and either access to o was exported to r or imported into r. The former makes r a mailbox. The latter ensures that $(o,r) \in$ IMP as well and r is a vertex in the import/export graph. Thus s has an incoming arc based on *Imported*(σ,o,s,r). In all cases s has an incoming arc, which contradicts our assumption.

Second, assume that a single vertex $s \neq o$ exists, such that there is no path from o to s in IE(σ,o). The vertex s is a region and can not be the defining region for o. That is either $(o,s) \in$ IMP, $(o,s) \in$ EXP, or s is a mailbox. The latter implies that there exists a region r such that *Exported*(σ,o,r,s) and no path exists from r to the source o. Thus we may assume that s is not a mailbox.

For both remaining cases (filter region and agent region), partial validity implies that there exists a structure path r_i (i=1..n, $s = r_n$, $(o,r_1) \in$ DEF) such that each region is used for an IMP or EXP tuple. This implies that all regions r_i are vertices in the IE-graph and that they form a directed path, because of the partial access flow *Imported* and *Exported* (construction of IE). The case that multiple vertices exist with the property is handled by induction. Thus, contrary to our assumption there exist a path from o to s, which makes the import/export graph a source graph. □

Theorem 4.3

Let $\sigma \in \Sigma$ and $o \in O$. If IE(σ,o) is source graph then *Pvalid*(σ,o).

Proof IE(σ,o) is a source graph implies that for each $r \in V-\{o\}$ there exists a structure path from the defining region of o to r. Moreover, the construction of the IE-graph ensures that each structure relation (r,t) implies either *Imported*(σ,o,r,t) or *Exported*(σ,o,r,t). Together, this implies that for each such r *Access*(σ,o,r) is true. As V exhausts all regions used in DEF, IMP, and EXP, the requirements for *Pvalid* (def. 4.6) are fulfilled. □

Theorem 4.3 provides a means to estimate the cost of establishing partial validity of a state. This cost is equal to the cost of constructing the import/export graph and to checking the source graph property. The latter can be determined using a depth-first search on the graph in $O(|E|)$.

Both the import/export graph and the structure are based on structure relations. Yet, import/export graphs are not subgraphs of the structure graph, because the directionality of the arcs is dictated by the import-export operations and because multiple arcs are allowed between the vertices. The cyclic behavior of the structure graph can be inferred from the import/export graph under limited conditions.

Theorem 4.4

Let σ be a valid state. If IMP$=\varnothing$ and IE(σ,o) is cyclic then SG(σ) is cyclic.

Proof Let $r_1 \ldots r_n$ be regions on a cycle in IE(σ,o). By Definition 4.15 these are vertices in SG(σ) too. IMP$=\varnothing$ means that each edge (r_i,r_{i+1}) is the result of an export. By Definition 4.15 this means that a structure relation (r_i,r_{i+1}) exists, but then SG(σ) contains a cycle. □

Corollary Let σ be a valid state. If EXP$=\varnothing$ and IE(σ,o) is cyclic then SG(σ) is cyclic.

Theorem 4.5

Let σ be a valid state. If SG(σ) is a DAG and IE(σ,o) is cyclic then there exists regions r,s,t in IE(σ,o) such that either Mailbox(σ,r,s,t) or Agent(σ,r,s,t) is true.

Proof Let $r_1 \ldots r_n$ be the regions on a cycle in IE(σ,o). Each edge (r_i,r_{i+1}) on this cycle corresponds with either $(r_i,r_{i+1}) \in$ STR when the object is *Exported*, or with $(r_{i+1},r_i) \in$ STR when the object is *Imported*. Because the structure graph SG(σ) is a DAG there exists at least one r_i such that either $(r_{i-1},r_i) \in$ STR and $(r_{i+1},r_i) \in$ STR, that is r_i is a mailbox, or $(r_i,r_{i-1}) \in$ STR and $(r_i,r_{i+1}) \in$ STR, that is r_i is an agent. □

4.2.4. Algorithmic costs

The introduction of static security constraints alone is insufficient without a means to test an arbitrary state for security. An efficient representation of the SPE states is needed, for which the complexity of deriving the security property is cheap. The three graphs introduced in the previous section provide a good handle on this problem, for they indicate that the security properties can be translated into (directed) graph properties. The consistency property is checked by repeatedly checking the elements of the sets. Any data structure representing sets with an efficient set retrieval operation can be used for this purpose. Acceptability of the state is checked using the ownership graph and the structure

graph, since a necessary prerequisite of state acceptability is that the ownership graph forms a bipartite graph without loose points. This is easily checked. The outdegree of all vertices representing objects in the structure graph should be 1 to satisfy the third acceptability constraint. The construction of both graphs is limited by the number of their elements.

Finally, state validity is checked using the structure graph and the import/export graphs. This is a costly operation, because the import/export graphs for all objects defined should be (virtually) constructed and checked for the source graph properties. However, the latter is limited by the number of edges in the import/export graph.

The value of the predicates *Defined* and *Owner* can be determined using table lookup of the corresponding tuple in the state component. The predicates *Imported* and *Exported* require at least two tuple searches, one for fixing the structure relation and one for the import and export respectively. That is, the cost is limited by the size of the contents (Section 3.1) or environment or the region considered. The value of the predicates *Access* and *Visible* require the construction of an import/export graph. However, when validity of the state is known beforehand, the costs are limited as well.

Theorem 4.6

Let σ be a valid state. Then the value of the predicate *Access*(σ,o,r) can be determined considering r and the regions in its contents only.

Proof State validity ensures that if an IMP or EXP tuple is associated with a region r, r is a node in IE(σ,o) (which forms a source graph). *Access*(σ,o,r) is true when either the object is defined in r, imported into r, or exported to r. The latter requires a check on the administration associated with the regions in the contents only. That is, does there exist a region s such that *Exported*(σ,o,s,r)? The other cases do not require other regions to be considered at all. □

The overall costs involved in checking state security, in the context of a static system, is one good reason to translate the security properties into state modification invariants, which are addressed next.

4.3. State transformations

For the definition of a secure programming environment, static security properties alone are not sufficient, because such an environment is dynamic in nature. We have seen that the dynamics can be described by an abstract machine, which is characterized by the notion of a state and by a mechanism to generate new states using an instruction set. Thus, we should extend the notion of static state security with dynamic aspects in such a way as to benefit from the definitions and results derived so far.

The mechanism to generate new states for the abstract machine is an instruction set. Instructions are mappings, which take a state of the machine and one or more user-supplied parameters, and generate a new state together with information (output) for the user. The user-supplied parameters and user oriented information depends on the design philosophy of the instruction set, such as which primitive actions are combined into a single instruction and which authorization policies should be enforced. The analysis presented in this section addresses their behavior with respect to the protection state only. A class of simple state mappings is defined, which take an SPE state as an argument and produce a new SPE state. For these transformations we derive the *pre-* and *post-*conditions for making security an invariant property of the mapping.

The definition of instruction sets opens a new area of security problems. That is, under what circumstances can a state be changed while preserving the security properties and who is permitted to apply an instruction to a state? Different schemes can be considered, ranging from allowing any user to apply all instructions, to a rigid scheme where the use of instructions is coupled to the existence of additional information in the state affected. For the SPE machine model an authorization policy is defined which couples the effects of an instruction with the user responsible for its invocation. That is, users are allowed to exercise instructions as long as they hold access permissions to the objects manipulated or own them. In that context, we extend the simple mappings with parameters, giving a characterization of different implementations of SPE based instruction sets.

Finally, some baseline properties are defined for arbitrary SPE instruction sets, which describe what should be considered as *well-defined* instruction sets. This characterization is used in the following chapter to define and analyze a particular set, the SPE instruction set; for which the behavior is specified by *pre-* and *post-*conditions and for which we show that it is *well-defined.*

4.3.1. Classes of state transformations

The class of simple state transformations, Φ, is the collection of all functions on the domain Σ and image Σ. This class is more than we really need for protection. For example, it includes functions mapping a consistent state into an inconsistent state. Therefore, we focus on a small subset of Φ, namely those functions which map secure states into secure states. As state security is a complex property, we concentrate on subclasses of Φ satisfying a single security property.

Definition 4.19

A state transformation $\phi:\Sigma \rightarrow \Sigma$ is *acceptable* iff $\forall\ \sigma \in A:\phi(\sigma) \in A$

Similarly, consistent and valid functions or state transformations are defined. Functions satisfying all three properties are called secure state transformations, the prime candidates for usage in the SPE instruction sets. A sample secure

state transformation is the identity function. The identity function is used later on for instructions which extract information from the state, leaving the state itself unchanged.

The introduction of secure state transformations as mappings with the security property as a mapping invariant has far-reaching consequences for the analysis of security questions, i.e. can a given insecure state be derived from any secure state, or does the state describe unauthorized access privileges? It implies that given an initial secure state and repetitive application of a secure transformation, a secure state is obtained. Thus, when the instruction set is chosen from the secure state transformations, from the protection point of view expressed by the security policy decisions in the SPE model, no insecure state can be reached.

A further reduction on the functions in Φ is obtained by considering two subclasses with simple behavioral properties only. The advantage of this approach will be a cost reduction in determining the security properties of the state generated. Moreover, the two classes exhibit opposite behavior. The first class, *MI*, consists of the incremental state transformations, functions which extend one or more of the state description components.

Definition 4.20

A state transformation $\phi \in \Phi$ is *incremental* if $\forall \sigma \in \Sigma$

$$\begin{array}{lll} U_{\phi(\sigma)} \supset U_\sigma & OWN_{\phi(\sigma)} \supset OWN_\sigma & IMP_{\phi(\sigma)} \supset IMP_\sigma \\ R_{\phi(\sigma)} \supset R_\sigma & STR_{\phi(\sigma)} \supset STR_\sigma & EXP_{\phi(\sigma)} \supset EXP_\sigma \\ O_{\phi(\sigma)} \supset O_\sigma & DEF_{\phi(\sigma)} \supset DEF_\sigma & \end{array}$$

The *decremental* state transformations *MD* are defined by replacing $\supset$ by $\subset$ in Definition 4.18. The intersection of the *incremental* and *decremental* state transformations consists of mappings which behave like the identity function, but which, by their implementation, may deliver information on the protection state to the user. For both classes we translate the consistency, acceptability, and validity property into *pre-* and *post-* conditions for the function.

4.3.2. Incremental state transformations

The nature of incremental state transformations, extending the state components, can be used directly to observe that accessible and visible objects in the initial state remain accessible and visible in the generated state. Thus, extension of the state does not have a drastic effect on the access permissions modeled. For the decremental state transformations this is not true, as we will see shortly.

Theorem 4.7

Let ϕ be an incremental state transformation. Then

$$\textit{Defines}(\sigma,o,r) \rightarrow \textit{Defines}(\phi(\sigma),o,r)$$
$$\textit{Exported}(\sigma,o,r,s) \rightarrow \textit{Exported}(\phi(\sigma),o,r,s)$$
$$\textit{Imported}(\sigma,o,r,s) \rightarrow \textit{Imported}(\phi(\sigma),o,r,s)$$
$$\textit{Access}(\sigma,o,r) \rightarrow \textit{Access}(\phi(\sigma),o,r)$$
$$\textit{Visible}(\sigma,o,u) \rightarrow \textit{Visible}(\phi(\sigma),o,u)$$

Proof Follows directly from the definitions. □

4.3.2.1. Consistent incremental state transformations

The consistent incremental transformations are characterized by keeping the consistency property invariant, which leads to the following observation.

Theorem 4.8

Let $\phi \in MI$; ϕ is consistent *iff* $\forall \sigma \in C$ and

1) $OWN_{\phi(\sigma)} - OWN_{\sigma} \subset U_{\phi(\sigma)} \times R_{\phi(\sigma)}$
2) $STR_{\phi(\sigma)} - STR_{\sigma} \subset R_{\phi(\sigma)} \times R_{\phi(\sigma)}$
3) $DEF_{\phi(\sigma)} - DEF_{\sigma} \subset O_{\phi(\sigma)} \times R_{\phi(\sigma)}$
4) $IMP_{\phi(\sigma)} - IMP_{\sigma} \subset O_{\phi(\sigma)} \times R_{\phi(\sigma)}$
5) $EXP_{\phi(\sigma)} - EXP_{\sigma} \subset O_{\phi(\sigma)} \times R_{\phi(\sigma)}$

Proof

$=>$ Let ϕ be consistent and $\sigma \in C$. Thus by definition $\phi(\sigma) \in C$. For a consistent state $OWN_{\sigma} \subseteq U_{\sigma} \times R_{\sigma}$ and for the image $OWN_{\phi(\sigma)} \subseteq U_{\phi(\sigma)} \times R_{\phi(\sigma)}$. As $(OWN_{\phi(\sigma)} - OWN_{\sigma}) \subseteq (OWN_{\phi(\sigma)} - OWN_{\sigma}) \cup OWN_{\sigma} \subseteq OWN_{\phi(\sigma)}$ requirement 1) is satisfied. A similar argument proves 2) to 5).

$<=$ One must show that the image of a consistent state is consistent under the proposed conditions. An incremental ϕ means that $OWN_{\sigma} \subseteq OWN_{\phi(\sigma)}$ and $OWN_{\sigma} \subseteq U_{\sigma} \times R_{\sigma} \subseteq U_{\phi(\sigma)} \times R_{\phi(\sigma)}$. Thus, when requirement 1) is satisfied, $OWN_{\phi(\sigma)} \subseteq U_{\phi(\sigma)} \times R_{\phi(\sigma)}$. The other conditions are proved analogously. □

4.3.2.2. Acceptable incremental state transformations

The transformations which keep the state acceptability property invariant are characterized by the following *pre-* and *post-* conditions on the state transformation.

Theorem 4.9

If an incremental state transformation ϕ changes the binary relations STR, EXP, and IMP only, then ϕ is acceptable.

Proof The acceptability property is a relationship between the three basic entity

sets and the binary relations OWN and DEF. Thus, whenever these components of a state are not changed by an incremental state transformation, the resulting state remains acceptable. □

Theorem 4.10

Let $\phi \in MI$; ϕ is acceptable *iff* $\sigma \in A$ and

1) $\forall u \in U_{\phi(\sigma)}\text{-}U_\sigma : \exists r: (u,r) \in OWN_{\phi(\sigma)}$
2) $\forall r \in R_{\phi(\sigma)}\text{-}R_\sigma : \exists u: (u,r) \in OWN_{\phi(\sigma)}$
3) $\forall o \in O_{\phi(\sigma)}\text{-}O_\sigma: \exists! r: (o,r) \in DEF_{\phi(\sigma)}$
4) $\forall (o,r) \in DEF_{\phi(\sigma)}\text{-}DEF_\sigma: o \notin O_\sigma$

Proof => When ϕ is an acceptable transformation, its image $\phi(\sigma)$ with $\sigma \in A$ is acceptable too. Since ϕ is incremental the acceptability constraints $a_1..a_3$ of Definition 4.3 are satisfied for the image of the state components, which imply constraints 1), 2), and 3). Constraint 4) is implied by assertion a_3 of def. 4.3 as well, because new object definitions cannot violate the uniqueness of the defining region property for objects in the state σ.

<= We prove that $\phi(\sigma) \in A$ given $\sigma \in A$ and the truth of the four conditions. Assume that $\phi(\sigma) \notin A$; then three cases should be considered.

First, there exists a user $u \in \phi(\sigma)$ such that there does not exist a region $r \in \phi(\sigma)$ with the property $(u,r) \in OWN_{\phi(\sigma)}$. Since ϕ is incremental u should belong to $U_{\phi(\sigma)}\text{-}U_\sigma$, but according to 1) there is a region with this property.

Second, a similar argument holds for regions and condition a_2 of def. 4.3.

Third, let $o \in \phi(\sigma)$. The assumption that ϕ is not acceptable means that either there does not exists a region in $\phi(\sigma)$ such that $(o,r) \in DEF_{\phi(\sigma)}$, or associated with o are multiple regions with this property. The former is prohibited by condition 3), which states that precisely one such region can be found for each new object. Uniqueness of (o,r) for objects in O_σ is guaranteed by condition 4. Conclusion: under the constraints posed ϕ can not be unacceptable. □

4.3.2.3. Valid incremental state transformations

The analysis of valid incremental state transformations is more involved, because the validity of a state depends on the combination of STR, DEF, IMP and EXP. In fact, the information embodied by STR and DEF constrain IMP and EXP. To reduce the complexity of the state validity analysis, each component is discussed separately.

Theorem 4.11

Let ϕ be an incremental state transformation; if only U,R,O and OWN are changed then ϕ is valid.

Proof Follows directly from the definition of validity. □

Theorem 4.12

Let ϕ be an incremental state transformation; if ϕ extends the component DEF only then ϕ is valid.

Proof Let (o,r) be a new DEF tuple. Then two cases should be distinguished. First, if there does not exist a $(o,s) \in DEF_\sigma$ with $s \neq r$ then by definition *Pvalid*$(\phi(\sigma),o)$ is true. Second, if there exists a $(o,s) \in DEF_\sigma$ with $s \neq r$ then $IE(\phi(\sigma),o)$ remains a source graph (although the state becomes unacceptable). Thus according to Theorem 4.3 *Pvalid*$(\phi(\sigma),o)$ is true. These observations hold for all objects $O_\sigma = O_{\phi(\sigma)}$. Thus by Definition 4.7 *Valid*$(\phi(\sigma))$ is true. □

Note that extension of the DEF component alone may lead to a valid but unacceptable state. For example, it may violate the constraint that each object is defined in one region only. Therefore, in the SPE instruction set DEF and O can be changed with one instruction. The theorems above show that this combined usage does not invalidate the state.

Theorem 4.13

Let ϕ be an incremental state transformation; if ϕ extends the component STR only then ϕ is valid.

Proof The easiest way to prove this is to consider the import/export graph of an object. The extension of STR is reflected in the graph with possible new edges. However, the resulting graph remains a source graph, because it can never change its source (which is the object) and is thus, by Theorem 4.2, partially valid. This property holds for all import/export graphs derived for objects and thus the final state is valid. □

Theorem 4.14

Let ϕ be an incremental state transformation which extends the component IMP only. If $\forall\ (o,r) \in IMP_{\phi(\sigma)}\text{-}IMP_\sigma\ \exists s$: $(r,s) \in STR_\sigma$ and *Access*(σ,o,s) then ϕ is valid.

Proof Let σ be a valid state and $(o,r) \in IMP_{\phi(\sigma)}\text{-}IMP_\sigma$ such that it satisfies the requirements. *Access*(σ,o,s) means that there exists a path from o to s in the import/export graph $IE(\sigma,o)$. Moreover, *Imported*$(\phi(\sigma),o,r,s)$ is true, because $(r,s) \in STR_\sigma$ and o is accessible in s. Thus by Definition 4.14 there exists a path from o to r in $IE(\phi(\sigma),o)$. This property holds for all r' such that $(o,r') \in IMP_{\phi(\sigma)}\text{-}IMP_\sigma$, which makes $\phi(\sigma)$ a source graph and therefore partially valid by Theorem 4.3. Since this property holds for all objects o mentioned in the extended import list, the final state is valid by definition. □

This theorem expresses a sufficient condition for validity for a limited extension of IMP only. In general, addition of an arbitrary number of elements to IMP requires a connectivity check for the corresponding import/export graphs, which

is more expensive than the limited case covered by this theorem. This theorem states that a local check suffices to guarantee validity. Thus any compound change to IMP should be rewritten as a series of small changes or shown to be equivalent to such a series without invalidating the security policy implemented by the instruction set.

Theorem 4.15

Let ϕ be an incremental state transformation which extends the component EXP only. If $\forall$ (o,r)$\in EXP_{\phi(\sigma)}$-EXP_{σ}: *Access*(σ,o,r) then ϕ is valid.

Proof Analogous to previous theorem.□

The separation of the validity invariant into properties of restricted function classes does not limit the class of derivable states. We can always consider a state transformation as a composite of a number of (more primitive) functions, each affecting a single component. When the primitive steps preserve validity, the composite is valid as well.

Theorem 4.16

Let ϕ be an incremental state transformation; if ϕ is the composition of one or more functions ϕ_i, i=1..n, such that each ϕ_i is valid then ϕ is a valid state transformation

Proof By transitivity of state transformations.□

4.3.3. Decremental state transformations

The complementary class of state transformations are the decremental transformations which reduce one or more of the state components. Unlike the incremental transformations, the predicates defined for the states, i.e. *Access*, *Imported*, *Exported*, and *Defines*, do not remain valid in all situations. For example, in general *Access*(σ,o,r) does not imply *Access*($\phi(\sigma)$,o,r), because when ϕ removes some essential structure elements or import/export relation IE($\phi(\sigma)$,o) is not a source graph.
The transformation of the state constraints for this class of functions is again presented by the partial state security constraints, i.e. consistency, acceptability and validity. Functions satisfying all three constraints form the class of secure decremental state transformations.

4.3.3.1. Consistent decremental state transformations

The consistency property of states is preserved easily for the decremental functions. As consistency binds the binary relations with the elements in the three basic sets, decremental functions can invalidate the consistency when the basic sets are reduced only. First, consider the situation that no changes are

made to the basic sets.

Theorem 4.17

If ϕ is a decremental state transformation and the three basic sets U, R, and O are not affected by ϕ, then ϕ is consistent.

Proof Follows directly from Definition 4.2 and Definition 4.17. □

When a binary relation is affected by ϕ, we should ensure that all information discarded from the state description is done properly. That is, whenever regions are removed, users associated with that region should own another region or be removed from the state as well. When users are removed then the region not owned by other users, should be removed too. Moreover, the removal of regions implies the removal of objects defined in that region. These observations are combined in the following theorem.

Theorem 4.18

Let ϕ be a decremental state transformation: ϕ is consistent iff $\forall \sigma \in C$

1) $\forall u \in U_\sigma - U_{\phi(\sigma)}$ not $\exists r$: $(u,r) \in OWN\phi(\sigma)$
2) $\forall o \in O_\sigma - O_{\phi(\sigma)}$ not $\exists r$: $(o,r) \in DEF\phi(\sigma) \vee$
   $(o,r) \in IMP \vee (o,r) \in EXP$
3) $\forall r \in R_\sigma - R_{\phi(\sigma)}$:
   $\neg \exists u$: $(u,r) \in OWN\phi(\sigma) \wedge \neg \exists s$: $(s,r) \in STR\phi(\sigma) \wedge$
   $\neg \exists t$: $(r,t) \in STR\phi(\sigma) \wedge \neg \exists o$: $(o,r) \in DEF\phi(\sigma)$

Proof

=> Definition of consistent transformation and Definition 4.2, 4.16, and 4.17.

<= Let $\sigma \in C$ and assume that ϕ is not a consistent state transformation. Then five different cases should be considered, based on the five composite state components.

Assume that there exists $(u,r) \in OWN\phi(\sigma)$ causing ϕ to be inconsistent and such that either u or r does not belong to $U\phi(\sigma)$ and $R\phi(\sigma)$ respectively. However, $u \notin U_{\phi(\sigma)}$ means that condition 1) is not true. When $r \notin R_{\phi(\sigma)}$ then condition 3) is to be obeyed, contradicting the assumption that (u,r) is a cause for inconsistency.

Similar arguments can be used to conclude that none of the other possibilities are in effect, which means that our assumption that ϕ is not consistent under the proposed conditions is not true. □

A few special cases are of interest in proving consistency of various instructions later on. Note, for example, that the removal of objects violates the consistency constraint if it isn't accompanied by the removal of the related EXP, IMP, or DEF.

Corollary Let ϕ be a decremental state transformation which changes DEF and O only. Then ϕ is consistent iff $\forall\sigma\in C\ \forall o\in O_\sigma\text{-}O_{\phi(\sigma)}$ $(o,r)\notin DEF_{\phi(\sigma)} \wedge (o,r)\notin IMP_{\phi(\sigma)} \wedge (o,r)\notin EXP_{\phi(\sigma)}$.

Corollary Let ϕ be a decremental state transformation which changes OWN and U only. Then ϕ is consistent iff $\forall\sigma\in C\ \forall u\in U_\sigma\text{-}U_{\phi(\sigma)}$: $(u,r)\notin OWN_{\phi(\sigma)}$

4.3.3.2. Acceptable decremental state transformations

The acceptable decremental functions remove users and objects from the state or revoke ownership of users over regions. Transformation of the security property to function invariants is done similarly to the validity check in section 4.3.2.2, i.e. using different classes of state transformations, which change the state in a limited way. First, observe a simple property of acceptable decremental state transformations.

Theorem 4.19

Let ϕ be a decremental state transformation; if U, R, O, DEF, and OWN are not changed then ϕ is acceptable.

Proof Follows from Definition 4.3 and the decremental behavior of ϕ.□

Theorem 4.20

Let ϕ be a decremental state transformation; if O is changed and R, U are not changed then ϕ is acceptable

Proof => Let σ be an acceptable state. Then $\forall o\in O_\sigma\ \exists r:(o,r)\in DEF_\sigma$, i.e. property a_3 holds. Since $O_\sigma \supset O_{\phi(\sigma)}$ property a_3 holds for $\phi(\sigma)$ too. As only O is changed, property a_1 and a_2 hold automatically for $\phi(\sigma)$, which makes $\phi(\sigma)$ acceptable.□

Note that a transformation of this kind leaves an acceptable, yet inconsistent state behind, which is a good reason to combine the removal of objects with removal of their definitions in instruction sets.

Theorem 4.21

Let ϕ be a decremental state transformation changing O and DEF only; ϕ is acceptable iff $\forall\sigma\in A\ \forall(o,r)\in DEF_\sigma\text{-}DEF_{\phi(\sigma)} : o\notin O_{\phi(\sigma)}$

Proof => Let ϕ and σ be acceptable and $(o,r)\in DEF_\sigma\text{-}DEF_{\phi(\sigma)}$ with $o\in O_{\phi(\sigma)}$. Since σ is an acceptable state there is only one region r with $(o,r)\in DEF_\sigma$ and removal of this (o,r) invalidates constraint a_3 of Definition 4.3 for $\phi(\sigma)$ when $o\in O_{\phi(\sigma)}$. But this contradicts our assumption that ϕ is acceptable and $o\in O_{\phi(\sigma)}$. Such an (o,r) does not exist.
<= Let σ be an acceptable state. Only O and DEF are changed, so a_1 and a_2 of Definition 4.3 are satisfied for $\phi(\sigma)$. To satisfy a_3 we should distinguish two

cases. First, Theorem 4.20 shows that the removal of an object alone won't invalidate a_3. Second, when $(o,r) \in DEF_\sigma\text{-}DEF_{\phi(\sigma)}$ then a_3 of Definition 4.3 is not true when the object o belongs to the objects in the final state, which is prohibited by the constraint. □

The intertwining of conditions a_1 and a_2 makes analysis of the acceptability invariant involving decremental changes to OWN, R, and U more involved.

Theorem 4.22

Let ϕ be a decremental state transformation changing R only; then ϕ is acceptable iff $\forall \sigma \in A\ \forall\ r \in R_\sigma\text{-}R_{\phi(\sigma)}$:

$$((u,r) \in OWN_\sigma \rightarrow \exists s:(u,s) \in OWN_{\phi(\sigma)}) \wedge \neg\ \exists o:(o,r) \in DEF_\sigma.$$

Proof => Let ϕ and σ be acceptable and assume that $r \in R_\sigma\text{-}R_{\phi(\sigma)}$. Then two cases should be considered.

First, assume that there does not exist an $s(\neq r)$ with $(u,s) \in OWN_{\phi(\sigma)}$. Then region r is the only region assigned to user u, and, because ϕ is acceptable, u should be removed from the state as well to ensure a_1. This, however, is prohibited by the assumption that only R is changed. Thus at least one other region s exists in $R_{\phi(\sigma)}$ with $(u,s) \in OWN_{\phi(\sigma)}$.

Second, assume there is an object o, such that $(o,r) \in DEF_\sigma$. As ϕ changes R only $(o,r) \in DEF_{\phi(\sigma)}$. Yet each object has one defining region, therefore assertion a_3 of def 4.3. does not hold for $\phi(\sigma)$, violating the assumption that ϕ is acceptable.

<= Let σ be acceptable and assume that ϕ is not acceptable. Then three cases should be considered.

First, assume that a_1 of def. 4.2 is not fulfilled for $\phi(\sigma)$. That is, there exist a user $u \in U_\sigma$ and no region $s \in R_{\phi(\sigma)}$ such that $(u,s) \in OWN_{\phi(\sigma)}$. As σ is acceptable there exists at least one region r with $(u,r) \in OWN_\sigma$ and $(u,r) \notin OWN_{\phi(\sigma)}$, thus $r \in R_\sigma\text{-}R_{\phi(\sigma)}$. But then there exists a region s with $(u,s) \in OWN_{\phi(\sigma)}$, which contradicts our assumption.

Second, assume that constraint a_2 is not fulfilled, that is, there exists a region $r \in R_{\phi(\sigma)}$ which is not owned by a user. As ϕ is decremental on R only, this contradicts the assumption that the state σ is acceptable in the first place.

Third, assume that a_3 is not satisfied, which means that there is an object o defined in multiple regions or there exists no such region at all. The former would contradict the assumption that σ is acceptable. The latter is prohibited by the constraints of the theorem. In conclusion, under the constraints posed ϕ is acceptable. □

Theorem 4.23

Let ϕ be a decremental state transformation changing U and OWN only. Then ϕ is acceptable iff $\forall\sigma\in A$

1) $\forall u\in U_\sigma\text{-}U_{\phi(\sigma)}\ \exists v\in U_{\phi(\sigma)}\ (u,r)\in OWN_\sigma \wedge \rightarrow (v,r)\in OWN_\sigma$

2) $\forall (u,r)\in OWN_\sigma\text{-}OWN_{\phi(\sigma)}\ \exists v\in U_{\phi(\sigma)}\ (v,r)\in OWN_{\phi(\sigma)}$

Proof => Let σ and ϕ be acceptable. Then, as only U and OWN are changed, constraints a_2 and a_3 of Definition 4.2 hold for $\phi(\sigma)$. If $u\in U_\sigma\text{-}U_{\phi(\sigma)}$ then a_1 is true by definition, and thus 1) holds. When $(u,r)\in OWN_\sigma\text{-}OWN_{\phi(\sigma)}$ it must be ensured that region r has an owner in $\phi(\sigma)$, which is prescribed by 2).
<= Assume that σ is acceptable and ϕ is not. That is, as only U and OWN are changed, there exists a tuple $(u,r)\in OWN_\sigma\text{-}OWN_{\phi(\sigma)}$, which requires two cases to be considered. First, $u\in U_{\phi(\sigma)}$ and no r' with $(u,r')\in OWN_{\phi(\sigma)}$, which is not allowed by condition 1). Second, no u' exists such that $u'\in U_{\phi(\sigma)}$ and $(u',r)\in OWN_{\phi(\sigma)}$, which is prohibited by constraint 2). Thus, the assumption that ϕ is not acceptable does not hold under the constraints of the theorem. □

4.3.3.3. Valid decremental state transformations

As with the analysis of the valid incremental functions, the validity constraints for decremental functions are analyzed on a component basis. In general, the validity of the state may be violated when one of the sets DEF, STR, IMP, or EXP is reduced. Conversely, whenever these components are not affected by a transformation, it is valid.

Theorem 4.24

Let ϕ be a decremental state transformation; if ϕ does not affect the state components STR, DEF, IMP, and EXP then ϕ is valid.

Proof Follows directly from definitions 4.8 and 4.17.□

Theorem 4.25

Let ϕ be a decremental state transformation; if ϕ changes the components DEF and O only and $\forall (o,r)\in DEF_\sigma\text{-}DEF_{\phi(\sigma)}$:

$\neg\ \exists s\ \mathit{Imported}(\sigma,o,r,s)\ \wedge$

$\neg\ \exists s\ \mathit{Exported}(\sigma,o,r,s)$

then ϕ is valid

Proof Assume that the conditions hold for $(o,r)\in DEF_\sigma\text{-}DEF_{\phi(\sigma)}$, $\sigma\in V$, and ϕ is not valid. In other words, there is an object o such that $IE(\phi(\sigma),o)$ is not a source graph. As the conditions hold for o, this source graph is either empty or consists of the single node o. The former can never invalidate the validity of the state. The latter is a source graph by definition, which contradicts the assumption that ϕ is not valid.□

The conditions in Theorem 4.25 ensure that no access permissions were outstanding for the object being removed, i.e. the import/export graph associated with o is an empty graph. In general, we should check removed structure relations for not being essential in the transfer of access permissions, that is, not making the state invalid. Checking a structure relation for this property is complicated, because a multitude of situations exists. For example, consider the situation depicted in Figure 4.7, where access has been passed by means of appropriate imports and exports to all regions, as indicated by its import/export graph.

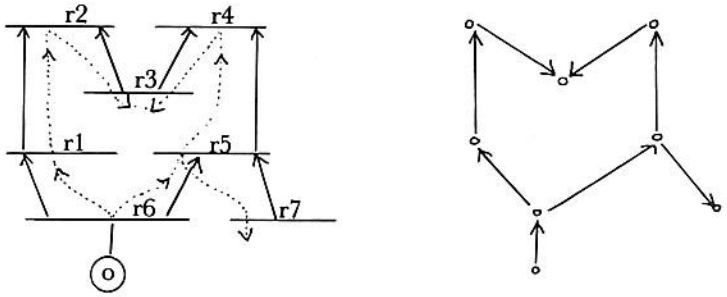


Figure 4.7 An example access control flow and IE graph

Removing the structure relation (r7,r5) is permissible when import of o into r7 has been undone first. Then no objects are imported into r7 any more and the structure relation can be removed without making the state invalid. Removing the structure relation (r1,r2) is more involved. Removing the export from r1 will not disconnect the graph. However, to see this, all regions which base their access permission on *Access*(σ,o,r2) should be checked for the existence of a different grantor. In this case region r3 plays such a role.

Theorem 4.26

Let ϕ be a decremental state transformation which changes only the component STR. If $\forall\ \sigma \in V$, $\forall\ (r,s) \in STR_{\sigma}\text{-}STR_{\phi(\sigma)}$ and for each object o such that either *Exported*(σ,o,r,s) or *Imported*(σ,o,r,s) the import/export graph IE(ϕ(σ),o) is a source graph, then ϕ is valid

Proof Follows directly using Theorem 4.3. □

As indicated in the example above, some situations can be checked more easily using local state information only. The next theorem tells us that whenever the subordinate region is not involved in any access flow, the structure can be removed without violation of the state validity.

Theorem 4.27

Let ϕ be a decremental state transformation which changes the component STR only. If $\forall(r,s) \in STR_\sigma\text{-}STR_{\phi(\sigma)}$: $\neg\exists o\ ((o,r) \in EXP_\sigma \vee (o,r) \in IMP_\sigma)$ then ϕ is valid.

Proof Assume that the conditions hold for a valid state σ and that ϕ is not valid. By Theorem 4.2 and 4.3 this means that there is an object o such that $IE(\sigma,o)$ is not a source graph, while $IE(\phi(\sigma),o)$ was. Thus the removal of the structure relation (r,s) makes $IE(\phi(\sigma),o)$ to contain a second source node, i.e. r. However, this situation is excluded by the theorem conditions, because r is not involved in access transfers, contradicting our assumptions. □

A similar theorem exists for the superordinate regions. In this case we have to check whether superordinate regions function as mailboxes in the transfer of access rights.

Theorem 4.28

Let ϕ be a decremental state transformation which changes STR only. If

$\forall(r,s) \in STR_\sigma\text{-}STR_{\phi(\sigma)}$

$\neg\exists(o,s) \in EXP_{\phi(\sigma)}$ and

$\neg\exists t\ Mailbox(\sigma,r,s,t)) \wedge ((o,r) \in EXP_{\phi(\sigma)} \wedge (o,t) \in IMP))$ and

$\neg\exists t\ Mailbox(\sigma,r,s,t)) \wedge ((o,r) \in IMP_{\phi(\sigma)} \wedge (o,t) \in EXP))$

then ϕ is valid

Proof The proof runs analogously to the one given for the previous theorem. Assume that the conditions hold for a valid state σ and that ϕ is not valid. By Theorem 4.3 and 4.4 this means that there is an object o such that $IE(\phi(\sigma),o)$ is not a source graph, while $IE(\sigma,o)$ was. Thus the last incoming edge for s has been removed by ϕ. This means that s is a second source node in $IE(\phi(\sigma)$. This criterion can be translated to the occurrence of export actions associated with s (s plays an active role in access flow) or s being used as a mailbox between r and another region t (s is passive). However, both situations are excluded by the three constraints, which means that no edges emanate from s and thus s can not be a node in $IE(\phi(\sigma),o)$, contradicting our assumption.□

Both theorems illustrate that removal of border nodes in the import/export graph, i.e. nodes without emanating edges, preserves validity of the state. The removal of an arbitrary structure relation leads to a connectivity check of the IE graphs of all objects affected by the removal.

The analysis of removing access grants is analogous to the approach taken for removing structures. Whenever access permissions are removed from the border nodes of an import/export graph, i.e. nodes without arcs emanating, we know that it remains connected and thus partially valid.

Theorem 4.29

Let ϕ be a decremental state transformation, which changes the component IMP only. If $\forall(o,r) \in IMP_{\sigma}\text{-}IMP_{\phi(\sigma)}$
$(o,r) \notin EXP \wedge \neg\exists t\ ((t,r) \in STR \wedge \mathit{Imported}(\phi(\sigma),o,r,t))$
then ϕ is valid.

Proof For each import removed, condition a) guarantees that the region is not used as an agent, which exports the access permissions. Constraint b) guarantees that no region in the contents of r makes use of access permission to o.□

Theorem 4.30

Let ϕ be a decremental state transformation, which changes the component EXP only. If $\forall(o,r) \in EXP_{\sigma}\text{-}EXP_{\phi(\sigma)}$
$\neg\exists t\ \neg\exists s\ (r,s) \in STR \wedge (o,s) \in EXP_{\phi(\sigma)} \wedge (o,t) \in IMP) \wedge$
$\mathit{Mailbox}(\sigma,r,s,t)$
then ϕ is valid.

Proof Analogous to previous theorem. □

4.3.4. Constraint variations

In a similar way, the properties of the state transformations can be checked for the models derived from SPE, as described in section 4.1.4. For example, we have introduced the DAG and hierarchical states as alternatives to describe access control properties. In general, for DAG states the cost involved in guaranteeing for non-cyclic behavior requires a graph connectivity algorithm. In a limited number of cases, the DAG property can be checked without resorting to a global graph analysis. For example, removing structure relations never invalidate the DAG property and addition of structure relations are sometimes checked easily.

Theorem 4.31

Let ϕ be an incremental state transformation. If for all DAG states σ
$\forall\ (r,s) \in STR_{\phi(\sigma)}\text{-}STR_{\sigma}$: in-degree($\phi(\sigma)$,r)=0 $\vee$
in-degree($\phi(\sigma)$,s)=0
then ϕ preserves the DAG property.

Proof Property of a DAG □.

Theorem 4.32

Let ϕ be a decremental state transformation then ϕ preserves the DAG property.

Proof Property of a DAG □.

4.4. Authorization

The state transformations were discussed in the previous section from a formal world perspective, ignoring the practical problem of how a state transformation is actually triggered. In implementing the SPE model, someone or something is needed from outside this closed world, such as an active entity which is held responsible for the application of the state change. This active entity introduces a new dimension in security problems, summarized in the single phrase "who may do what and when?". More specifically, under what conditions can a user apply a state transformation and how are these conditions represented?

The answer to this kind of protection problems is the use of a decision procedure, which implements an authorization policy. Each state transformation is first checked against this policy and when the decision procedure does not prohibit the transformation, it is performed. The authorization policy consists of a set of (self referential or static) rules using information extracted from the protection state and the identity of the user responsible. For example, the authorization policy might enforce that only an owner of an object may remove it, or that a user initiating a "create object" request has to be in control over the environment where the object is placed. This requires both a static rule to express the constraints and information from the protection state to which it is applied.

4.4.1. Authorized state transformations

Formalization of authorization in the SPE model requires a new class of functions, the authorized state transformations Γ, defined below. The informal introduction of authorization above indicates that three dimensions are relevant for the state transformation: the activators, the instruction set, and the states of the protection system.

Definition 4.21

An authorized state transformation $\gamma \in \Gamma$ is defined by the function

$$\gamma: \text{USERS} \times \Phi \times \Sigma \rightarrow \Sigma$$

and the authorization function

$$\textit{Authorized}: \text{U} \times \Sigma \times \Phi \rightarrow \{\text{true,false}\}$$

such that

$$\gamma(u,\phi,\sigma) \begin{cases} \phi(\sigma) & \text{when } \textit{Authorized}(u,\phi,\sigma)=\textit{true} \\ \sigma & \text{otherwise} \end{cases}$$

Each authorized state transformation in the context of SPE is a state transformation defined over the domain of *activators*, i.e. the active entities, an instruction set, i.e. the state transformations, and the set of SPE states. The application of authorized state transformation leads to a new state $\phi(\sigma)$ if and only if the activator u is authorized to apply the state transformation. When the

activator is not authorized, the result of the invocation is the application of the identity function. Note that the authorization function is defined on users represented in the protection state only.
The introduction of the authorized transformations as an extension to the state transformations implies that the results derived in the previous sections on the behavior of these mappings carry over to the authorized state transformations. In particular, we will use the subset of secure authorized state transformations in the construction of an SPE instruction set.

4.4.2. Authorization policies

Before we present a formal definition of the authorization policy of the SPE model, some alternatives are presented. The authorization policies, as described in the literature, can be divided into two classes: the static and the dynamic policies. Under a static policy the authorization information to evaluate the predicate *Authorized* is fixed. It is defined with the creation of the protection system. The static policy approach is illustrated by four examples, each of which emphasizes different components of the authorization function.
Example 1 The static authorization policy with emphasis on the activators and the instruction set can be found in the architecture of many computer systems. For example, the DEC PDP-11 machines use a two-level architecture of user and kernel mode, which can be considered as the activators of the machine instructions. Part of the machine instruction set is not available within user mode and the association of mode with instruction is static, i.e. built into the hardware.
Example 2 A static policy with emphasis on the states and the instruction set can be found in compilation of high-level programming languages. Here, there is one activator only, the programmer, and the authorization for an instruction invocation depends on the accessibility of the object, which is often restricted to the scope of its definition, i.e. the state of the compiling process.
Example 3 An authorization policy with emphasis on the activators, operations, and the state of the system can be found in the military environment, such as the multi-level security classes [Bell74] where a user may read/write all the documents for which his clearance level equals or exceeds the level associated with the document.
Example 4 Within the context of SPE a static policy can be defined involving both a group of activators G, an instruction set OP and a subspace of Σ. The policy enforces that users in G can issue any state transformation, provided that the state remains secure. Other users are restricted to the instruction set OP.

$$\mathit{Authorized}(u,\phi,\sigma) \equiv \phi(\sigma) \in S \wedge (u \in G \vee (u \in U\text{-}G \wedge \phi \in OP))$$

Dynamic authorization policies are characterized by using the state of the protection system in combination with the identity of the activator. That is, the

authorization information, as well as the decision procedure, is represented as objects within the protection state. This use of the protection state provides a means to change the authorization information dynamically. One can effectively propagate access rights to newly defined users or change the security level of objects.

Example 5 An example of a dynamic authorization procedure is the password scheme found in a multi-user systems. The decision procedure for system access makes use of the database of <user, password> associations to allow or deny service.

Example 6 An example of a dynamic authorization policy is the theoretical model of Harrison-Ruzzo-Ullman [Harrison76], where the value of *Authorized* is defined by an access matrix M, the protection state description within this model. Each entry M[u,o] designates the function(s) (or right(s)) of a user u with respect to the object o. The access matrix defines the value of *Authorized*(u,ϕ,σ) by:

$$\textit{Authorized}(u,\phi,\sigma) \equiv M[u,o]=\phi \wedge (o\in O_\sigma \vee o\in U_\sigma) \wedge u\in U_\sigma$$

Observe that a dynamic policy needs information on static rules to describe the policy employed for redefinition of the authorization information too. For example, part of the policy inherent to the HRU model is implied by the access matrix structure. That is, adding a new right to an element of the table is authorized if the element already exists. Alternatively, one cannot give rights to nonexistent users nor manipulate nonexistent objects.

4.4.3. SPE authorization policy

The authorization policy for SPE is based on two concepts: the affected region and cooperation when access flow is concerned. A region is considered an affected region if it plays a role in the state change caused by a SPE instruction. Thus, addition and deletion of binary relations from a state turns the region mentioned into an affected region, formally defined by:

Definition 4.22

Let $\phi\in\Phi$; the set of *affected* regions is defined by

$$\begin{aligned}AR(\phi,\sigma)\\ &\{r:\ (u,r)\in OWN_{\phi(\sigma)}\text{-}OWN_\sigma\} \cup \{r:\ (u,r)\in OWN_\sigma\text{-}OWN_{\phi(\sigma)}\} \cup\\ &\{r:\ (o,r)\in EXP_{\phi(\sigma)}\text{-}EXP_\sigma\} \cup \{r:\ (o,r)\in EXP_\sigma\text{-}EXP_{\phi(\sigma)}\} \cup\\ &\{r:\ (o,r)\in IMP_{\phi(\sigma)}\text{-}IMP_\sigma\} \cup \{r:\ (o,r)\in IMP_\sigma\text{-}IMP_{\phi(\sigma)}\} \cup\\ &\{r:\ (o,r)\in DEF_{\phi(\sigma)}\text{-}DEF_\sigma\} \cup \{r:\ (o,r)\in DEF_\sigma\text{-}DEF_{\phi(\sigma)}\}\end{aligned}$$

The concept of affected regions does not apply to changes made to the structure relation. Establishment of a communication path requires participants to cooperate, while revocation can be issued by either one. These constraints are being taken care of separately, which gives the following formal definition of the

SPE authorization policy:

Definition 4.23

The SPE authorization predicate *Authorized*(u,ϕ,σ) is true when

a) $\forall\ r \in AR(\phi,\sigma)$: $(u,r) \in OWN_\sigma$ and

b) $\forall\ (r,s) \in STR_{\phi(\sigma)}\text{-}STR_\sigma$: $(u,r) \in OWN_\sigma \wedge (u,s) \in OWN_\sigma$ and

c) $\forall\ (r,s) \in STR_\sigma\text{-}STR_{\phi(\sigma)}$: $(u,r) \in OWN_\sigma \vee (u,s) \in OWN_\sigma$

This definition indicates that implementation of the model can be centered around the region concept. Access authorization is based on the affected regions, therefore storage of the protection state can be decentralized to the region objects. This approach differs from traditional user oriented storage (capability lists) and object oriented storage (access control lists) of authorization information.

4.5. SPE instruction sets

Now that we have defined both the concept of authorized state transformations and the predicate *Authorized*, instruction sets for the SPE model can be defined.

Definition 4.24

An SPE instruction set is a finite set of authorized secure state transformations which enforce the SPE authorization policy.

This definition of an instruction set is still too abstract from an implementation point of view. Different instruction sets can be perceived and what constitute a good set is an open question. First, however, we address the syntactic means to specify an instruction, which in turn requires a specification technique.

Until now we have discussed the SPE state transformation ϕ on an abstract level. It has been characterized by its effects on the protection state, without precise specification of its syntax. The design and analysis of an instruction set requires more details. Within this thesis, an SPE instruction is specified in terms of a name, a parameter list, a *pre*-condition, and a *post*-condition, similar to the O-function in [Parnas72]. To emphasize the authorization decision it is specified separately from the instruction *pre*-condition. The interpretation of an instruction call is the same as defined for authorized state transformations, whenever the *auth*- and *pre*-condition hold, the state is changed as defined by the *post-condition*.

A V-function in this specification technique is used to define the state analysis instructions, those instructions which extract information from the protection state description and transform it to readable information, or to a Boolean value for decision control.

For example, an instruction is defined to introduce a region 'new' to be owned by 'friend' and to provide access to a single object 'file' within 'new'.

command share(actor, area, file, friend, new)
auth (actor, area) $\in$ OWN
pre (file, area) $\in$ DEF
post (friend, new) $\in$ OWN $\wedge$
(area, new) $\in$ STR $\wedge$
(file, area) $\in$ EXP

Figure 4.8 A sample SPE instruction definition.

In the discussion of authorized state transformations we stated that failure of the *Authorized* results in applying the identity state transformation. Similarly for the *pre*-condition. In this respect the *pre*- and *post*- conditions specify the effect of an if-then statement in algorithmic programming languages. Note, though, that in an implementation of the model information about errors should be sent to the activator or security officer when surveillance is needed.

4.5.1. Well-defined instruction sets

In the foregoing sections the SPE protection model is formally defined, both in terms of state components and in terms of state transformation invariants. The state transformations and their constraints were used to give an outline of instruction sets for the SPE model, including the definition of an SPE authorization policy. By no means are these security properties the only possible view of the world regarding access control. For example, the DAG and hierarchical states can be considered as a model derivable from the SPE model. Moreover, the specification technique for the instruction sets provides the means to extend the authorization policy. For example, the *pre*-condition of the instruction in Figure 4.8 can be extended to require the existence of a particular object or the negation of some state property.

The potentially large number of instruction sets based on the SPE model raises the question what properties they should have. We conclude this chapter with an indication of *well-defined* instruction sets and analyze whether this property can be derived for an arbitrary SPE instruction set algorithmically.

4.5.2. Completeness criterion

The first property we consider important for *well-defined* SPE instruction sets is the ability to generate all secure states from a given secure initial state.

Definition 4.25

An instruction set I satisfies the *completeness* criterion if there exists a secure state σ_0 such that for each secure state σ there exists a series ϕ_j (j=1..n) and $\sigma=\phi_n \bullet .. \bullet \phi_1 (\sigma_0)$

The restriction of this definition to a initial secure state stems from the authorization policy, which requires each action to be triggered by a known activator. Therefore, the initial state minimally consists of a single user u who owns a region r. Such a state has been classified as a primitive state and is known to be secure.

The completeness property of instruction sets can not be decided by a simple algorithm, for it can be shown that the behavior of a Turing Machine can be encoded in the SPE model in such a way that if completeness were decidable, then so is the halting problem. Because the halting problem is known to be undecidable, determining the completeness of a SPE instruction set is undecidable.

Theorem 4.33

It is undecidable whether an arbitrary SPE instruction set is complete.

Proof Let T be an arbitrary Turing machine. We show how its states are encoded into SPE states and how the moves of T are mapped to SPE instructions. The tape symbols are associated with unique user names, and the

non-blank cells are associated with unique regions. The Turing Machine states p,q,... are associated with objects p,q,... in the protection model. The SPE state representing the Turing Machine state has precisely one such object in a region to represent the head of machine.
Let T be in state q, then T has scanned a finite number of k cells, and the cells k+1,k+2,.. are blank. The state q is represented by a secure SPE state σ such that:

1) If cell s of the tape contains the symbol X, then $(X,s) \in OWN_{\sigma}$
2) $(s,s+1) \in STR$, which orders the tape cells
3) $(End,k) \in DEF$, which marks the end of the tape
4) If the tape head is positioned at cell s, then $(q,s) \in DEF$

An example encoding of the Turing machine in state q whose first cells hold 'rst', and the tape head positioned at cell 2, is shown in Figure 4.9.

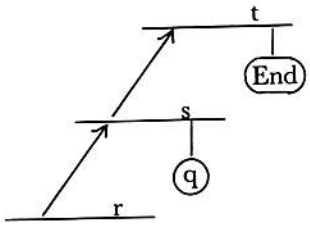


Figure 4.9 Turing machine tape simulation

A move f(q,X)=(p,Y,L) is represented by an SPE instruction MOVEqXL that removes the object q from region r and grants p to region s, where r represents the current position of the tape head, and s represents the cell to the left of r (which is determined by the STR relation). The SPE instruction becomes:

MOVEqXL(r,s)
pre $(s,r) \in STR \wedge (q,r) \in DEF \wedge (X,r) \in OWN$
post $(q,r) \notin DEF \wedge (X,r) \notin OWN \wedge (Y,r) \in OWN \wedge (p,s) \in DEF$

The move f(q,X)=(p,Y,R) requires two cases to be considered, one for the state containing End.

MOVEqXR(r,s)
pre (r,s)∈STR ∧ (q,r)∈DEF ∧ (X,r)∈OWN
post (q,r)∉DEF ∧ (X,r)∉OWN ∧ (Y,s)∈OWN ∧ (p,s)∈DEF

MOVEqEnd(r,s)
pre s∉R ∧ (End,r)∈DEF ∧ (q,r)∈OWN ∧ (q,r)∈OWN
post (End,r)∉DEF ∧ (X,r)∉OWN ∧ (r,s)∈STR
∧ (Y,s)∈OWN ∧ (End,s)∈DEF

If the Turing machine reaches its final state q_f, then q_f will be entered into some region r. Equivalently, halting of the Turing machine means that the TM enters the final state q_f i.e. the object End is entered in some region. If the completeness problem were decidable, it would be possible to decide beforehand, given the above set of instructions, whether the final state is entered, which means that we have a decision procedure for the halting problem, which is known not to exist. □

This theorem shows that there is no hope for finding a general algorithm to decide *completeness* for an arbitrary instruction set and initial state. This result need not discourage us in searching instruction sets with predictable behavior. For example, if the domains REGIONS, OBJECTS, and USERS as well as the instruction set are finite, completeness becomes decidable, since a brute force method can be applied to check each possible state for reachability.

4.5.3. Compensation criterion

The second criterion for a *well-defined* instruction set is that it provides the means to describe and cause not only the dissemination of access permissions, but also the revocation of these permissions. That is, for each incremental instruction applied to a secure state one can find a sequence which brings back the original state. We restrict ourselves to instruction sets of incremental and decremental instructions only, where the compensation criterion is satisfied when for each incremental instruction there exists a composition of (decremental) instructions to undo its effect.

Definition 4.26

An SPE instruction set I is said to satisfy the *compensation* criterion if each incremental $\phi \in I$ is either incremental or decremental and for each $\phi \in I$ and secure state σ there exists a series decremental instructions $\phi_j \in I$ (j=1..n) such that $\phi_n \circ ... \circ \phi_1 \circ \phi(\sigma) = \sigma$

The compensation criterion does not impose constraints on the SPE authorization policy. Activators of the decremental instructions may be either identical to the activator of the initial incremental instruction or may be users with the same access rights, which allows for the construction of systems where permissions are revoked by system-wide users (e.g. the super-user in UNIX).

Definition 4.24 rules out instruction sets containing state transformations which both extend and reduce the protection state components. In general, such an instruction can be considered a composition of (more primitive) incremental and decremental instructions. As far as authorization is concerned, such a decomposition may require authorization information to be saved between the execution of the component mappings in the protection state, i.e. turns it into a dynamic authorization policy.

Note that Definition 4.24 requires instruction sets to be able to compensate for incremental instructions immediately; information added to the protection state can be removed as long as it has not been used. For example, object definitions can be undone before access rights on it are passed to other regions. The general problem of undoing side effects is handled by a revocation policy, as described in section 5.3.

The definition of the compensation concept raises the question as to the existence of an algorithm to determine this property for an arbitrary SPE instruction set. Obviously, when each incremental instruction is accompanied by an inverse decremental instruction, i.e. the *post*-condition of the former implies the *pre*-condition of the latter under all circumstances and the combined application is equivalent to the no-op operation, the compensation criterion is guaranteed.

Theorem 4.34

If I is a SPE instruction set and each incremental state transformation has an inverse in I then I fulfills the *compensation* criterion.

Proof Trivial.□

Again there is no hope of finding an algorithm which decides this property for arbitrary SPE instruction sets. Assume that such an algorithm exists. Then it would work for the commands defined in the proof of Theorem 4.33 as well. This means that the algorithm solves the halting problem by coupling the original state with the final state of the Turing machine.

A handle on the problem is obtained by placing restrictions on the instruction sets considered. If all instructions are classified as either incremental or decremental and behave independently of the protection state then a brute force method exists to check this property. By independent behavior we mean that the number of elements added to the protection state by an incremental instruction does not depend on the state to which it is applied. This way, one such extension is representative for all cases.

Theorem 4.35

There exists an algorithm which decides the *compensation* criterion for SPE instruction sets consisting of incremental and decremental instructions only, each of which behaves independently of the protection state.

Proof We should prove that for each incremental $\phi \in I$ there exists a series ϕ_j (j=1..n) such that ϕ_j is decremental and for each secure state σ

$$\phi_n \circ \ldots \circ \phi_1 \circ \phi(\sigma) = \sigma$$

Each protection state is described by finite sets, thus the incremental (decremental) state transformation extend (reduce) the state with (by) a finite number of elements. Moreover, we assumed that the number of elements added does not depend on the state it is applied. Therefore, to undo the effect of an incremental instruction, all elements added should be removed, which places an upper bound on the number of decremental instructions to be applied. If we assume that an incremental instruction adds N elements to the binary relations, then all sequences of length N can be generated and checked, because the instruction set is finite as well. □

4.5.4. Minimality criterion

The third criterion for *well-defined* instruction sets is minimality, that is, the set consists of a minimal number of instructions. An extreme situation would be an instruction set consisting of a single instruction with many parameters. Obviously, this is an unworkable situation, because the interpretation of this instruction has to deal with many cases. Minimality in this respect should be coupled with the notion of instruction orthogonality or instruction independence, which is influenced by the access control constraints and the authorization policy. Therefore, we adopt as definition of minimality the non-existence of instruction overlap. For each instruction (changing the state description) it should be impossible to give an equivalent series from the same instruction set.

Definition 4.27

An instruction set I satisfies the *minimality* criterion if $\forall \phi \in I$ $(\phi \neq \iota)$ there does not exists a series $\phi_j \neq \phi$ (j=0..n), such that for any secure state σ

$$\phi(\sigma) = \phi_n \circ \ldots \circ \phi_0(\sigma)$$

The question whether within the context of SPE an algorithm exists to determine minimality for an arbitrary instruction set should be answered negatively. As with the compensation criterion, minimality of the instruction set depends on the ability to generate a particular secure state, which is known to be undecidable for the unbounded name space NAMES and the unconstrained form of the instruction set. When the name space becomes bounded minimality can be decided for the finite instruction set at considerable expense.

One method to prove minimality for a restricted class of instruction sets, a set of decremental/incremental instructions behaving independent of the protection state, runs as follows. Separate the instructions into groups affecting the same portion of the protection state. Then for each group the *pre-* conditions can be checked for overlap using a proof technique tailored to the set.

4.6. Summary

In this chapter we have introduced the SPE model by formally specifying the static and dynamic properties of SPE protection states. An SPE protection state is considered secure when it satisfies the independent properties *consistency*, *acceptability*, and *validity*, which model the security properties introduced in Section 3.3. The security properties of SPE states have been related with directed graphs, which gave a handle on their algorithmic complexity. The dynamic properties of SPE states are cast into state transformation invariants, which thereby describe properties for many SPE instruction sets. In Section 4.4 we have indicated how authorization policies enter the scene as a separate dimension of access control. A particular authorization policy for the SPE model has been introduced, which models the rule that each user changing the protection state should own (or be responsible for) the protection domains where the changes take place. Finally, in section 4.5 desirable properties of SPE instruction sets, i.e. minimality criterion, compensation criterion and completeness criterion, have been introduced as forming a basis for *well-defined* instruction sets. Although no algorithm exists that can check the well-definedness property handed an arbitrary SPE instruction set, a given set may well be proved to satisfy as shown in the next chapter.

5

AN SPE INSTRUCTION SET

The SPE model does not prescribe a single instruction set, but forms a basis for many instruction sets. In this section we formally specify one such instruction set using the specification technique introduced in section 4.4 and show that is satisfies the well-definedness properties. Next, this instruction set is applied to the problem of revocation, undoing protection actions of the past. After an introduction of the semantic problems associated with revocation in general and an indication of the implementation approaches taken in various systems, two revocation policies for SPE are described: a chronological and a goal-seeking revocation policy. A sketch of the associated algorithms is given.

In Section 5.5 an important question for all protection systems is addressed: "who can steal privileges?" First, we study the capabilities of the instruction set in predicting derivable states. Then, the notions of stealing and conspiracy are given a more formal basis. The SPE instruction set is used to predict the leakage of privileges and the means to prohibit this. Finally, the observations are used to construct a model (and representation) to analyze this problem in an SPE environment.

In Section 5.6 we address the construction of protection system commands, i.e. SPE programs, and their use in the construction of alternative SPE based protection systems. The technique is illustrated by simulating two formal access control models [Harrison76, Jones76], which shows the modeling power of our proposed scheme and simultaneously illustrates the differences with the aforementioned models.

5.1. An SPE instruction set

The SPE instruction set *OP* is split into three categories: the incremental, the decremental, and the state analysis instructions. These categories are denoted by *IP*,*DP*, and *AP* respectively. The former two categories are O-functions in Parnas's terminology, the latter are V-functions, i.e. value returning instructions. As the V-functions do not change the protection state, they are no threat to security being modeled. Note, however, that they may breach security policies related to confidentiality. We restrict ourselves in this thesis to a formal specification of the O-functions. The state analysis instructions are introduced where appropriate.

As described before, the semantics of the instructions are specified by the *auth*-, *pre*-, and *post*-conditions. To differentiate the initial and final state of each instruction, the component sets of the final state are marked with a single quote. Moreover, the final state is considered identical to the initial state, except for those components mentioned explicitly in the *post*-condition.

Whenever a *pre*- or *auth*-condition fails for an activation of an instruction, or the *post*-condition of the state can not be guaranteed to hold, the instruction is considered in *error* and the result of the instruction is the same as would be obtained by applying the identity state transformation. We have omitted the formal description of this behavior from the specifications. In any implementation of the instruction set an error message should be returned to the activator, who, by convention, is denoted by the first parameter of an instruction.

5.1.1. Incremental instructions

The incremental instructions *IP* extend one or more components of an SPE state and consist of the instructions:

add_region(u,s)	- introduce a new region
add_object(u,r,o)	- introduce a new object in a region
add_owner(u,r,v)	- introduce a co-owner for a region
add_struct(u,r,s)	- introduce a structure relation
add_export(u,r,o)	- export an object
add_import(u,r,o)	- import an object

The first instruction, add_region(u,s), introduces a new region s and designates u as its owner. The *auth*-condition implies that the activator of the command is known within the initial state. For these instructions we will show that they are authorized secure state transformations in the sense of Definition 4.21.

Definition 5.1

add_region(u:USERS;s:REGIONS)
auth $u \in U$
pre $s \notin R$
post $R' = R \cup \{s\} \wedge OWN' = OWN \cup \{(u,s)\}$

The parameter list of an instruction includes the parameter type, which can be seen as an integrity constraint on the actual parameter. Note, however, that these constraints are primarily of importance in an implementation of the instruction set. Within the model all three domains are unbounded and disjoint.

Theorem 5.1

The instruction add_region(u,s) is an authorized secure state transformation.

Proof We should prove that the result of applying the instruction to a secure initial state σ is secure.
According to Definition 4.17 add_region is an incremental state transformation. The instruction is consistent, because the *post*-condition ensures that region s is included in the base set R, a necessary requirement to fulfill the conditions of Theorem 4.8. The *post*-condition enforces that in the derived state the region s has been assigned an owner, no objects are introduced, and no structure relations. Thus, the conditions for Theorem 4.10 are fulfilled, making add_region an acceptable state transformation. The state transformation is valid, because increment of R and OWN does not invalidate existing access privileges (Theorem 4.11), which makes the instruction a secure state transformation. The newly defined region is not an affected region in the sense of Definition 4.19, but as the activator of the instruction becomes its owner, the instruction is authorized by Definition 4.21.□

Co-owners are introduced for a region using add_owner(u,r,v). The activator u grants ownership rights on region r to user v. Hence, it makes v equally powerful to u in relationship to this region. This instruction is the only way to introduce new users, because, as for the definition of a new object, someone must be qualified to change the protection state.

Definition 5.2

add_owner(u:USERS; r:REGIONS; v:USERS)
auth $(u,r) \in OWN$
pre $(v,r) \notin OWN$
post $OWN' = OWN \cup \{(v,r)\} \wedge U' = U \cup \{v\}$

Note that security of the initial state ensures that $(u,r) \in OWN$ implies $r \in R$.

Theorem 5.2

The instruction add_owner(u,r,v) is an authorized secure state transformation.

Proof Definition 4.17 tells us that add_owner is an incremental state transformation. The *post*-condition ensures that condition 1) of Theorem 4.10 is satisfied. Conditions 2), 3), and 4) are satisfied by definition. Thus, add_owner is an acceptable state transformation. Theorem 4.8 shows that add_owner is consistent. Validity is obtained from the behavior condition and not changing any of the components IMP, EXP, DEF, and O. Finally, the instruction is an authorized secure state transformation, because the *pre*-condition ensures that the activator of the command, u, is an owner of the affected region r.□

Extension of the structure relationship STR is provided for by add_struct(u,r,s), which requires bidirectional cooperation to satisfy security property SP-4; the activator should be an owner of both regions involved in the establishment of an communication path. In particular, the authorization condition may require an action to introduce u as a co-owner of the region s first.

Definition 5.3

add_struct(u:USERS; r,s:REGIONS)
auth $(u,r)\in OWN \wedge (u,s)\in OWN$
pre $(r,s)\notin STR$
post $STR'=STR \cup \{(r,s)\}$

Theorem 5.3

The instruction add_struct(u,r,s) is an authorized secure state transformation.

Proof According to Definition 4.17 add_struct is an incremental state transformation. The add_region instruction is consistent by Theorem 4.8 and by the implication of the *post*-condition that $(r,s)\in R'\times R'$. The instruction is acceptable by Theorem 4.10 and valid by Theorem 4.13. Together, the instruction is a secure state transformation. The set of affected regions is empty, but by definition Authorized(u,add_region,σ) is true.□

New objects are introduced with the add_object instruction, which expects the name of the activator, the name of the new object, and the name of the region with which the object is to be associated, i.e. its defining region.

Definition 5.4

add_object(u:USERS; r:REGIONS; o:OBJECTS)
auth $(u,r)\in OWN$
pre $o\notin O$
post $O'=O\cup\{o\} \wedge DEF'=DEF\cup\{(o,r)\}$

Theorem 5.4

The instruction add_object(u,r,o) is an authorized secure state transformation.

Proof According to Definition 4.17 add_region is an incremental state transformation. The instruction is consistent by Theorem 4.8 and the fact that the *post*-condition implies $(o,r) \in O' \times R'$. Theorem 4.10's condition 1) and 2) are fulfilled automatically and conditions 3) and 4) are satisfied by the constraint on the result. Thus, the instruction is an acceptable state transformation. The add_object instruction is valid by Theorem 4.11 and the observation that the new object is added to O. Together, the instruction is a secure state transformation. The set of affected regions is $\{r\}$ and the authorization prescribes that the activator owns r. Thus, by Definition 4.21 the instruction is authorized. □

The add_import and add_export instructions are the object sharing instructions. That is, add_export makes an object accessible within the environment of a region, while add_import makes an object accessible within the environment accessible within the region itself.

Definition 5.5

add_import(u:USERS; r:REGIONS; o:OBJECTS)
auth $(u,r) \in OWN$
pre $(o,r) \notin IMP \wedge \exists s \in environment(\sigma,r): Access(\sigma,o,s)$
post $IMP' = IMP \cup \{(o,r)\}$

Theorem 5.5

The instruction add_import(u,r,o) is an authorized secure state transformation.

Proof The instruction is incremental by Definition 4.17. Consistency is guaranteed by the fact that both r and o are known in the initial state and thus the conditions for Theorem 4.8 are fulfilled. Acceptability of the instruction follows directly from Theorem 4.9, while the transformation is valid by Theorem 4.14. Together, add_import is a secure state transformation. The instruction is authorized, because u owns the affected region r. □

Definition 5.6

add_export(u:USERS; r:REGIONS; o:OBJECTS)
auth $(u,r) \in OWN$
pre $(o,r) \notin EXP \wedge Access(\sigma,o,r)$
post $EXP' = EXP \cup \{(o,r)\}$

Theorem 5.6

The instruction add_export(u,r,o) is an authorized secure state transformation.

Proof Analogous to previous proof using Theorems 4.8, 4.9, and 4.15. □

5.1.2. Decremental instructions

The second category of SPE instructions is formed by the decremental authorized secure state transformations *DP*, summarized by:

del_region(u,r)	- remove a region
del_object(u,r,o)	- remove an object
del_owner(u,r,v)	- remove a co-owner
del_struct(u,r,s)	- remove a structure relation
del_import(u,r,o)	- remove an import
del_export(u,r,o)	- remove an export

The instructions in this category are defined such that application of the instruction immediately after its corresponding incremental instruction functions as a no-op. In fact, these instructions are compensating instructions for the incremental instructions. For example, let σ be a state and apply add_object(u,r,o) giving σ' then the application of del_object(u,o,r) to the state σ' brings back the state σ. These properties turn the proposed instruction set into a *well-defined* SPE instruction set, which is shown formally shortly.

The regions in SPE play a central role and the removal of a region r in the compensation instruction requires that the region is not used for object definition, nor is it used within structure relations. The former implies that no exports are outstanding for local objects, the latter that no imports exists for r (in a secure state of course) and thus no exports for imported objects either. A boolean V-function is defined for this purpose, called Notused(r), which is formally specified by:

function Notused(r:REGIONS): boolean
begin
Notused := ($\neg$ $\exists$o:(o,r)$\in$ DEF) $\wedge$ ($\neg$ $\exists$s:((r,s)$\in$STR $\vee$ (s,r)$\in$STR))
end

With the compensation criterion in mind, the instruction del_region is defined by:

Definition 5.7
del_region(u:USERS;r:REGIONS)
auth (u,r)$\in$OWN
pre $\neg$($\exists$v$\neq$u:(v,r)$\in$OWN) $\wedge$ Notused(r) $\wedge$ $\exists$s$\neq$r:(u,s)$\in$OWN
post OWN'=OWN-{(u,r)} $\wedge$ R'=R-{r}

Theorem 5.7
The instruction del_region(u,r) is an authorized secure state transformation.

Proof We shall show that if the instruction is applied to a secure state then the

final state is secure too. Firstly, observe that the instruction is decremental by Definition 4.17. Theorem 4.18 is used to prove consistency of the mapping. U and O do not change; therefore conditions 1) and 2) are fulfilled. Furthermore the *pre*-condition, i.e. the term Notused, ensures that no (u,r)∈OWN, no (o,r)∈DEF, no (r,s)∈STR, and no (s,r)∈STR exists. Thus Theorem 4.18 ensures consistency.
Assume that the *pre*-condition holds. Then, to be acceptable, the constraints in Definition 4.3 should be true. This can be seen as follows. The sole user associated with r is u, who owns another region r' by the *pre*-condition. Thus, the mapping satisfies constraint a_1 for acceptability. Second, no objects are defined in the region being removed, thus satisfying constraint a_2. Finally, constraint a_3 is satisfied by definition. Thus, the mapping is acceptable. Validity of the mapping follows directly from Theorem 4.24. Together, the instruction del_region is a secure state transformation. Finally, the *pre*-condition ensures that the invoker of the command is associated with the affected regions, making it an authorized secure transformation. □

Co-owners are removed with the del_owner instruction, which ensures that users are equally powerful.

Definition 5.8

del_owner(u:USERS; r:REGIONS; v:USERS)
auth (u,r)∈ OWN
pre (v,r)∈ OWN
post OWN'=OWN-{(v,r)} ∧ (∃s≠r:(v,s)∈OWN ∨ v∉ U')

Theorem 5.8

The instruction del_owner is an authorized secure state transformation.

Proof Use Theorems 4.17, 4.22, 4.23. □

Removal of structure relations requires a complex *pre*-condition, for it must be ensured that all access flow through the structure is not essential. That is, all access rights can be obtained from a second source. In general, this requires a reconstruction of the import-export graph for all objects accessible in regions mentioned and a connectivity check. The check is modeled by the function Notessential sketched below. Simplified cases are discussed in section 5.3.

```
function Notessential(r,s:REGIONS):boolean;
begin
    foreach object o accessible in r or s
    begin
        construct import-export graph
        check connectivity after removing structure relation (r,s)
        if the graph becomes disconnected
            then return false
    end;
    return true
end
```

Definition 5.9

del_struct(u:USERS; r,s:REGIONS)
auth ((u,r)∈ OWN ∨ (u,s)∈OWN)
pre (r,s)∈ STR ∧ Notessential(r,s)
post STR'=STR-{(r,s)}

Theorem 5.9

The instruction del_struct(u,r,s) is an authorized secure state transformation.

Proof Del_struct is a decremental instruction by Definition 4.17. Consistency follows directly from Theorem 4.18. Acceptability is guaranteed by the *pre*-condition, which ensures that no access rights can be transported through the structure relation. Validity is ensured by the function Notessential, which tells that all import-export graphs remain source graphs after removal of the structure. The authorization condition ensures that all affected regions are owned by the invoker.□

Removal of an object is allowed when it is not exported or imported any more. This requires a simple check on the protection state.

Definition 5.10

del_object(u:USERS; r:REGIONS; o:OBJECTS)
auth (u,r)∈ OWN
pre (o,r)∈ DEF ∧ (o,r)∉EXP ∧
¬∃s∈contents(σ,r):(o,s)∈IMP
post O'=O-{o} ∧ DEF'=DEF-{(o,r)}

Theorem 5.10

The instruction del_object(u,r,o) is an authorized secure state transformation.

Proof The instruction is decremental by Definition 4.17. Since conditions 1, 3a,

3b, and 3c of Theorem 4.17 are satisfied by definition, the *post*-condition ensures condition 2 and 3d, and thus the instruction is consistent. The *post*-condition satisfies the condition of Theorem 4.20, making it acceptable. The validity of the instruction follows directly from Theorem 4.24 and the *pre*-condition of the instruction. Authorization is fulfilled, for the activator is owner of the affected regions. □

Undoing export and import instructions requires a complex check on the protection state. Removal is allowed as long as the final state remains valid. However, state validity requires the import-export graph associated with the object to remain a source graph, which, due to the cycles in the import-export graph, can not be decided by a local check of the protection state. A boolean function Remainsvalid is introduced to check for the validity property; the detailed description is left as an exercise.

Definition 5.11

del_import(u:USERS; r:REGIONS; o:OBJECTS)
auth (u,r)∈ OWN
pre (o,r)∈ IMP ∧ Remainsvalid(o,r,IMP)
post IMP'=IMP-{(o,r)}

Theorem 5.11

The instruction del_import(u,r,o) is an authorized secure state transformation.

Proof Del_import is a decremental instruction by Definition 4.17. Use Theorem 4.17 to prove consistency, Theorem 4.19 for acceptability. Validity is enforced by the *pre*-condition, which test the source graph property of the final state. Authorization is guaranteed by the *auth*-condition. □

Definition 5.12

del_export(u:USERS; r:REGIONS; o:OBJECTS)
auth (u,r)∈ OWN
pre (o,r)∈ EXP ∧ Remainsvalid(o,r,EXP)
post EXP'=EXP-{(o,r)}

Theorem 5.12

The instruction del_export is an authorized secure state transformation.

Proof Similar to proof of previous theorem. □

5.2. Well-definedness of the SPE instruction set

In this section we show that the proposed instruction set fulfills the criteria for well-defined SPE instruction sets as introduced in section 4.5.1. First, the instruction set is proved to satisfy the compensation criterion by showing that each incremental instruction is accompanied by a decremental instruction such that a sequential composition of the two is equivalent to a no-op, or identity transformation. Second, the minimality criterion of the instruction set is shown and, finally, we prove that the instruction set is complete with respect to the trivial states.

5.2.1. Compensation criterion

The compensation criterion introduced in section 4.5.3 requires that the instruction set can be split into two categories, based on the definition of incremental and decremental state transformations, such that each incremental instruction can be undone by a sequence of decremental instructions when applied immediately thereafter. Checking this property for the proposed instruction set is straightforward, because the instructions were defined with this property in mind. For each instruction in *IP* we show that the *pre-* and *post-*condition of the incremental instruction imply the *pre*-condition of the corresponding decremental instruction and that the net result of the two instructions applied to a state in sequence is equivalent to applying the identity transformation.

The compensation criterion does not prescribe any policy on the activators of the decremental sequence, yet to effectively undo actions, the decremental instructions should be authorized. If we choose the activator to be the same one as used in the incremental instruction then inspection of the instructions show that all are authorized. Therefore, the authorization problem as it should be addressed in the proof of the next theorem is considered fulfilled and left out.

Theorem 5.13

Let σ be a secure state. Then for each incremental $\phi \in I$ the effects of on σ can be undone by a decremental instruction ϕ' such that

$$\sigma'' = \phi'(\phi(\sigma)) = \sigma$$

Proof Let σ be a secure state. Each incremental instruction is analyzed in turn.

add_region(u,r); del_region(u,r)

Assume that the *pre*-condition of add_region(u,r) holds. Then its effect are that $r \in R_{\phi(\sigma)}$ and $(u,r) \in OWN_{\phi(\sigma)}$, which implies that del_region(u,r) is authorized. Moreover, there does not exists a user $u' \neq u$ with $(u',r) \in OWN_{\phi(\sigma)}$, because del_region is applied immediately. Therefore, the *pre*-condition of del_region holds and the result is that the final state is identical to the state σ.

add_object(u,r,o) ; del_object(u,r,o)
Assume that the *pre*-condition of add_object(u,r,o) holds; then $o \in O_{\phi(\sigma)}$ and $(o,r) \in DEF_{\phi(\sigma)}$. Moreover, del_object when applied immediately thereafter ensures that there does not exists a region r with $(o,r) \in EXP_{\phi(\sigma)}$ nor does there exist a region s such that $(o,s) \in IMP_{\sigma}$. Thus the *pre*-condition holds and the *post*-condition ensures that σ'' equals σ.

add_owner(u,r,v) ; del_owner(u,r,v)
Assume that the *pre*-condition of add_owner(u,r,v) holds; then $v \in U_{\sigma}$ and $(v,r) \in OWN_{\sigma}$. Moreover, the *pre*-condition of add_owner, which ensures that $u \neq v$, and the *post*-condition of add_owner, $(v,r) \in OWN_{\sigma}$, imply the *pre*-condition of del_owner. The *post*-condition of del_owner ensures that $(v,r) \notin OWN$. Furthermore two cases must be distinguished. First, if $v \notin U_{\sigma}$ then v has been added to U by add_owner, but then there does not exist a region $s \in R_{\phi(\sigma)}$ such that $s \neq r$ and $(v,s) \in OWN_{\phi(\sigma)}$, which implies that v is removed from U by del_owner. Second, if $v \in U$ then $\exists s \in R_{\sigma}$ such that $s \neq r$ and $(v,s) \in OWN_{\sigma}$ this implies the last part of the *post*-condition.

add_struct(u,r,s) ; del_struct(u,r,s)
Assume that the *pre*-condition of add_struct(u,r,s) holds; then $(r,s) \in STR_{\phi(\sigma)}$ and $(u,r) \in OWN_{\phi(\sigma)}$ and $(u,s) \in OWN_{\phi(\sigma)}$. Moreover, the function Notessential returns true, for no extra access flow occurs in between, i.e. all access flow remains consistent. Thus, *pre*-condition of del_struct holds and the net result of is $(r,s) \notin STR_{\sigma}$,,.

add_import(u,r,o) ; del_import(u,r,o)
If the *pre*-condition of add_import(u,r,o) holds then $(o,r) \in IMP_{\phi(\sigma)}$. Moreover, the function Remainsvalid returns true, because del_import can not invalidate the source graph property when the access permission is not further distributed. Nor can removal invalidate the source graph property when the object o was accessible in r in the first place. Therefore the result of del_import is $(o,r) \notin IMP_{\sigma}$,,.
Similarly sequence add_export(u,r,o) ; del_export(u,r,o) is analyzed.

Together, all SPE incremental instructions have a counterpart as required in Definition 4.24, thus it satisfies the compensation criterion. □

Corollary For each incremental instruction ϕ there exists precisely one decremental instruction ϕ' such that ϕ' compensates for the effects of ϕ.

5.2.2. Minimality criterion

Minimality of the instruction is best shown by making a table of the effects of the instructions on the state components. In Figure 5.1 we have indicated dependencies between incremental instructions and the changes made to the state components. From this figure we conclude that all instructions affect different state components except for add_owner and add_region, which overlap on the OWN component only. Nevertheless, neither add_region nor add_owner can be replaced by another instruction with the same results in all situations, because they differ in changes made to the basic sets. The decremental instructions have been shown to correspond with their incremental counterparts, since they affect the same state components. This result implies that none of these instructions can be replaced by an equivalent, shorter sequence and thus by definition the SPE instruction set is minimal.

	U	R	O	OWN	STR	DEF	IMP	EXP
add_owner	+			+				
add_region		+		+				
add_object			+			+		
add_struct					+			
add_import							+	
add_export								+

Figure 5.1 Dependencies between instruction and state component

5.2.3. Completeness

An instruction set is said to be complete if we can indicate one or more initial secure states and generate all possible secure states from them. The empty state E can not be used for this purpose, because the SPE authorization policy requires the activator of an instruction to be known in the initial state. This indicates that at least one user must be present and, to get a secure state, this user must be associated with at least one region. States with these properties have been introduced as a trivial state.

Theorem 5.14

The SPE instruction set is complete for T-E

Proof Observe that in each trivial state a user and region exists, which is sufficient to satisfy all *auth*-conditions in the instruction set. Given an arbitrary secure state σ, then a sequence to derive this state from the trivial state σ', with user u and region r, is obtained by the following algorithm.

First, perform add_region transformations for all regions defined within σ with u as invoker. Each of the instructions is authorized and the result is a secure

state. Similar, perform add_owner and add_object for all owners and objects in σ respectively. Subsequently, perform add_struct operations with u as invoker (u owns all regions already). As all import and export relationships are consistent with the structure relation in σ and access paths can be traced back to the object, sequences of import and exports can be generated to achieve the structure defined in σ. Finally, the user u and region r are removed from the result if necessary. As all sets and relationships are finite, this algorithm results in a finite sequence. □

In conclusion, the analysis of the instruction set properties together satisfy the *well-definedness* property as introduced in section 4.5.1.

5.3. Reducing the cost for decremental instructions

A disadvantage of SPE instruction set is that for decremental instructions the cost to guarantee state security require more than a local check on the protection state. During the removal of import/export grants the import-export graph is checked for source graph property. Removing structure relations requires a series of such checks, namely for all objects accessible within the regions named.

The cause of this asymmetric behavior stems from the requirement to keep the protection state secure at all times and the side effects by not specifying the source of grants. Still the cost is limited to the size of the source graph, because removal of a given edge in the import-export graph requires at most all edges to be inspected. The special case that an edge is the end of all access paths, can be checked locally.

To avoid the costly checks on the applicability of decremental instructions, one might conceive an instruction set with local checks only. For example, the instruction del_struct, del_import and del_export defined below all require a local check. Both are secure state transformations, but the modified instruction set is not *well-defined*, as shown shortly.

Definition 5.13

del_struct(u:USERS; r,s:REGIONS)
auth ((u,r)∈ OWN ∨ (u,s)∈OWN)
pre (r,s)∈ STR ∧ ¬∃o:((o,r)∈EXP ∨ (o,r)∈IMP)
post STR'=STR-{(r,s)}

Theorem 5.15

The instruction del_struct(u,r,s) is an authorized secure state transformation.

Proof Similar to 5.9 using Theorem 4.27.□

Definition 5.14

del_import(u:USERS; r:REGIONS; o:OBJECTS)

auth (u,r)∈ OWN

pre (o,r)∈ IMP ∧ ¬∃o':(o',r)∈ EXP ∧
¬∃s: ((s,r)∈STR ∧ (o,s)∈$IMP_{\phi(\sigma)}$)

post IMP'=IMP-{(o,r)}

Theorem 5.16

The instruction del_import(u,r,s) is an authorized secure state transformation.

Proof Similar to 5.14 using Theorem 4.29.□

Definition 5.15

del_export(u:USERS; r:REGIONS; o:OBJECTS)

auth (u,r)∈ OWN

pre (o,r)∈ EXP ∧ ¬(∃s∈ environment(σ,r):(o,s)∈EXP ∨
∃t∈contents(s):(o,s)∈$IMP_{\phi(\sigma)}$)

post EXP'=EXP-{(o,r)}

Theorem 5.17

The instruction del_export(u,r,s) is an authorized secure state transformation.

Proof Similar to 5.15 using Theorem 4.30.□

The disadvantage of the modified instruction set is that it blocks the removal of regions and grants; and therefore does not satisfy the compensation criterion of *well-defined* instruction sets. To illustrate this, Figure 5.2 shows the protection state before and after the action add_import(u,r,o). Although each effect can be nullified, it can not be done without a mixture of decremental and incremental instructions. In this particular case, first the import into s has to be removed, second del_import(u,r,o) is applicable, and finally the old situation is restored by execution of add_import(u,s,o).

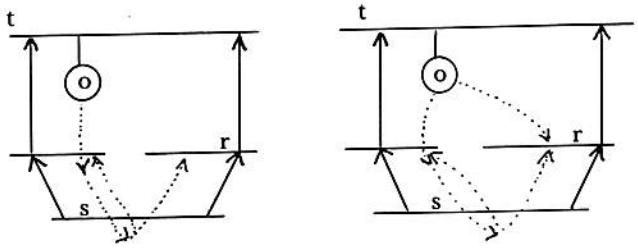


Figure 5.2 Side-effects of importation.

5.4. Revocation

The definition of the SPE model in terms of its secure states and the SPE instruction set allows for the analysis of security issues at a higher level of abstraction. In particular, the SPE instruction set concentrates on security properties of at most two instructions at a time. A new class of security questions arises when longer series are considered and when algorithms are developed which generate an instruction series to achieve a certain (protection) goal.

In this section we address the problem of finding series with the property that some action 'performed in the past' is canceled or compensated for, i.e. the *revocation* problem. A revocation scheme associated with a protection system prescribes (and enforces) a policy for the removal of access permissions, as well as the removal of users and objects administered. A revocation scheme, like an authorization scheme, is characteristic of a protection system. The choice not only affects the security properties, but also the ways to implement the protection system.

The basis for any revocation scheme in SPE is provided by the compensation property of the instruction set, for each incremental instruction can be undone immediately by its associated decremental counterpart. In general, however, an incremental instruction is followed by many state-modifying instructions before it is considered for revocation. Then, application of its decremental counterpart is not guaranteed to succeed. Moreover, its application may breach the security policy decisions made by the users of the system. For example, when a protection system protects the definition of objects and the transfer of its access permissions only, something is amiss if any user could freely remove them using a revocation primitive.

In the following sections we address different issues related to revocation schemes. First, the notion of a protection state history is defined to obtain more formal definitions of revocation in the context of the SPE model. Next, the semantic limitations of a revocation scheme are discussed in terms of activators and security semantics embodied by the protection state as perceived by the user. Example revocation schemes from the operating system and database area are presented. Finally, two specific schemes are presented and illustrated by an revocation algorithm.

5.4.1. The history of a protection state

Revocation deals with undoing actions applied to a protection state in the past, which indicates the need for the notion of a protection state history. The history information is used to deduce what actions can possibly be revoked and what time relations exist between actions. In turn, this information is needed to guarantee state security invariants during and after the revocation act.

The occurrence of failed instructions, due to invalid *auth-* and *pre*-condition, is normally administered in the history as well to implement security threat

monitoring. This information won't be used in the process of revocation, because of there null effect on the protection state. Moreover, as illustrated shortly, security is easily violated when unsuccessful actions from the past are executed by the revocation process. A formal definition of protection state history in terms of SPE concepts becomes:

Definition 5.16

The action $i \in I$ succeeds, denoted by *Succeeds*(σ,i) if its *auth-* and *pre-*condition holds for the state σ.

Definition 5.17

A history of the state σ', denoted by $H(\sigma,I)$, is the series of instructions $I=i_n \ldots i_1$ applied to the state σ such that (for i=1..n)

$$\sigma_i = i_i(\sigma_{i-1}) \text{ and } \sigma = \sigma_0 \text{ and } \sigma' = \sigma_n \text{ and } Succeeds(\sigma_i, i_{i-1})$$

The history of a state is a list of instructions as applied to a given initial state. For each instruction, the history list includes the actual parameters, i.e. the name of activator, the object names, and the region names involved, which is emphasized by the convention in this context to use i rather than ϕ. Notational convention: $I(\sigma)= i_n\, i_{n-1} \ldots i_1\, (\sigma)$.

5.4.2. A definition of revocation

A recurring definition for revoking an action i is to rebuild the protection state such as it would have been if the instruction i was not executed in the first place. This means that if an action is revoked all dependent actions should be revoked as well. Application of this approach to the SPE model results in the following definition:

Definition 5.18

Let $H(\sigma,I)$ be the history of the state σ'. Then *revocation* of $i_j \in I$ by a user u, denoted by revoke(u,i_j), is the application of a sequence $I'=i_k \ldots i_1$ to the state σ' such that

$$I'(\sigma') = i_k \ldots i_1\, (\sigma') = i_n \ldots i_{j+1}\, i_{j-1} \ldots i_0(\sigma)$$

The user u is called the revoker and the sequence I' is called the revocation sequence. In leaving out i_j, some of the successful instructions i_m (m=i+1..n) may become unsuccessful, because the *auth-* and *pre-*condition depends on the effects of the action revoked and should be revoked as well.
Note that revocation of unsuccessful actions is covered by this definition as well, provided they are represented in the history as identity state transformations, which properly describes their effects on the protection state.

The revocation schemes presented in the literature often ignore the existence and the role of the revocation sequence. Instead, revocation is seen as a simple atomic action to rebuild the state [Wood79, Griffiths76, Tanenbaum81]. Moreover, a revocation sequence provides useful information for a user considering a revoke action. Instead of being forced to analyze the protection

state and the required state to obtain the differences, inspection of the revocation sequence shows all actions potentially affected by the revoke action.

5.4.3. The revocation sequence

The revocation sequence properties in Definition 5.18 were explicitly left undefined. Yet protection systems addressing the concept of revocation should spell out the revocation policy to avoid semantic problems and bring the protection system in line with the organizational policies. A revocation policy, like an authorization policy, consists of a number of rules which describes the conditions under which an action can be revoked and what the consequences are in terms of changes made to the protection state. In particular, a revocation policy addresses the issues; 'Who is allowed to revoke an action ?' and 'How are the effects of the revoked action undone ?'

One possible revocation policy is to limit the sequence to decremental actions. Unfortunately such a simple policy does not work in reality, because an action and its consequences are mostly unrevocable. The best one can hope for is to be able to compensate for the effects of the action being revoked. A simple example illustrates this phenomenon.

Let the director of a bank grant a branch manager access to a class of accounts. The branch manager in turn grants some of his accessible accounts to his clerks. When the branch manager is found fraudulent the bank director will revoke the rights of the branch manager. In general, however, the access permissions of the clerks should not be revoked (they depend on the rights of the branch manager) to avoid disruption of client handling. The grants of the clerks should be considered obtained from the new branch manager instead.

Therefore, we propose the following less stringent revocation policy: the complementary action of the instruction being revoked is executed successfully once. In terms of the example, it is possible to remove the grants given to the branch manager without disrupting the rest of the system. The solution in the example above would be, just like SPE, to make the new branch manager co-owner of the rights on the branch first, after which the rights may be stripped from the fraudulent manager without making the protection state unsafe.

The resulting protection policy has a drawback. It allows revocation sequences changing more of the protection state than strictly necessary. For example, it is undesirable that the new branch manager can, using the revocation situation of the protection system introduce his nephew as a clerk. Therefore, a general and more realistic definition of a revocation becomes:

Definition 5.19

Let $H(\sigma,I)$ be the history of the state σ'. Then *weak revocation* of $i \in I$ by user u, revoke(u,i), is the application of a successful revocation sequence $I'=i'_m \ldots i'_1$ to the state σ' such that

$$Succeeds(I'(\sigma),i^{-1})$$

and I' is of minimal length and includes the compensation action for i, denoted by i^{-1}.

5.4.4. Limitations of revocation

Definitions 5.18 and 5.19 allow any action to be revoked. This generality is unnecessary and results in severe (semantic and implementation) problems, which are illustrated for SPE using revocation with side-effects and revocation of decremental instructions. However, similar remarks can be made for other protection systems as well.

As indicated before, unsuccessful actions when included in the history can not be revoked. The only reasonable instruction to be considered in such a situation as the revocation sequence would be an identity state transformation, because in SPE an erroneous instruction does not change the state either. Generating a revocation sequence based on the parameters of the unsuccessful action alone would lead to undesirable behavior.

For example, consider the historic act add_object(u,r,o) and that it is recorded as unsuccessful. Then the revocation sequence should not contain the action del_object(u,r,o) or add_object(u,r,o). The former is senseless, because it is bound to fail. The latter results in the undesirable situation that a user is confronted with the creation of an object long forgotten or compensated for by other (successful) actions.

A more serious problem occurs when decremental instructions are considered for revocation. Allowing these instructions to be revoked easily leads to violations of protection policies imposed by users (but not formalized in SPE), since the protection policy used by users may include decisions based on the protection state at the moment of decision making. If a compensation action were to be revoked then the user should incorporate both the past and the future of the state in his decision making, because at any time an action of the past may be revoked and violate his policy, as may each action in the future.

For example, consider two regions r and s, owned by users u and v, respectively and the following actions to establish a communication path:

add_owner(u,r,v)
add_struct(v,s,r)
del_owner(u,r,v)

Each of these actions is an authorized state transformation and from this point on, user v can share objects with u and vice versa. Now, assume that u terminates this 'contract' and applies

del_struct(u,s,r)

If later on v exports an object to its environment it has to include the possibility that the action del_struct(u,s,r) might be revoked in the future. Clearly an unworkable situation. If u has given up his rights on exported objects from v, he should first negotiate for a new contract, rather than reactivate the old contract. Another example is the revocation of deleted objects, which would require the protection system to keep deleted objects for an indefinite time.

In conclusion, any revocation policy which allows for the revocation of unsuccessful or decremental instructions is bound to lead to semantic problems. Therefore, we restrict the definition of SPE revocation policies to the successful incremental SPE instructions included in the history of a protection state.

5.4.5. The role of activators

Let $H(\sigma,I)$ be the history of σ' and $I' = i_n \ldots i_1$ be a revocation sequence for a successful incremental SPE instruction i in I. Then, for n=1, the revocation sequence consists of a single decremental instruction, the compensation instruction for i in SPE. This holds for all protection systems with the compensation property for their instruction set. However, for n>1 problems arise.

For example, consider the following sequence of authorized SPE instructions.

add_object(u,r,o) (1)
...
add_export(u,r,o) (2)
...
add_export(u',s,o) (3)
...

Let revoke(u,(1)) be applied to this state, then a natural revocation sequence becomes:

del_export(u',s,o) (4)
del_export(u,r,o) (5)
del_object(u,r,o) (6)

In other words, the object o is removed after all grants relating to it are removed first. This sequence, however, includes a number of assumptions. It assumes that both u and u' exist when revoking (1), that they will cooperate with v, and that indeed (2) and (3) should be revoked as well. The latter is easy, for not removing (2) nor (3) would result in an unsafe (invalid) protection state.

The SPE instruction revoke(u,i) like all protection system instructions, has an activator (u) and should be authorized before it takes effect. The authorization policy for SPE requires u to be a user represented in the protection state and owner of the affected regions. However, this rule is neither sufficient nor desirable as a basis for a revocation policy. First, assume that it would be sufficient to be a known user. Then each user could revoke all actions, clearly an

undesirable, insecure system. Second, assume that we would require that the revoker owns all affected regions. Then a user could prevent revocation by removing the potential revoker as co-owner from his regions.

An alternative revocation policy is to allow the original activator to be the revoker or allow this to those users having the same "power" as the original activator. In the example above u and u' would be the potential revokers, user u is authorized to revoke (1) and (2), u' to revoke (3). If v were a co-owner of u in region r then this policy would consider them equally powerful and v would be allowed to revoke the actions as well.

A rule exploited in access control list organizations is to restrict all revocation actions to the one object owner. Such a scheme will be referred to as an object-oriented approach and is relevant for protection systems neglecting the flow of access permissions.

Revoke(u,(1)) illustrates a domino-effect, where all actions solely based on the revoked action are revoked too, i.e. revoke(u,(2)) and revoke(u',(3)) are triggered. This means that the protection system both generates the requests and selects the proper activators, u and u' respectively. A more democratic revocation rule is to execute a revoke when all actions generated under a domino-effect policy are revoked first. This prohibits revocation of (1) until u' has revoked (3). Unfortunately, such a democracy does not work, because then any access right obtained by a user can be safeguarded against removal through isolation, i.e. removal of all co-owners.

The last assumption hidden in the example above is that both u and u' exist when revocation is triggered. That user u exists is evident, otherwise revoke(u,(6)) is not authorized, but u' should exist also, otherwise the protection state contained inconsistent information.

In conclusion, a revocation policy can be seen as an extension of the authorization policy aimed at regulating the undoing of state changes. Shortly an authorization policy for the SPE system is defined and supplied with algorithms to enforce the policy in an implementation of the model.

5.4.6. Revocation algorithm classification

The two revocation policies presented later on in this chapter for the SPE model differ considerably in the approach taken to represent a state and form of revocation algorithm. To place the approaches in a broader perspective we illustrate the possibilities of protection state representation and algorithmic revocation strategies first.

Representations of protection states can roughly be divided into two classes: a chronological approach and a state oriented approach. A chronological representation is directly based on the history of the protection state and consists of a time stamped administration of the actions. These time stamps enable the user to derive relevant parts of the history for analysis and revocation. Two sub-

categories of this class are the space conservative and the pure representation. In a space conservative representation the history of the protection state is compressed to include successful and non-revoked actions only. Under a purely chronological approach all actions, successful/ unsuccessful and revoked/ non-revoked actions are administered. This approach lends itself not only to the enforcement of the protection policies, but also to realize of effective threat monitoring. Unfortunately, no efficient realization of a purely chronological representation is known, because of the excessive amount of storage involved.

The state oriented approach ignores the time aspect of the actions involved. Thus, given the protection state, limited information is available on the sequence of actions which took place and no threat monitoring is possible. Most operating system protection systems use this scheme. Any state oriented approach is faced with what has become known as the frame problem in the AI area. That is, it is difficult to infer the actions responsible for changes in successive states. This results in a costly analysis of a revocation sequence and thus severely restricts the class of policies to be considered.

To alleviate the shortcomings of both classes a hybrid system is mostly used. A state description is used to authorize new requests, while the history (transaction log) is used to generate the revocation sequence.

Both major categories of state representations lend themselves to the same kinds of revocation algorithms once a revocation policy is fixed. A possible categorization is: simulation based, goal based, and compensation based algorithms.

A simulation based revocation algorithm for a chronological state representation starts from the initial state and re-evaluates the actions, leaving out the action revoked to arrive at the new protection state, i.e. a modified action list. The final list can be used to replace the current protection state or the simulation shows the actions required to arrive at a state satisfying a constraint revocation policy. When a state oriented approach is used, the revocation procedure requires a simple rule to generate the sequence to avoid the frame problem. Either the revocation action is obvious or it can be deduced by generating a few potential new states which are checked for compliance with the revocation policy. The prime advantage of a simulation based algorithm is the ease with which the new state is derived or a revocation sequence is generated. A disadvantage, though, is that it may be time-consuming and requires full control over the protection state representation.

Under a goal based revocation algorithm no simulation of the protection state is used, the algorithm tries to find the shortest sequence to successfully revoke the instruction i instead. It takes the current state, the *pre*-condition of the compensation instruction for i, and the additional revocation rules and decides which actions are necessary to bring a successful execution of i^{-1} closer. For a chronological representation scheme it implies that the action list is traversed backwards. This algorithm does not require full control of the state. Local

decisions are sufficient to bring the goal closer, provided that during the revocation process no concurrent actions are issued which postpone reaching the goal indefinitely.

The last algorithm differs from the previous two in adding information rather than removing or changing the protection information. We will call it the compensation approach. It enters information to the state which declares information as outdated. Either the information that some action is revoked is stored or a set of actions is generated to mark all relevant parts as being revoked. Again, such an approach in a chronological state representation is relevant for threat monitoring, because at any time the complete history of the protection system is available. Note that representations which mark the object as being deleted without marking all derived access rights as such leads to a protection system which is not locally secure. It means that no local check suffices to determine accessibility.

5.4.7. Example revocation policies

Operating system revocation.

A well known architecture for secure operating systems is MULTICS [Organick72] This system uses an access control list to administer the users' access permission regarding the objects stored. The revocation policy permits the one owner of the object to remove any entry from this list. No additional dependencies are kept for the propagation of access rights and thus no domino-effect occurs. MULTICS uses a state oriented representation of the protection state and its revocation algorithm falls within the class of goal-seeking approaches, which is simplified by the straightforward selection of the compensation action.

An exponent of the capability-based approaches to secure operating system construction is the AMOEBA [Tanenbaum81] distributed system. In this system access rights are represented by encrypted objects, i.e. a state oriented representation. The revocation policy restricts revocation to those users having implicitly or explicitly obtained revocation permission. A capability is declared invalid and is administered as such by the object managers, i.e. a compensation algorithm is used. The scheme is not space-conservative nor locally secure, because it permits invalid capabilities to remain in existence, while a new capability to represent access to the object prohibits their use.

Database system revocation

An example space-conservative chronological protection state description is the model proposed by Wood-Summers-Fernandez [Wood79]. A short description of which is given in section 2.5.3.3. The following revocation policy is used. Whenever Run grant permission on the object, i.e. the right to distributed Run permission to others, is delegated to a subject, the grantor receives the right to revoke this grant and looses the right to issue any further Run grant permission on the object. The protection state of this model can be represented as a hierarchy; granting a right can be considered as introducing a node, with the right to revoke the grant represented by an arc between grantor and grantee. In fact, this hierarchy describes the dependencies among the actions.
In this approach, the goal-seeking revocation policy is implemented by a set of access control decisions, i.e. the revocation right is exercised. Revoke includes a domino effect, for all the access rights associated with the revoked right are passed to the revoker.
For example, assume that grants are transmitted as depicted in Figure 5.3, where u grants a right to the user v. This right is further granted by v to user w. Revocation of the action issued by u results in the removal of the edge (u,v) representing the grant action and rewriting the hierarchy such that all actions issued by v are associated with u, the grantor of v, instead.

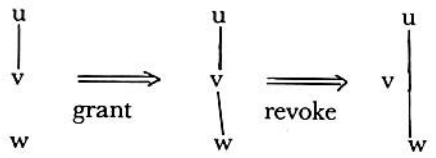


Figure 5.3 WSF granting scheme

Another chronological scheme is the authorization model defined by Griffiths-Wade [Griffiths76] and the refinement by Fagin [Fagin77]. In the Griffiths-Wade model the actions are time-stamped and kept in a list, which effectively represents the history of the protection state. Each grantor can revoke the privileges granted and revocation has a domino-effect, so that all privileges (exclusively) based on the action revoked are undone. The disadvantages of the goal based revocation algorithm are overcome in this model by storing the protection state in a database and using the transaction notion to ensure atomicity of the revocation analysis.

Programming language based revocation

A form of revocation can be found in most (algorithmic) languages as well. The symbol table of a language compiler functions as a protection state representation, the authorization and revocation policy being described by the scope rules. For example, in a language like Pascal all local object declarations are revoked when a procedure returns, an example of a goal directed revocation algorithm. However, the protection state representation of dynamic objects is handled by the programmer in his data structures. Notice that only one user is involved at the program definition level, but that each procedure invocation can be seen as an active entity during program execution. In fact, each procedure might run on a different machine with different operating system security regulations.

5.4.8. SPE revocation policies

In the definition of revocation algorithms for the SPE model we are restricted to simulation and goal oriented algorithms, because there is no primitive concept in the model which can be used to administer compensated actions. In addition, a simulation approach requires, in addition to the protection state representation, a chronological representation scheme. The goal oriented approach can be based on the set representation.

Revocation limitations and implementation examples described before provide the background for two revocation policies for the SPE model and instruction set. First, revocation is provided by an extension of the SPE primitive set with the instruction revoke(u, action) which implements the revocation policy of Definition 5.18. This policy is illustrated using a hybrid state representation in combination with a simulation algorithm, which highlights commutative aspects of the SPE instruction set. Second, an example goal oriented policy is defined, where only the ability to execute the compensation action at some derivable state is required. That is, Definition 5.19, a weak revocation scheme, is implemented. An algorithm is given and its difference with the former approach is illustrated.

5.4.8.1. SPE chronological revocation

A chronological revocation scheme for SPE requires an inspection of the information available from the state history, which should be maintained separately in a list. First, notice that each action represented in the state history is either successful or unsuccessful, and revoked or non-revoked. The examples given above suffice to restrict revocation to the successful actions. Second, the SPE instruction set satisfies the compensation criterion, i.e. the decremental instructions undo the incremental instructions. Thus, a decremental instruction applied to a state can be considered as a revocation sequence of length one.

Conversely, this implies that whenever an action is considered for revocation the history description should not already include such a revocation action. To ease the recognition of these dependencies a predicate is introduced.

Definition 5.20

Let $H(\sigma,I)$ be the history of the state σ'. The predicate *Isrevoked*$(H(\sigma,I),i_j)$ is true if i_j is revoked by an action which succeeds in I.

A revocation policy can now be defined for SPE as follows:

Definition 5.21

Let $H(\sigma,I)$ be the history of the state σ'. An action i_j in I can be revoked by the instruction revoke(u,i_j), if

1) $i_j \in I = i_m \ldots i_1$ and
2) i_j is an incremental instruction and
3) *Succeeds*(σ_{j-1}, I_j) and
4) not *Isrevoked*$(H(\sigma,I),i_j)$ and
5) u is an owner of the regions affected by i_j^{-1}

by a revocation sequence Γ such that

a) $\Gamma(I(\sigma)) = i_k \ldots i_{m+1} I(\sigma)\ i_m \ldots i_{j+1} i_{j-1} \ldots i_0(\sigma)$

b) $\forall i \in \{1..m\}$ not *Succeeds*$(H(\sigma,I),i) \rightarrow$ not *Succeeds*$(H(\sigma,\Gamma I),i)$

c) $\forall i \in \{1..m\}$ *Isrevoked*$(H(\sigma,I),I) \rightarrow$ *Isrevoked*$(H(\sigma,\Gamma I),i)$

Condition a) enforces the revocation policy of Definition 5.18, while b) and c) take care of the semantic problems described before by guaranteeing that revocation does not change the status of non-successful and revoked actions.
This policy allows all owners of the affected regions implied by the compensation action to revoke the corresponding action. This means that an object owner can always remove the objects and its access rights, while other users can always revoke their import/export grants. This relaxation on the activator identity solves problems arising from the removal of users from a state or retracting (co-)ownership.

A straightforward implementation of this policy, assuming a hybrid state description, is to simulate the behavior of the protection state by re-execution of the state history and replacing the current state by the simulated state and replacing the history list by the modified list, which is extended with the revocation list, in the end. Although simulation certainly results in a secure state, it should be shown that the revocation sequence appended to the history list satisfies the constraint of Definition 5.18, i.e. the revocation sequence can indeed be applied to the state and leads to a secure state with the desired properties. Analysis of the revocation sequence generated by the simulation algorithm not only answers this question, but provides useful insight in the commutative aspects of SPE sequences.
A sketch of algorithm Algorithm-H is given in Figure 5.4. The protection system is represented by three global variables, the initial state, the list of actions applied, and the current state. The revocation procedure accepts the name of

an activator and an index in the history of the action to be revoked. First, the current state used to authorize the revoke action, i.e. does the user own the affected regions and is the action revocable. Second, the intermediate state is reconstructed up to the action to be revoked. Third, the compensation action is saved in the revocation stack and all subsequent actions are evaluated. Finally, the revocation sequence is appended to the history. This process results in a new state and a revocation sequence. The new state can replace the current state when we can prove that the obtained state satisfies the revocation definition.

```
{ This program is not supposed to compile directly }

type Operators = (add_region,{...},del_export);
        Action=record
                instruction:Instructions;
                activator:SPEname;
                region,parm:SPEname;
                success, isrevoked: boolean;
        end; History = array[1..infinite] of Action;
        State  = SPE_state_description;

var  h: History;              { history of the current state }
     hlim:integer;      { length history }
     istate,           { initial state }
     cstate,           { current state }
     s: State;         { temporary state }
     rs : Sequence;     { revocation sequence }
     rslim:integer;     { length of revocation sequence }

function authorized(s:State; a:Action):boolean;
begin
{ returns true when the action a is authorized for state s }
end;

function applicable(a:Action):boolean;
begin
        applicable:= not a.isrevoked and a.successful
end;

function owner(s:State; region:SPEname): SPEname;
begin { return name of a region (co-)owner } end;

function complement(op:Operators):Operators;
begin { as expected } end;

procedure compensate(act:Action; var newaction:Action);
```

```
begin
        newaction:= act;
        newaction.activator:= owner(s,act.region);
        newaction.instruction:= complement(act.instruction)
end;

procedure transform(act:Action; var state:State);
begin
        case act.instruction of
          { change the state accordingly }
        end
end;

procedure pushaction(a:Action);
begin
        compensate(a, rs[rslim]);
        rslim:= rslim+1
end;

procedure revoke(u:user; actnr:integer);
var i:integer; ca:Action;
begin
        compensate(cstate,h[actnr],ca);
        ca.activator:= u;
        if authorized(cstate,ca) and applicable(ca) then
        begin
                s:= initial;    { construct intermediate state }
                for i=1 to actnr-1 do transform(h[actnr], s);

                rs[0]:= ca; rslim:=1;
                hist[actnr].isrevoked:= true;

                for i:=actnr+1 to hlim do
                if applicable(h[i]) and authorized(s,h[i])
                        then transform(h[i],s)
                        else begin
                                compensate(s,h[i],ca)
                                pushaction(ca)
                        end
        end;
        for i:= hlim+1 to hlim+rslim do h[i]:= rs[i-hlim-1]
end
```

Figure 5.4 Algorithm-H, a history based revocation algorithm.

5.4.8.2. Characteristics of revocation sequences

Algorithm-H generates a sequence I' which satisfies Definition 5.18 as a revocation sequence to revoke(u,i_j) when the revocation is authorized. This claim is shown in a few steps. First, assume that i_j is the last, successful, incremental, and non-revoked authorized action in the sequence I, i.e. $\sigma'=i_jI''(\sigma)$. Then inspection of the algorithm shows it to generate a single compensation action by applying the routine compensate once. The SPE instruction set guarantees that $i_j^1\ i_j\ I''(\sigma)$ succeeds and that it is equivalent to the identity transformation, or no-op transformation, and the final state $\sigma''=\sigma$ satisfies the policy set forth in Definition 5.20 and Definition 5.21

If i_j is not the last action, but is followed by a sequence of one more actions, say Im, it is not immediately clear that

$$i_j^1\ Im\ i_j\ I''(\sigma) \quad (1)$$

succeeds, except when all actions in Im are unsuccessful (or revoked) i.e. they are equivalent to a no-op. Assume that algorithm-H generates precisely one action. Then necessarily all actions in Im are independent of i_j, because otherwise the algorithm would have failed to generate a simulated state with one action in the revoke sequence. To show that it implies that i_j^1 can be applied to the state successfully with the results of 5.21, we should prove either that (1) is equivalent to

$$Im\ i_j^1\ i_j\ I''(\sigma) \quad (2)$$

or that it is equivalent to the sequence

$$i_j^1\ i_j\ Im\ I''(\sigma) \quad (3)$$

Unfortunately, the latter can not be proved for all cases, as illustrated by the following counterexample. Let the history be composed of two instructions only,

del_owner(v,r,u) ; add_struct(u,r,s) (σ)

and assume that both instructions succeed. Add_struct(u,r,s) can be revoked by user v and algorithm-H generates the compensation action del_struct(v,r,s). The simulation process concludes that del_owner can be successfully applied to σ and does not depend on the effects of add_struct(u,r,s), giving the sequences:

del_struct(v,r,s); del_owner(v,r,u); add_struct(u,r,s) (σ) (4)

del_struct(u,r,s); add_struct(u,r,s); del_owner(v,r,u) (σ) (5)

However, sequence (4) is not equivalent to (5), because the *pre*-condition of add_struct is no longer satisfied after applying del_owner. Sequence (4), obtained from (2), succeeds, which shows that apparently del_struct and del_owner commute. Property (2) for a single action revocation sequence is proved by induction on the number of actions in Im and the SPE instruction set.

Theorem 5.18

Let H(σ,*I*) be the history of the state σ' with $I=i_m\ i_{m-1}$. If algorithm-H generates a single action i_{m-1}^{-1} to revoke i_{m-1} then

Succeeds(H(σ,*I*), i_{m-1}^{-1}) and
$i_{m-1}^{-1}\ i_m\ i_{m-1}\ (\sigma)= i_m\ i_{m-1}^{-1}\ i_{m-1}\ (\sigma)= i_m\ (\sigma)$

for all SPE primitive operations i_m.

Proof The existence of a single revocation action means that i_m can be applied successfully to the sequence $i_m\ i_{m-1}^{-1}\ i_{m-1}\ (\sigma)= i_m\ (\sigma)= \sigma'$. This result leaves us to show that $i_{m-1}^{-1}\ i_m = i_m\ i_{m-1}^{-1}$ for all SPE instructions i_m. This means that we have to prove that the *post*-condition of i_m does not invalidate the *pre*-condition nor the authorization constraint of i_{m-1}^{-1}.

The authorization condition of i_{m-1}^{-1} is invalidated by i_m by removing an activator or an affected region only. Assume that i_m is a del_owner(v,r,u) operation removing the activator u from an affected region. Then algorithm-H would not authorize the revoke action due to violation of constraint 4) in definition 5.21 and thus the action generated can not be this del_owner operation. When i_m removes an affected region, for example del_region(v,r) is applied, we know that its successful application implies that r was not used for the definition of structure, objects, import, and exports, nor that it was owned by other users, but algorithm-H has generated an action, so r can not be an affected region of i_{m-1} and thus can not invalidate the authorization of i_{m-1}^{-1}.

To prove that the *post*-condition of i_m does not invalidate the *pre*-condition of i_{m-1}^{-1} we should consider six cases only, because i_{m-1} can be undone by a decremental instruction only.

Let i_{m-1}^{-1} be a del_owner(u,r,v) operation compensating add_owner(u,r,v). Then the *pre*-condition of del_owner fails if u is removed by i_m such that v is the sole remaining owner of r. Then, however, i_m would also have failed during the simulation as well and a longer sequence would have been generated.

Let i_{m-1}^{-1} be the operation del_region(u,r) compensating add_region(u,r). Then its *pre*-condition fails if either i_m introduces a co-owner of r, or uses the region in the definition of an object or a structure relation, or makes r the last region owned by u. However, the simulation process in algorithm-H ensures that i_m was successful against the state obtained by leaving out i_{m-1}. This means that i_m can not affect region r at all.

Let i_{m-1}^{-1} be the operation del_object(u,r,o) compensating add_object(u,r,o).

Then i_m invalidates its *pre*-condition if it removes the object from the state, which is precluded by the algorithm (non-revoked), or applies an export or import operation to the object. The latter, however, would fail against the state obtained by leaving out the add_object instruction.

Let i_{m-1}^{-1} be the operation del_struct(u,r,s) compensating add_struct(u,r,s). Then i_m invalidates the *pre*-condition if (r,s) forms an essential link used by i_m in the transfer of access. However, the algorithm ensures that any transfer of access could be done without the existence of the structure relation.

Let i_{m-1}^{-1} be the instruction del_import(u,r,o) compensating add_import(u,r,o). Then it is invalidated when either the object is removed or it is not accessible in r any more. First, the object can not be removed by i_m, because the instruction del_object requires all imports to be revoked first. Second, the only instruction to remove accessibility of the object from r is del_import(u,r,o), which would mean that an empty sequence is generated by algorithm-H. A similar argument holds for del_import. □

This theorem can be applied directly to all sequences where a single revocation action is generated.

Theorem 5.19

Let H(σ,*I*) be the history of the state σ' with $I=Pi$ ($P=i_m...i_1$). If Algorithm-H generates a single action i^{-1} to revoke i_j then

$$i^{-1}\ P\ i\ (\sigma) = P\ i^{-1}\ i\ (\sigma) = P\ (\sigma)$$

Proof The proof is given by induction on m, the number of instructions separating i^{-1} from i. For m=1 the previous theorem provides the result.
Assume that the theorem holds for m=k. Then we should prove the property for m=k+1, that is,

$$i^{-1}\ i_{k+1}\ i_k...i_1\ i(\sigma) = \quad (1)$$

$$i_{k+1}\ i^{-1}\ i_k...i_1 i(\sigma) = \quad (2)$$

$$i_{k+1}\ i_k...i_1\ i^{-1}\ i\ (\sigma) = \quad (3)$$

$$i_{k+1}\ i_k...i_1(\sigma) = \quad (4)$$

To go from step (2) to (3) follows from the induction hypothesis. This means that the authorization condition nor the pre-condition of i^{-1} is invalidated by the sequence $i_k\ i\ ...\ i_1$. With these properties in mind we can apply arguments similar to those of the previous theorem.

First, assume that the authorization constraint of i^{-1} is invalidated by i_{k+1}, which means that either the activator and/or affected region is removed. Removing

the activator from an affected region would make the revoke action unauthorized by constraint 4) of definition 5.21, which is not the case, because a compensation action is generated. Removal of an affected region can not invalidate i^{-1} either, because in that case it was not used for any other purpose at all.

Proving the non-interference of i_{k+1} with the *pre*-condition is shown similarly as in the previous theorem, one illustrative case will be shown only.

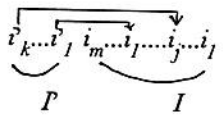
Let i^{-1} be the operation del_region(u,r) compensating add_region(u,r). Then its *pre*-condition fails if either i_{k+1} introduces other users to r, or uses the region for the definition of objects or structure relations, or makes r the last region owned by u only. The induction hypothesis gives that $i_k...i_1$ does not interfere with the execution of i_{-1}. Moreover, the simulation in algorithm-H ensures that i_{k+1} is successful against the state leaving out i_j as well. Combination with the knowledge that i_j was successful and not revoked, which means that i_{k+1} does not affect region r at all. Otherwise it would have failed in the simulation.□

Theorem 5.20

Let H(σ,*I*) be the history of the state σ'. Then Algorithm-H generates a revocation sequence $I'=i'_k...i'_1$, such that

$$\mathit{Succeeds}(\mathrm{H}(\sigma,I),I') \text{ and } I'(\sigma')=i'_k...i'_1(\sigma')\ i_m...i_{j+1}\ i_{j-1}...i_0(\sigma)$$

Proof The proof is based on two observations. First, the simulation of Algorithm-H ensures that all the actions depending solely on the action revoked at the time of interpretation are handled by the routine compensate, which generates a compensation action and pushes it onto the revocation stack. This results in the following sequence



applied to the state σ, where the arrows indicate which actions are compensated.

Second, this sequence is successfully applied to the state σ by repeated application of the previous lemma, starting with i'_1. Algorithm-H ensures that the action compensated by i'_1 is not followed by an action which could invalidate its authorization or *pre*-condition. Thus, the previous lemma can be applied to obtain the sequence: $i'_k...i'_2$.□

5.4.8.3. SPE goal-seeking revocation

A disadvantage algorithm-H is that it requires full control of the protection state during revocation and rebuilds the intermediate states. This may become a bottleneck when an extensive history is kept or when a distributed system with the protection state partitioned over the nodes is considered. A goal-seeking revocation algorithm may provide a solution to these problems.

A goal-seeking revocation algorithm generates one revocation action at a time and applies it to the state. The algorithm stops when it is able to apply the compensation instruction for the action being revoked. Moreover, with the assumption of no interference (exclusive access to the protection state) the sequence should be minimal in terms of reaching its goal. Such an approach implements a weak revocation scheme (Definition 5.19). A formal definition of the goal oriented revocation scheme becomes:

Definition 5.22

Let H(σ,I) be the history of the state σ'. An action i_j in I is revoked by the instruction revoke(u,i), if

1) $i_j \in I = i_m .. i_1$ and
2) i_j is an incremental instruction and
3) *Succeeds*(σ_{j-1},I_j) and
4) not *Isrevoked*(H(σ,I),i_j) and
5) u is an owner of the regions affected by i_j^1

then a minimal revocation sequence I' is generated such that *Succeed*(i_j^1 $i_1 .. i_{n+1}$ (σ')).

Before we give a sketch of Algorithm-G, observe the dependencies between the incremental SPE primitives as shown in Figure 5.5. This picture is read as follows. An edge between two nodes indicates that in order to revoke the action referenced by the node name, first related information as introduced by operations pointed at should be revoked. For example, before a region definition can be revoked, all objects defined within the region and its structure relations should be revoked. Another example, before an add_export operation can take effect, other dependent add_export and/or add_import operations should be revoked. This dependency framework forms the basis of the revocation algorithm.

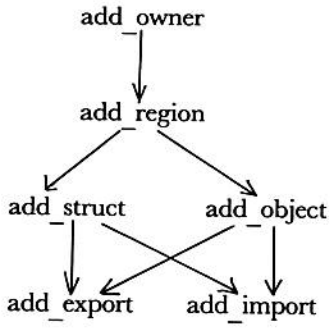


Figure 5.5 Dependencies among incremental instructions.

A sketch of the goal-seeking algorithm is given in Figure 5.6 using a Pascal-like language notation. The function revoke is called with a description of the action to be revoked, which authorizes the request. If this is successful, a compensation action is determined and applied to the state. If this instruction fails, the cause is determined and a remedy is sought. In particular, this may involve a recursive call of the procedure goal.

Note that authorization of a revoke takes place only once. Thus, it may happen that during the revocation process, the revoker is removed from the protection state. Moreover, this algorithm may run indefinitely due to interference when concurrent actions are allowed to occur. However, when concurrency is prohibited the process terminates with execution of the compensation action selected.

```
type action=record
{ This program is not supposed to compile }

type Operators = (add_region,{...},del_export);
        Action=record
                instruction:Instructions;
                activator:SPEname;
                region,parm3:SPEName;
                success, isrevoked: boolean;
        end;
        State  = SPE_state_description;

var  cstate:State;              { current state }

{ use routines specified in Figure 5.4 }

function essential(object,source,destination:SPEname):boolean;
begin
   { return true when the structure relation between source
     and destination for the object is essential. }
end;

function passed_on(object,source):boolean;
begin
   { return true when the access right on object is passed on
     to other regions }
end;

procedure goal(ca:Action);
var newac:Action;
begin
     if not authorized(cstate,ca) then return; { with some error message}
     {try to execute the compensation action}
     if transform(ca,cstate) then return; { with success}
     newac:= ca;
     case act.instruction of
     del_owner:
          { this means that ca.activator is the last owner of the region}
          { remove the region completely}
          newac.instruction:= del_region;
          goal(newac);
          break;
     del_region:
          { failure indicates existence of objects, multiple owners,
            structure relations, or import/exports. Remove these first.  }
          newac.instruction:= del_object;
```

```
            foreach object in ac.region
            begin
               newac.parm:= object;
               goal(newac)
            end;
            foreach co_owner in ac.region
            begin
               newac.parm:= co_owner;
               goal(newac)
            end;
            break;
        del_struct:
            { failure indicates essential use of the structure
              for import/export. Revoke these first. }
            foreach object in ac.region
            if essential(object,ac.region,ac.parm) then
            begin
               newac.instruction:= del_import;
               newac.parm:= object;
               goal(newac)
            end;
            { similar for revoking exports }
            break;
        del_object:
            { failure indicates existence of export/imports }
            foreach object in ac.region
            begin
               newac.instruction:= del_import;
               newac.parm:= object;
               goal(newac)
            end;
            { similar for revoking exports }
            break;
        del_import:
        del_export:
            { failure indicates further essential transport of export right }
            foreach object in ac.region
            if not passed_on(object,ac.region,ac.parm) then
            begin
               newac.parm:= object;
               goal(newac)
            end;
        end;
        {obstructive stuff has been removed, try original goal.}
        goal(ac);
    end;
```

```
procedure revoke(u:user; ca:Action);
begin
        ca.activator:= u;
        if authorized(cstate,ca) then goal(ca);
    end;
```

Figure 5.6 Algorithm-G, a goal-seeking algorithm.

The advantage of the goal-seeking scheme over a chronological scheme is that it does not require the state to be stable during the interpretation of revoke. This means that concurrently new objects can be defined and used. It is even possible that an object definition is revoked, while still in the process of revoking an export action on the object. The prime disadvantage of the scheme is that the precise revocation structure is nondeterministic and it implements a policy satisfying Definition 5.19.

The chronological revocation scheme and the goal oriented scheme are not equivalent, that is, they will not produce the same revocation sequence under the same starting conditions, not even when the goal-seeking procedure is assumed to have full control over the protection state. A simple example suffices to prove this. Consider the SPE state in Figure 5.5.a and its history in Figure 5.5.b.

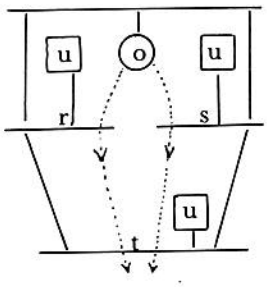


Figure 5.5.a SPE state

add_import(u,r,o) (1)

add_import(u,t,o) (2)

add_import(u,s,o) (3)

Figure 5.5.b

Then application Revoke(u,(1)) with the chronological scheme results in the revocation of action (2) too. Under the goal based scheme only one action is generated, namely del_import(u,r,o), because access of o in the region t is supplied by a different path as well. To summarize, the goal oriented approach ignores the order in which the protection state is reached.

5.5. SPE state predictions

The previous section illustrated how to generate a protection state that guarantees the revocation policy constraints. This section addresses the more general problem of how to characterize the set of reachable states using subsets of the SPE instruction set. For example, removing the add_user from the SPE instruction set fixes the protection state to users known in the initial state, but also places an upper bound in the ability to share access to objects, because in this particular case introduction of structure relations is limited.

Although reduction of the instruction set reduces the number of reachable states considerably, an analysis of the consequences reveals useful information on SPE implementations where, for example, the add_user instruction is available to a restricted class of users only. This way, the analysis sheds light on the SPE 'looseness', in the sense of [Lipton78] and the consequences for achieving both an acceptable level of protection and flexibility.

The reachable states are characterized by predicates and theorems, which relate an initial state to a class of applicable instructions. First, the structure graph properties are characterized by the predicate *Can.connect*, which forms the basis for transfer of access rights and which naturally leads to the predicates *Can.share* and *Can.obtain*. A second class of predicates is obtained by including information on the instruction activators. This leads to predicates characterizing stealing and conspiracy, which answer questions like: "Can a user obtain access to an object without help from any other user," and "how many users (and who) are needed to cooperate in accessing a particular object".

5.5.1. Connected regions

A central concept in the transfer of access rights is the existence of a (directed) path in the structure graph of a state. The predicate *path-connected* indicates whether such a path can be constructed using a subset of the SPE instruction set.

Definition 5.23

Two regions r and s are said to be *path-connected* in a state σ, if there exists an undirected path between r and s in the structure graph for the state σ.

Definition 5.24

Let σ be a secure state and P be a instuction set, then two regions r and s can be made *path-connected*, denoted by *Can.connect*(σ,r,s,P), if there exists a finite sequence of $p_i \in P$ such that both

$\sigma' = p_n \cdots p_1(\sigma)$ and r and s are *path-connected* in σ'

The initial state should be secure, otherwise a sequence with the desired properties does not exist. Moreover, to make r and s *path-connected*, the STR component should be extended, which can be done with the instruction add_struct only and therefore should belong to P. This instruction, however, assumes that both regions involved are members of R in the first place. If this

is not the case, a necessary prerequisite on the class P is that it contains the instruction add_region as well.

Note that the predicate *Can.connect* holds for those cases where the two regions are *path-connected* in the initial state already, because an empty sequence suffices. Moreover, when r and s are *path-connected* they belong to the same connected component of the structure graph and, alternatively, *Can.connect* expresses that two disconnected structure graph components can be connected using the instructions provided.

Theorem 5.21

If $r,s \in R$ and $\{add\_struct, add\_owner\} \subset P$ then *Can.connect*(σ,r,s,P).

Proof Assume that r and s are not *path-connected* in σ and have different owners, say u and v. Then the following sequence makes r and s *path-connected* in the final state;

add_owner(u,r,v); add_struct(v,s,r)

Both actions succeed, because ownership of r by u satisfies the authorization condition and add_owner pre-condition. The post-condition of add_owner implies that v is an owner of r too, which suffice to apply add_struct. If r and s have an owner in common, the first action is not needed. □

Theorem 5.22

If *Can.connect*(σ,r,s,P) and $\{add\_struct, add\_owner\} \subset P$ then a sequence of at most two actions suffice to make the regions r and s *path-connected.*

Proof The proof of the previous theorem gives an effective sequence.□

The constraint can be weakened somewhat by observing that the instruction add_owner is not needed for those cases where r and s have as least one owner in common, who takes the role of activator in both instruction calls.

Theorem 5.23

If $r,s \in R$ and $add\_struct \in P$ and $usr\_reg(r) \cap usr\_reg(s) \neq \emptyset$ then *Can.connect*(σ,r,s,P).

Proof Assume that r and s are not *path-connected* and let v be an owner common to both r and s. Then *Succeed*$(\sigma$,add_struct(v,r,s)) holds. Thus, (r,s) belongs to the structure component of the final state, which means that r and s are *path-connected.*□

The situations characterized by this theorem can simply be checked by looking at P and the owners of the regions involved. When $add\_owner \notin P$ nor $usr\_reg(r) \cap usr\_reg(s) \neq \emptyset$ then it is still possible that two regions can be made path-connected, provided that a sequence of regions can be found such that each pair in this sequence has an owner in common. Moreover, if it is known that two regions can be made *path-connected* then this can be done without extending

the class of regions.

Theorem 5.24

Can.connect(σ,r,s,*P*+{add_struct}-{add_owner}) iff there exist regions r_i (i=1..n) such that both

a) r_1=r, s=r_n and
b) $r_i \in R_\sigma$ and
c) usr_reg(r_i)∩ usr_reg(r_{i+1})≠∅ or
(r_i,r_{i+1})∈STR_σ or
(r_{i+1},r_i)∈STR_σ

Proof => Assume *Can.connect*(σ,r,s,*P*+{add_struct}-{add_owner}) and r and s are not *path-connected* in σ. Then there exists a sequence of instructions applied to σ such that in the final state σ' an undirected path between r and s exists, say r=r_1....r_n=s. Let r_j be the first not included in R_σ. Then a shorter sequence can be constructed with the path-connectedness property which does not include r_j, for r_j can effectively participate in the path when it added to the state together with both (r_{j-1},r_j) and (r_j,r_{j+1}) as structure relations. As no co-owners can be introduced, the owner of r_j should be a co-owner of both r_{j-1} and r_{j+1}. This implies that r_{j-1} and r_{j+1} can be connected directly with the add_struct instruction. Thus, each region not belonging to σ can be removed from the sequence giving constraint b).
Constraint c) expresses the situation that two successive regions in the path were path-connected already or can be made so by applying Theorem 5.23.
<= Induction on the number of regions and Theorem 5.23.□

The definition of the *path-connectedness* places no constraints on the direction of the structure relation. Neither does the SPE instruction add_struct, which leads to the following property:

Theorem 5.25

Can.connect(σ,r,s,*P*) ⟷ *Can.connect*(σ,s,r,*P*)

Proof Trivial.□

Note that *path-connectedness* does not require new users to be introduced either. Adding a new user implies the use of the add_owner instruction, which, together with add_struct, is already sufficient to construct a path directly (Theorem 5.21).

5.5.2. Sharing access between regions

The next step is to analyze under what conditions the predicate *Can.connect*(σ,r,s,*P*) and the instructions *P* can be used to transfer accessibility to an object between regions.

Definition 5.25

Let σ be a secure state. Then a region $r \in R_\sigma$ can share access to $o \in O_\sigma$ with region $s \in R_\sigma$, denoted by *Can.share*(σ,r,s,o,P), if there exists a finite sequence of $p_i \in P \subset OP$ such that

a) $\sigma' = p_n \cdots p_0(\sigma)$ and
b) *Access*(σ,o,r)= true and
c) *Access*(σ',o,s)= true

Condition b) restricts the predicate to regions and objects known in the initial state. To study access sharing with unknown regions and/or unknown objects, just pretend that they are properly included in the initial state, for this can be checked separately by inspecting P for inclusion of add_region and add_object, respectively.

Recall that the construction of IE graphs was based on the existence of a path in the structure graph. Thus *Can.connect* is a pre-requisite for *Can.share*. The predicate *Can.share* expresses the ability of users to extend the import-export graph associated with the object o such that a path between r and s is established. It is sufficient to check *Can.share*(σ,o,d,s,P) where d is the defining region of o, because when *Access*(σ,o,r) holds a path exists in the import-export graph from d to r and *Access*(σ',o,s) implies a path from d to s. Together, they make up an undirected path between r and s in the import-export graph.

Theorem 5.26

If {add_import,add_export} $\subset P$ and *Can.connect*(σ,r,s,P) then *Can.share*(σ,r,s,o,P);

Proof Assume that $(r=)r_1,\ldots,r_n(=s)$ is the structure path obtained by execution of the sequence of instructions defined by *Can.connect* giving σ', and let r_i be the last region in this sequence where *Access*(σ',o,r_i). Two cases are considered. If $(r_i,r_{i+1}) \in STR_\sigma'$ then apply add_export(u,r_i,o) by some owner of r_i giving σ''. If $(r_{i+1},r_i) \in STR_\sigma'$ then apply add_import(u,r_{i+1},o) by some owner of r_{i+1} giving σ''. The combined result is that *Access*(σ'',o,r_{i+1}) holds. This process can be repeated for all regions r_l ($l>i$).□

Access rights can be transferred between two regions with only the add_import instruction, provided that the structure path between r and s has the correct directionality or can constructed this way. The latter can be guaranteed when *Can.connect* holds and both regions are not *path-connected* in the initial state.

Theorem 5.27

If r and s are not *path-connected* in σ and *Can.connect*(σ,r,s, P+{add_import}) such that for each region pair (r_i,r_{i+1}) making up this structure path both

$r_i \in R_\sigma$, i=1..m and
$r=r_1$ and $s=r_m$ and *Access*(σ,o,r) and
$(r_i,r_{i+1}) \in STR_\sigma$' and
(usr_reg(r_i)$\cap$usr_reg(r_{i+1})$\neq\emptyset$ or $(r_{i+1},r_i) \in STR_\sigma$')

then *Can.share*(σ,r,s,o,P).

Proof The operation add_struct should be applied at least once, because r and s are not *path-connected* in σ. That is, add_struct is a member of P. Assume that $r=r_1...r_m=s$ is the structure path obtained by execution of the sequence defined by *Can.connect* giving σ'. Let r_i be the last region in this sequence where *Access*(σ',o,r_i). Two cases are considered: If $(r_i,r_{i+1}) \in STR_\sigma$' and usr_reg($r_i$)$\cap$usr_reg($r_{i+1}$)$\neq\emptyset$ then add_struct(u, r_{i+1},r_i) can be applied by some owner of both r_i and r_{i+1}, after which add_import(v,r_{i+1},o) can be applied by any owner of r_{i+1}. If $(r_{i+1},r_i) \in STR_\sigma$' then add_import(u,$r_{i+1}$,o) can be applied by any owner of r_i.
This process can be repeated for the tail of the region sequence. □

Notice that in the previous theorem *Can.connect* alone is not enough. An example to illustrate this is shown in Figure 5.7, where r and s can be made *path-connected* but where no access permission can be transmitted using just the import instruction. Region t plays the role of an agent, so that both an add_import and add_export instruction are required.

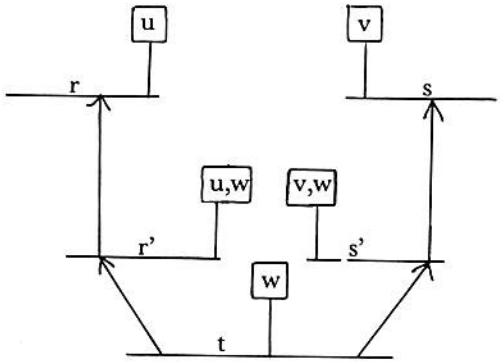


Figure 5.7

Theorem 5.28

If r and s are not *path-connected* in σ and *Can.connect*(σ,r,s, P+{add_export}) such that for each region pair (r_i,r_{i+1}) making up this structure path both (i=1..m)

$r_i \in R_\sigma$ and
$r=r_1$ and $s=r_m$ and *Access*(σ,o,r) and
$(r_{i+1},r_i) \in STR_\sigma$' and
$(usr\_reg(r_i) \cap usr\_reg(r_{i+1}) \neq \emptyset$ or $(r_{i+1},r_i) \in STR_{\sigma'})$

then *Can.share*(σ,r,s,o,P).

Proof Analogous to previous theorem.□

Knowledge about the validity of the predicate *Can.share* reveals the following properties:

Prop 1 If *Access*(σ,o,s) and *Access*(σ,o,r) then *Can.share*(σ,r,s,o,P) for any $P \subset OP$.

Prop 2 If *Can.share*(σ,r,s,o,P) then *Can.connect*(σ,r,s,P)

5.5.3. Sharing access by users

The two predicates *Can.connect* and *Can.share* characterize which regions can obtain access to objects. The next predicate in this series extends these concepts to describe the visibility of objects in reachable states.

Definition 5.26

Let σ be a secure state then a user $u \in U_\sigma$ can obtain access to the object o, denoted by *Can.obtain*(σ,u,o,P), if there exists a finite sequence of $p_i \in P \subset OP$ such that both

$\sigma'=p_n \cdots p_0(\sigma)$ and not *Visible*(σ,o,u) and *Visible*(σ',o,u)

If u doesn't belong to the initial set of users U_σ, the class P should contain the instruction add_ownerer, and when the object o does not belong to O_σ, the instruction add_object is needed.

Theorem 5.29

If add_owner$\in P$ then $\forall u \in U_\sigma$ and $\forall o \in O_\sigma$ *Can.obtain*(σ,u,o,P)

Proof Assume $o \in O_\sigma$, defined in the region r, and not *Visible*(σ,o,u). Then a new state can be generated by:

add_owner(v,r,u)

by some owner v of r, resulting in a new state σ' in which *Visible*(σ',o,u).□

This theorem indicates once again that the instruction add_owner is powerful in providing privileges to other users. Another characterization, more restrictive, is obtained by explicitly removing this instruction from the class P.

Theorem 5.30

Let $(u,s) \in OWN_\sigma$. If $\exists r \in R_\sigma$ and *Can.share*$(\sigma,r,s,o,P\text{-}\{add\_owner\})$ then *Can.obtain*$(\sigma,u,o,P\text{-}\{add\_owner\})$.

Proof Follows directly from the definition of *Can.share* and *Visible*. □

To increase the confidence on the protection properties of a subset P and the initial state, negations of the predicates described before are of interest too. For example, under what conditions are we sure that regions cannot be connected and users cannot obtain access to a particular object? Without proof a few easily derived properties are illustrated.

Theorem 5.31

Let r and s belong to R_σ. If r and s are not *path-connected* in σ and $add\_struct \notin P$ then *Can.connect*(σ,r,s,P) is false.

Theorem 5.32

Let r and s belong to R_σ. If not *Can.connect*(σ,r,s,P) then for all (un)defined objects o in σ *Can.share*(σ,r,s,o,P) is false.

Theorem 5.33

If $P \cap \{add\_import, add\_export\} = \varnothing$ and not *Access*(σ,r,o) then *Can.share*(σ,r,s,o,P) is false for all regions s.

Theorem 5.34

Let $o \in O_\sigma$, $r \in R_\sigma$, $u \in U_\sigma$, and *Access*(σ,o,r). If for each s such that $(u,s) \in OWN_\sigma$ and not *Can.share*$(\sigma,r,s,o,P\text{-}\{add\_owner\})$ then *Can.obtain*$(\sigma,u,o,P\text{-}\{add\_owner\})$ is false.

From the perspective of ensuring a certain level of protection we conclude that the instruction add_owner is far and away the most dangerous instruction, for it allows to freely pass along privileges, just by the introduction of co-owners. A useful property of all predicates is that they can be answered either in constant time, by inspection of P, or by analysis of the graphs derived from the protection state, for which the complexity of the algorithms involved is bounded by the number of regions or structure relations.

5.5.4. Stealing

In the definition of the predicates *Can.obtain*, *Can.share* and *Can.connect* we were concerned whether access rights could be transferred by applying a subset of the SPE instruction set. Thus, we hypothesized complete cooperation from all users. In the definition of the next predicate, *Can.steal*, the set of users willing to cooperate in the transfer of access rights is limited and made explicit. This predicate models questions like "can a user obtain access to an object, given a subset of the SPE instruction set and a subset of the users," or more specifically,

"can a (non-trustworthy) subject steal access to a given object with the aid of a group of conspirators."

Our notion of stealing differs slightly from similar definitions in [Snyder79, Snyder81] by considering users having access to the object as potential conspirators as well. This generality extends the semantics one normally attaches to the word stealing, i.e. a person in possession of some objects would not actively help a thief. Yet, our definition allows us to easily identify those users subject to bribery too.

Definition 5.27

Let σ be a secure state. Then a user u can steal the object o with collaboration of conspirators in *Csp*, $u \in Csp \subset USERS$ and denoted by *Can.steal*(σ,u,o,*P*,*Csp*), if there exists a finite sequence of state transformations $p_i \in P \subset OP$ such that

$\sigma' = p_n \ldots p_0(\sigma)$ and not *Visible*(σ,o,u) and *Visible*(σ',o,u) and

$\forall\, p_i$ (i=o..n) the activator of p_i belongs to *Csp*.

Note that this definition allows for the situation where the object is accessible to one or more members of the group *Csp* already. Furthermore, it assumes that the potential thief u doesn't have access to the object, otherwise it would not reveal more information than the predicate *Visible*. Note too that *Can.steal* is a refinement of the predicate *Can.obtain* and that *Can.steal* logically implies *Can.obtain*.

This definition of stealing can be used to answer the following questions as well: "can any object be stolen by a user u from a particular region r" and "can an object o defined in region r be stolen by any user v", because the former problem can be translated into defining a new object o' in the region r and evaluating the predicate *Can.steal*(σ,u,o',*P*,{u}). The latter problem is solved by considering all non-owners of r as conspirators.

First assume that *Csp*={u}, where the potential thief is the only user who can issue actions. If $u \notin U_\sigma$ then again the predicate is inherently false, because unknown users can not gain access to any object at all without the help of at least one conspirator. The active cooperation of a user is essential here, for he has to make the malicious user known within the SPE protection state using the instruction add_owner. Therefore, for the rest of this analysis we assume that u belongs to the users U_σ already.

Before proceeding with our analysis let us introduce a concept found in other protection models and approaches, namely the access relationship or access matrix [Harrison76]. The access relation is defined between users and objects and administers the objects visible to a particular user. A similar notion can be defined for SPE as well:

Definition 5.28

Let u be a user in the secure state σ. Then the set of visible objects VO for u is defined as follows:

$$VO(\sigma,u)= \{ o \in O_\sigma : \mathit{Visible}(\sigma,o,u)\}$$

The predicate *Can.steal*, like *Can.obtain*, is easily translated into conditions on the set VO, for stealing the object o means applying a sequence of instructions from P by the users in Csp, such that $o \notin VO(\sigma,u)$ and $o \in VO(\sigma',u)$. Therefore, we should derive the conditions under which $VO(\sigma,u)$ is affected by the SPE instructions. An important result is that any malicious user can only gain access to existing objects, for which it had no access before, by applying the instruction add_import.

Theorem 5.35

Let the set Csp consist of a single user u. Then $VO(\sigma,u)$ can be extended using add_import or add_object instruction only.

Proof The proof is given by analysis of the SPE instructions. Obviously, only the incremental instructions should be considered. First, usage of add_owner to introduce new owners only does not affect $VO(\sigma,u)$. However, VO is changed when u is made co-owner of, say, the region r, but this can only take place with the cooperation of other users and u can never do it alone. Thus, add_owner does not affect $VO(\sigma,u)$ when no additional conspirators exist.
Applying the instruction add_region clearly does not alter $VO(\sigma,u)$. The creation of new objects with the add_object instruction does and the objects are visible in their defining region, owned by u.
The add_export(v,r,o) extends VO for all owners of the regions in the environment of r, which may include the user u. However, as u is considered the only activator, the object o already belongs to $VO(\sigma,u)$. Thus add_export does not extend $VO(\sigma,u)$.
The instruction add_struct(v,r,s) couples two regions and may change $VO(\sigma,w)$ for all owners w of s as a side-effect of export operations, i.e. for some o' $(o',r) \in EXP_\sigma$. However, the authorization constraint of add_struct requires that the activator is owner of both regions involved. Thus, o' was visible to u already and $VO(\sigma,u)$ is not extended.
This leaves us with the instruction add_import, which indeed can be used to make objects accessible within the environment of a region, accessible within the region itself. □

This theorem gives a simple characterization of the objects which can be stolen by a single user u and the instructions needed. All objects subject to stealing by u alone are found in the environment of the regions owned by u. Conversely, it means that if a user is isolated, in the sense that the environment of each of his regions is empty, he is unable to steal any objects.

Corollary *Can.steal*(σ,u,o,*P*,{u}) iff add_import∈*P* and for some region r: *Access*(σ,o,r) and some s such that $\overline{(u,s)}$∈OWN and r∈environment(s).

Stealing an object need not be performed by the malicious user u himself. It may happen that one of the users v∈*Csp*-{u} has access to the object o in σ and is able to share this access with u. Alternatively, user v∈*Csp*-{u} may be able to steal it, before passing it along. Therefore, a necessary prerequisite for *Can.steal*(σ,u,o,*P*,*Csp*) is the following constraint on the set of conspirators:

$$\exists v \in Csp\ \exists r{:}(v,r) \in \mathrm{OWN} \ \wedge\ (Access(\sigma,r,o) \vee Can.steal(\sigma,v,o,P,Csp\text{-}\{u\}).$$

To investigate the role of conspirators, assume that *Can.steal*(σ,u,o,*P*,{u}) is false and that for some user v *Can.steal*(σ,v,o,*P*,*Csp*-{u}). Then we should consider the question: under what conditions the user v can share access to o with u? The approach taken in the next section is to derive predicates, related to *Can.connect*, *Can.share*, and *Can.obtain*, and to apply a condensation technique on the protection state, which results in cheap algorithms to determine conspiracy.

5.5.5. Kernel, import, and export areas

The question whether or not a conspirator can effectively help a malicious user in obtaining access to an object is called the conspirator problem. It requires a characterization of the conditions under which a user v can pass its access permissions on an object o from a region r to some user u. The conspirator problem is generalized by considering the question: if *Access*(σ,o,r) and (v,r)∈OWN then what are the regions where user v can make the object accessible using *P*.

The analysis of the previous sections indicates the power of the add_owner instruction in transferring access permissions, a good reason to omit this instruction from *P* here. Availability of add_owner requires at most one user to conspire in stealing, namely, any user having access to the object or being able to import it from its environment. With this restriction on the instructions available in mind only, the regions belonging to the initial state must be considered only, as illustrated by the next theorem.

Theorem 5.36

Let r and s be regions in σ owned by u and v, respectively. If an access right can be passed from r to s using a region t∉Rσ with a finite sequence of state transformations from *P*-{add_owner}, then there exists a finite sequence for passing access rights between r and s without t.

Proof Assume that $p_n \ldots p_1$ denotes the finite sequence generating σ' and both *Access*(σ,o,r) and *Access*(σ',o,s). Moreover, assume that the region t is essential in this transport, that means, there exists instructions such that

- p_i introduces t as a new region, i.e. add_region is applied, and
- p_j (j>i) and p_k(k>i) are applications of add_struct to connect region t with two other regions, say r' and s', and
- p_l(l>j) and p_m(m>k) are either add_import and/or add_export to transfer access from r' to t and from t to s' respectively.

The activator v of p_i becomes the sole owner of the new region t. Applying add_struct (p_j and p_k) requires that v owns both regions involved, which means that usr_reg(r')∩usr_reg(t)={v} and usr_reg(t)∩usr_reg(s')={v}, because the add_owner instruction is not available. However, then add_struct can be applied by v to directly connect r' and s', which means that a shorter sequence can be obtained by dropping the five instructions p_i,p_j,p_k,p_l,p_m and introducing two others. □

This theorem shows that we can characterize conspirators using the regions and users in the initial state and instruction set only. In turn, we can group regions and users with similar properties and thereby abstract from the SPE state, which is a condensation technique applied to many graph oriented problems. For this purpose we introduce three new concepts, the *export area*, the *import area*, and the *kernel area*.

Definition 5.29

An *export area* $EA(\sigma,u,r,P)$ is the set of regions s in R_σ such that if $Access(\sigma,o,r)$ (for any object $o\in O_\sigma$) then a finite sequence of $p_i\in P$ (i=0..n) exists with:

$\sigma' = p_n \cdots p_0 \sigma$ and
$Access(\sigma',o,s)$ and
u is the activator of p_i.

The export area characterizes all the regions which can receive access permissions available in the region r, called the area center, by actions (at least one) of a single user u, called the area owner. An export area is necessarily qualified by both a user and a region, because it may happen that a user owns two regions which can not be made *path-connected* using the instructions P and thus access rights can never be passed between them. Observe the similarity with the previously defined predicate *Can.share* with the only difference being lack of constraints on the activator.

Definition 5.30

An *import area* $IA(\sigma,u,r,P)$ is the set of regions s in R_σ such that if $Access(\sigma,o,s)$ (for any object $o\in O_\sigma$) then a finite sequence of $p_i\in P$ (i=0..n) exists with:

$\sigma' = p_n \cdots p_0 \sigma$ and
$Access(\sigma',o,r)$ and
u is the activator of p_i.

The import area characterizes all regions from which access permissions can

be transferred to the area center r by actions issued by the area owner u. The relationship between export and import areas is not symmetric. That is, $s \in EA(\sigma,u,r,P)$ does not necessarily imply that $r \in EA(\sigma,u,s,P)$, because the access flow instructions have a different directionality.

Definition 5.31

A *kernel area* $KA(\sigma,u,r,P)$ is the set of regions s in $EA(\sigma,u,r,P) \cap IA(\sigma,u,r,P)$

The kernel area characterizes regions interchangeable under the given protection primitives. Any access permission for r can be transported to all regions in the kernel area and, conversely, each access permission associated with a kernel region can be transported to r, its center. This implies that any region in the kernel area owned by u can be chosen as area center. The properties of the kernel area regions are stipulated by the following theorems:

Theorem 5.37

$$KA(\sigma,u,r,P) \cap KA(\sigma,u,s,P) \neq \emptyset \rightarrow KA(\sigma,u,r,P) = KA(\sigma,u,s,P)$$

Proof First, we show that $KA(\sigma,u,r,P) \subseteq KA(\sigma,u,s,P)$. Let $t \in KA(\sigma,u,r,P) \cap KA(\sigma,u,s,P)$. Then there exists sequences of $p_i \in P$ which transfer access permissions from r to t and vice versa. As $t \in KA(\sigma,u,s,P)$ access can be transferred from t to the center s too. Combination of the paths makes r a member of $KA(\sigma,u,s,P)$. Induction on all other regions in $KA(\sigma,u,r,P)$ ensures: $KA(\sigma,u,r,P) \subset KA(\sigma,u,s,P)$. A similar argument can be used to prove $KA(\sigma,u,r,P) \supseteq KA(\sigma,u,s,P)$.□

The non-empty intersection conditions in this theorem can be refined by focusing on the properties of import and export. The next theorem shows that for a given area center and availability of both the add_import and add_export instructions, kernel-, import-, and export- areas are the same.

Theorem 5.38

If $\{add\_import, add\_export\} \subset P$ then

$$IA(\sigma,u,r,P) = EA(\sigma,u,r,P) = KA(\sigma,u,r,P)$$

Proof We shall show the first equality only, as the second follows directly from the definition of kernel areas. Let $t \in IA(u,r,\sigma,P)$. Then there exists a sequence which transfers access from region t to the area center using a structure path. However, the same path can be used to transfer access from r to t using add_export. This means that t belongs to $EA(\sigma,u,r,P)$. A similar argument proves that any region t in $EA(\sigma,u,r,P)$ belongs to $IA(\sigma,u,r,P)$.□

Theorem 5.39

If $r \neq s$ and $IA(\sigma,u,s,P) \cap EA(\sigma,u,r,P) \neq \emptyset$ and $\{add\_import, add\_export\} \subset P$ then $KA(\sigma,u,s,P) = KA(\sigma,u,r,P)$

Proof The previous theorem states that $IA(\sigma,u,s,P) = KA(\sigma,u,s,P)$ and

$EA(\sigma,u,r,P)=KA(\sigma,u,r,P)$. Applying Theorem 5.37 gives equality of kernel areas.□

Observe that a non-empty intersection of import and export areas centered around r alone does not imply that these areas contain the same regions. A simple counterexample is obtained by excluding the add_export instruction from P. Then the export area centered around r contains no other regions, i.e. $EA(\sigma,u,r,P)=\{r\}= KA(\sigma,u,r,P)$, while the import area for r contains all the regions in the environment (or potential environment by application of add_struct).

The three area concepts abstract from the initial state and can now be used to solve the conspirator problem directly. First, a user can pass access permissions to all regions in the export areas for which he is an area owner. Second, overlap of an import area $IA(\sigma,u,r,P)$ with the export area $IA(\sigma,v,s,P)$ is sufficient to pass access from s to r.

Theorem 5.40

Let $IA(\sigma,v,s,P)\cap EA(\sigma,u,r,P)\neq\varnothing$ and $Access(\sigma,o,r)$. Then user u can share access to o with v in region s.

Proof Let $t\in EA(\sigma,u,r,P)\cap IA(\sigma,u,sP)$. According to Definition 5.29 there exists a finite sequence of state transformations, issued by u, such that $Access(\sigma',o,t)$. Moreover, t is a member of the import area of $IA(\sigma,v,s,P)$ and thus there exists a finite sequence, activated by v, such that $Access(\sigma'',o,s)$ holds. □

This analysis solves the stealing problem and conspiracy problem, because objects potentially being stolen by a single user u should be accessible in a region belonging to an import area owned by u, say r. Moreover, the area concept shows where the stolen access right can be used by u (or passed on to u), namely in all regions belonging to the kernel area $KA(\sigma,u,r,IP)$.

Stealing with conspiracy requires the conspirators to transfer access rights between their kernel areas. This concept is visualized by a *conspiracy graph* using the notational convention that $IA(\sigma,u,r,P)$ denotes an area owned by u with center r.

Definition 5.32

The *conspiracy graph*, $CG(\sigma,P,Csp)$ is a directed graph in which the nodes correspond to all kernel areas of the conspirators Csp. An edge (r,s) between nodes $IA(\sigma,u,r,P)$ and $IA(\sigma,v,s,P)$ exists if

$$EA(\sigma,u,r,P)\cap IA(\sigma,v,s,P)\neq\varnothing$$

A node in the conspiracy graphs represents all regions in a kernel area. Arcs depict the possibility to send access rights form one kernel area to another. The result of this construction is a directed, cyclic graph with the following property:

Theorem 5.41

Let *Access*(σ,o,r) and let u own region r. Then u can transfer access rights on o to v in region s using conspirators *Csp*+{u} iff there exists a directed path between r and s in CG(σ,*P*,*Csp*)

Proof Use Definition 5.10 and Theorem 5.20. □

Corollary *Can.steal*(σ,u,o,*P*,*Csp*) is true if there exists a directed path in the conspiracy graph CG(σ,*P*,*Csp*) between two node *IA*(σ,v,s,*P*) and *IA*(σ,u,r,*P*) such that object o can be imported into s or is accessible in the kernel of *IA*(σ,v,s,*P*).

Using the conspiracy graph we can obtain the identity of the minimum collection of conspirators for the malicious u in obtaining access to an object o using a simple graph algorithm. Consider the kernel area where o is accessible or into which is may be imported. Then let S1 be the class of all paths between r and these kernel. For each path in S1 count the number of different users; sorting gives the minimal number of users. Sorting S1 on the path length gives the minimal number of areas involved, i.e. a measure for the minimal amount of work.

5.6. SPE programs

The SPE instruction set can be considered as the instruction set of an abstract protection machine and the predicates derived in the previous section define the reachable states and the conditions to be met when reachability of states is to be precluded. This analysis is based on the assumption that arbitrary sequences of SPE instructions can be issued. In this section the concept of an SPE program is defined and security questions for a fixed group of programs are considered. By means of a program concept new abstract machines can be defined with additional security semantics. It is necessary, though, that the SPE machine itself can be hidden in an actual implementation.

The SPE programs are introduced as composite SPE operations, that is, linear sequences of SPE instructions. To accommodate additional protection policies SPE programs may be qualified with constraints on the applicable state and function parameters, leading to the notion of conditional SPE programs or guarded SPE instruction sequences.

The SPE program concept is used to simulate two well-known formal access control models, the Harrison-Ruzzo-Ullman model and the Take-Grant model, and we show the differences in handling the security problems. Subsequently we indicate how additional security properties can be attacked and to what extent the predicates derived for SPE help. This approach provides the stepping stone for the analysis of the protection of a given situation and to indicate what should

be checked at run-time. Note, however, that a well-founded theory to support such an analysis in all its details is far beyond the scope of this thesis.

5.6.1. A model for SPE programs

A commonly used definition of a program defines it as an abstraction of a particular sequence of instructions. In other words, a program is a partial function taking the domain of input values over into the domain of output values. This definition can be carried over to the domain of the SPE model by considering the initial state and parameters given to the SPE instructions as input parameters and the final state as its output. With this in mind an SPE program can be considered as a composite SPE state transformation.

Definition 5.33

An SPE program $f(x_1,x_2,..,x_n)$ is a composite mapping
$\phi_1 \circ .. \circ \phi_n : S -> S$
of SPE instructions.

Following the programming language conventions, the SPE instructions ϕ_i are called statements and x_i the program parameters. For simplicity we assume that each program parameter is used and that the program consists of at least one statement. The statements considered are drawn from the incremental and decremental SPE instructions only, which ensures that whenever the *auth-* and the *pre*-condition are satisfied, a new secure state is derived. Observe that this definition of a program differs from its use in a programming language environment in that individual statements may be invoked or activated by different users, each with different authorization properties.

Example

newuser(u,r,v)= del_owner(v,r,u) ∘ add_ownerer(u,r,v) ∘ add_region(u,r)
defines an SPE program, which creates a region r and makes v the sole owner of it. In the sequel a programlike syntax is used. References to entities in the global protection state are enclosed by single quotes. The aforementioned example is written as follows:

```
command newuser(u,r,v)
begin
        add_region(u,r);
        add_ownerer(u,r,v);
        del_owner(v,r,u)
end
```

Definition 5.34

The effect of an SPE program $f(x_1,x_2,...,x_n)$ on the state $\sigma \in \Sigma$ is defined by

$$f(x_1,x_2,...,x_n)\ (\sigma) = \textbf{if } \forall \phi_i\ (i=1..k)\ \phi_i(\sigma_i) \neq \sigma_{i-1} \textbf{ then } \phi_k \circ ... \circ \phi_1(\sigma) \textbf{ else } \sigma$$

This definition is interpreted as follows. The program f with parameters x_1 to x_n applied to the secure state σ generates a new state σ' as defined by the composite state transformation, or generates an error and makes $\sigma'=\sigma$. An error occurs when any of the individual components did not generate a new state.

The requirement that each statement produces a new state simplifies the analysis of the security property of SPE programs, because it disallows erroneous programs and no-op instructions. Notice that this simplification does not restrict SPE programs to simple error free straight line programs, but merely requires a straight line program for each possible correct path through the program instead.

The definition of SPE programs as (partial) composite state transformations provides limited tools for the design and analysis of new protection policies. One should also be able to constrain the execution of a program, beyond the conditions of with the SPE instructions. One approach to control the applicability of a program execution is to extend the heading with a constraint on the the actual state and parameters, i.e. introduce a condition on the invocation of an SPE program. This predicate is evaluated upon program activation as part of its initialization. The resulting program is called a conditional SPE program and is formally defined by:

Definition 5.35

A conditional SPE program $f(x_1,x_2,..,x_n,Q)$ is a mapping from S into S, with Q a predicate, such that the effect of program f on the state $\sigma \in S$ is defined by

$$f(x_1,x_2,..,x_n,\ Q\ )(\sigma) = \textbf{if } Q(x_1,..,x_n,\sigma) \textbf{ and } \forall \phi_i\ (i=1..k)\ \phi_i(\sigma_i) \neq \sigma \textbf{ then } \phi_k \circ ... \circ \phi_1(\sigma) \textbf{ else } \sigma$$

Example The program newuser defined above can be constrained as follows

```
command newuser(u,r,v) if u ∈ {'root','sys'} begin
        add_region(u,r);
        add_ownerer(u,r,v);
        del_owner(v,r,u) end
```

which ensures that the two named users can execute this program. The notion of a conditional program is closely related to the *activator* concept in the operational model proposed by Minsky [Minsky78a] and provides a basis for dynamic access protection for databases in [Fernandez81], because it can be

used to place conditional guards on the execution of protection system programs.

The notion of SPE programs allows for the definition of a hierarchy of *abstract machines*, an approach first used by Dijkstra in the THE operating system [Dijkstra66]. The lowest level, level 0, is formed by the SPE model and its instructions. A group of (conditional) SPE programs defines a new abstract machine as level 1. These programs are defined by instructions of level 0. Moreover, the programs inherit the security properties of level-0, that is, a proper implementation of level 0 ensures that no insecure state can be derived. In general, multiple levels can be introduced this way, giving a series with a program at level i defined in terms of programs of level i-1.

The introduction of new abstract machines allows to abstract from the underlying machine; that is to change, enhance, or restrict the security policies provided by the lower levels. This method is illustrated in the following sections by showing how the Harrison-Ruzzo-Ullman and Take-Grant model can be simulated by a set of conditional SPE programs. The gain of this exercise is twofold. First, it illustrates the potentials of the SPE model in building alternative protection systems on top of it and to enforce its protection policy. Second, it illustrates the differences between these three models. In particular, it shows that the access matrix model implementations, capability lists and access control lists, ignore the possibility to take the entries of the access matrix as a departure point for access control implementation. The latter, however, closely resembles the premises in the SPE model, where neither the users nor the objects are taken as a starting point for access description, but their relationship in a particular environment. Moreover, it shows that the access flow is ignored in the HRU model.

5.6.2. Simulation of HRU with SPE

The real world in HRU is modeled by the triple (S,OB,AM) called a protection system configuration, with AM=S×OB×R an access matrix, S is a set of subjects, OB a set of objects, and R a set of rights. It is assumed that S is a subset of OB, which makes it possible to model access relations among users as well. In the simulation given below we restrict the set of generic rights to one element, namely accessibility. An alternative scheme which handles multiple rights is indicated shortly.

The HRU access matrix shown in Figure 5.8 is mapped to the SPE structure as shown in Figure 5.9 and the corresponding state description is given in Figure 5.10. The subjects in HRU are associated with users in SPE, the rows are mapped to regions owned by the subject associated with the row and, finally, the columns are represented as objects in a single region. Having access to an object or a subject is equivalent to having access to its SPE object representation.

	s_1	..	s_n	o_1	..	o_m
s_1						
..						
s_n				r		r

Figure 5.8 HRU Access matrix

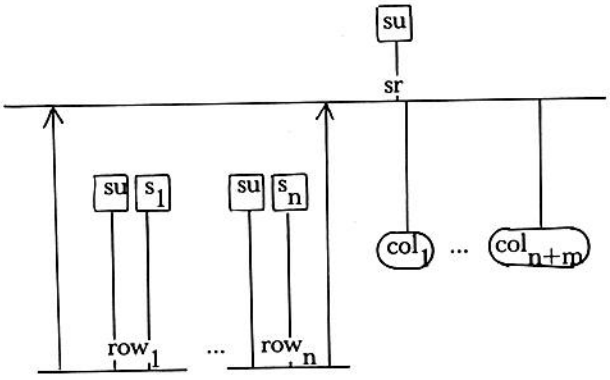


Figure 5.9 The SPE representation of the access control matrix.

HRU= <U,O,R,OWN,STR,DEF,IMP,EXP>

$U = \{su, s_i: i=1..n\}$
$O = \{col_i: i=1..n+m\}$
$R = \{sr, row_i: i=1..n\}$
$OWN = \{(s_i, row_i): i=1..n\} \cup \{(su,sr)\} \cup \{(su, row_i): i=1..n\}$
$STR = \{(row_i, sr): i=1..n\}$
$DEF = \{(col_i, sr): i=1..n+m\}$
$IMP = \emptyset$
$EXP = \emptyset$

Figure 5.10 SPE state description.

The HRU instruction set consists of six operators, which are mapped to conditional SPE programs as shown in Figure 5.11.

HRU model	*SPE model*
enter right into (s,o)	**command** enter(s,row,o)
	if (s,row)∈OWN

```
                    begin
                        add_import(s,row,o)
                    end
delete right from (s,o)  command delete(s, row, o)
                    if (s,row)∈OWN
                    begin
                        del_import(s,row,o)
                    end
create subject s    command cr_subject(s,row,col)
                    begin
                        add_region('su',row);
                        add_owner('su',row,s);
                        add_struct('su',row,'sr');
                        add_object('su','sr',col)
                    end
delete subject s    command del_subject(s,row,col)
                    begin
                        del_object('su','sr',col);
                        del_struct('su',row,'sr');
                        del_owner('su',row,s);
                        del_region('su',row)
                    end
create object o  command cr_object(o)
                    begin
                        add_object('su','sr',o)
                    end
delete object o  command del_object(o)
                    begin
                        del_object('su','sr',o)
                    end
```

Figure 5.11 SPE commands for HRU instructions

To illustrate, the HRU instruction 'enter right into (S,O)' can be simulated with the SPE program 'enter(S,Row,O)'. Note that an extra parameter is needed to indicate the region where accessibility is required. Moreover, enter is constrainted, which implies that when S does not own the region Row or O is not an object an error occurs and the program leaves the protection state unchanged. Another example, the usage of the operation 'delete subject S', i.e. del_subject(S,Row,Col), consists of three SPE instructions to remove row, column and user, but which can only be executed successfully when all rights, i.e. imports, are removed first.

An obvious distinction between SPE and HRU is the lack of an authorization constraints within the HRU model. In principle each user can add/remove users and objects. This is circumvented in the simulation using the user 'su', who has access to all regions and objects and who is the only subject to introduce/remove subjects and objects. It represents the user held responsible

for the contents of the access matrix.

As the HRU instruction set does not incorporate the notion of authorization, no real access protection is provided at all. Any user who can apply the instructions can transfer access rights or remove objects from the protection state. The solution proposed to control the use of rights is based on an abstraction mechanism, similar to the conditional programs in SPE. HRU assumes that all changes are performed through calling guarded commands, rather than one of its instructions. An example of such a command is shown in Figure 5.12. No command is executed when any of the instructions would fail against the actual state or when the condition does not evaluate to true. Higher level protection policies, like flow control of access rights, should be encoded by the user in the form of these commands.

command $\alpha(X_{s1},..,X_{sm},X_{o1},..,X_{on})$
if r_1 **in** (X_{s1},X_{o1}) and
  r_2 **in** (X_{s2},X_{o2}) and
  ...
  r_m **in** (X_{sm},X_{on})
begin
  op_1
  ...
  op_k
end

Figure 5.12 Harrison-Ruzzo-Ullman command structure

A second important distinction between SPE and the HRU model is the definition of the security problem. In SPE security is defined as a collection of state transformation invariants. In HRU security violation is associated with the ability to reach some undesirable state, called an unsafe state. More specifically, HRU defines lack of safety as the ability to modify the access matrix such that a right can be entered in a position where it wasn't before. Interpreted in the context of a class of non-trustworthy subjects it corresponds to the stealing problems in SPE. A formal definition of this notion is given by: [Harrison76]

Given a protection system and a generic right r, we say that the initial configuration Q_0 is unsafe for r if there exists a configuration Q_f and a command alpha such that:

1) $Q_0 \vdash^{*} Q_f$
2) alpha leaks r from Q_0

This general notion of safety (=state security) is, although theoretical interesting, of limited use for the construction of secure systems. Its prime result is that the general safety question is undecidable. That is, no algorithm exists which, given a set of commands and an arbitrary state, tells whether the system is safe for this

state or not. Yet, its point of departure is radically different from the approach taken in this thesis where security is an invariant property of state transformations, rather than a property of reaching an undesirable protection state. Once all commands have been proved to guarantee the security invariants and implemented correctly, no insecure state can ever be derived.

5.6.3. Simulation of SPE with HRU

The previous analysis showed that the HRU model and its instructions are easily modeled by conditional SPE programs. The question arises whether the converse is true as well, that is, is it possible to simulate the SPE model using the HRU instructions and its command construction facility. First, the SPE protection state is mapped into the access matrix AM. HRU contains the notion of user (called subjects) and objects already, thus they can be used to represent the SPE users and objects, respectively. The mapping of entity sets and the binary relations are represented by generic rights, i.e.

R={User, Object, Region, Own, Str, Def, Imp, Exp}

The first row of the access matrix is used to denote the type associated with the column, it contains a single right from {User, Object, Region}. This row is assigned a fictive user 'type' and provides a means to express simple type constraints, like $u \in U$. The regions in SPE are used in both dimensions of the binary relations and thus each region should be represented similarly to a user in HRU. That is, a new region results in the creation of both a row and a column with the same name. The structure relation (r,s) results in the entry M[s,r] with the right Str. Regions are used as intermediaries between users and objects, which means that the matrix elements M[u,u'] (u,u' are users) and M[u,o] (o object) remain empty. Moreover, a compound right ({Own,..,Exp} guarantees proper types are available for the components.

In Figure 5.13 a small SPE protection state is depicted, the corresponding HRU access matrix is shown in Figure 5.14.

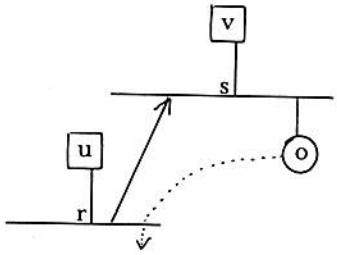


Figure 5.13 Example SPE protection state.

	u	v	r	s	o
type	User	User	Region	Region	Object
u			Own		
v				Own	
r				Str	Imp
s					Def

Figure 5.14 HRU access matrix representation for SPE

SPE instruction	*HRU command*
add_region(u,s)	**command** add_region(u,r) **if** User ∈ M['type',u] **begin** create subject r create object r enter Region into M['type',r] enter Own into M[u,r] **end**
add_object(u,r,o)	**command** add_object(u,r,o) **if** Own in M[u,r] **begin** create object o enter Object into M['type',o] enter Def into M[r,o] **end**
add_owner(u,r,v)	**command** add_owner(u,r,v) **if** Own in M[u,r] $\overline{\wedge}$ User in M['type',v] **begin** enter Own into M[v,r] **end** **command** add_owner(u,r,v) **if** Own in M[u,r] **begin** create subject v create object v enter Own into M[v,r] **end**
add_struct(u,r,s)	**command** add_struct(u,r,s) **if** Own in M[u,r] $\overline{\wedge}$ Own in M[u,s] **begin** enter Str into M[s,r] **end**
add_export(u,r,o)	**command** add_export(u,r,o) **if** Own in M[u,r] $\wedge$ Def in M[r,o]

```
                    begin
                          enter Exp into M[r,o]
                    end
                    command add_export(u,r,o)
                    if Own in M[u,r] ∧ Imp in M[r,o]
                    begin
                          enter Exp into M[r,o]
                    end
                    command add_export(u,r,o)
                    if Own in M[u,r] ∧ ∃s: (Exp in M[s,o] ∧ Str in M[s,r])
                    begin
                          enter Exp into M[r,o]
                    end
add_import(u,r,o)   command add_import(u,r,o)
                    if Own in M[u,r] ∧ ∃s: (Def in M[s,o] ∧ Str in M[r,s])
                    begin
                          enter Imp into M[r,o]
                    end
                    command add_import(u,r,o)
                    if Own in M[u,r] ∧ ∃s: (Imp in M[s,o] ∧ Str in M[r,s])
                    begin
                          enter Imp into M[r,o]
                    end
                    command add_import(u,r,o)
                    if Own in M[u,r] ∧ ∃s,t (Exp in M[s,o] ∧ Str in M[s,t] ∧ Str in M[r,t])
                    begin
                          enter Imp into M[r,o]
                    end
```

Figure 5.15 Mapping the SPE incremental instructions

Figure 5.15 shows that the mapping of SPE instruction into HRU commands is relatively straightforward. The authorization conditions are mapped to a test on the availability of the proper rights or a test on rights administered in the 'type' row. Moreover, the semantics of HRU commands is used to guarantee the additional constraints. For example, in the routine add_region no check is made on the existence of region r in the initial state, because the interpretation of the HRU command ensures that either all actions succeed or none. Unfortunately we have to resort to multiple commands to cover all cases. Moreover, the existential quantifier is needed to handle import and export of privileges, a concept not used within the original definition of HRU. However, as the quantifier runs over a finite state description it can be considered a shorthand for a disjunction involving all elements. Disjunctions are allowed implicitly through command replication in [Harrison76].

SPE instruction	*HRU command*
del_region(u,r)	**command** del_region(u,r) **if** Own in M[u,r] ∧ ∀t:M[r,t]=∅ **begin** delete Own from M[u,r] destroy subject r destroy object r **end**
del_object(u,r,o)	**command** del_object(u,r,o) **if** Own in M[u,r] ∧ ∀t:M[t,o]∪{Imp,Exp}=∅ **begin** destroy object o **end**
del_owner(u,r,v)	**command** del_owner(u,r,v) **if** Own in M[u,r] ∧ Own in M[v,r] ∧ ∃s≠r: Own in M[v,s] **begin** delete Own from M[v,r] **end** **command** del_owner(u,r,v) **if** Own in M[u,r] ∧ Own in M[v,r] ∧ not ∃ s≠r: Own in M[v,s] **begin** delete User from M['type',v] delete Own from M[v,r] destroy subject v destroy object v **end**
del_struct(u,r,s)	**command** del_struct(u,r,s) **if** (Own in M[u,r] ∨ Own in M[u,s]) ∧ Notused(r,s) **begin** delete Str from M[r,s] **end**
del_import(u,r,o)	**command** del_import(u,r,o) **if** Own in M[u,r] ∧ Imp in M[r,o] ∧ remainsvalid(r,o) **begin** delete Imp from M[r,o] **end**
del_export(u,r,o)	**command** del_export(u,r,o) **if** Own in M[u,r] ∧ Exp in M[r,o] ∧ remainsvalid(r,o) **begin** delete Exp from M[r,o] **end**

Figure 5.16 HRU commands to simulate decremental instructions

Similar to the definition of the SPE instruction set in section 5.1 the constraints placed on decremental parameters results in complex *pre*-conditions.

Conjunction testing for the presence of rights in the matrix elements, as proposed in [Harrison76], is not enough; quantifiers and negation are needed as well. The predicate *remainsvalid* is similar to its use in del_import, it checks for validity after removal of the grant.
This simulation exercises show that the SPE model and an extended HRU model are equally powerful in representing access matrix oriented protection systems. However, a few differences should be be pointed out.

- The access matrix in HRU is primarily used to administer the state of affair, rights are merely used as labels to control the use of commands. By contrast, the SPE instructions incorporate constraints on the flow of access permissions, that is, access flow need not be enforced through defining an abstract machine, a configuration or set of commands, first.
- As mentioned before, the HRU model does not recognize the role of an activator. In the SPE model each action is triggered by a user which is held responsible and it is ensured that the user can not affect those portions of the protection state for which he is not considered an owner. The HRU model does not place restrictions on the use of its define/remove subjects/objects instructions.
- In the HRU model users play a double role, since they are represented as both rows and columns, which make it possible to model the roles between subjects. By contrast, in the SPE model, user and objects play a single role. Relations between users with different roles should be modeled with the region concept and the structure relation.
- The command definition facility provided for HRU does not provide full abstraction instructions, i.e. right identifiers (and commands) can not be used as parameters in the command heading, which prohibits the definition of generic commands to control the flow of rights. Rights (and programs) are represented as objects in the SPE model, which can be used freely as parameters in a SPE program.

5.6.4. Semantics of rights

Rights are used in protection systems like HRU to regulate both the use of objects by subjects, as to regulate the modification of the protection state itself. This difference can be used to divide the class of rights into 'controlling' rights and 'non-controlling' rights. The former consists of those rights used for controlling the modification of the protection state only. That is, the controlling rights are used to distribute the non-controlling rights. The latter regulate the semantic use of objects by the subjects, i.e. application of operations to objects. Separation of the rights according to their role was first proposed by Weyuker [Weyuker78]. It provides a good basis for the analysis of the flow of access rights and leads to a modular protection system. In dynamic systems, where

changes to the protection state are encoded in commands subject to the protection being modeled, the controlling rights form a subset of the non-controlling rights. The elements of the HRU access matrix contain one or more rights taken from a (fixed) set or rights. All rights used to control the modification of the matrix, i.e. those mentioned in the conditions of the commands, can be considered as controlling rights.

In the formal definition of the SPE model no special attention has been given to the different classes of access one might wish to distinguish in real situations, i.e. the non-controlling rights were missing. Rather the one concept, accessibility of an object to a subject, forms the basis for access control. The SPE simulation using an access matrix shows that the binary relations can be considered as controlling rights instead. In this section we show how accessibility can be used to accommodate non-controlling rights, like 'read', 'write', and 'execute' rights, found in other models.

5.6.5. A three-dimensional access matrix

The easiest way to accommodate semantic rights in the SPE model is to associate them as object labels, similar to a capability based approach. However, such an approach works when the right labels perform a similar function only. That is, they are constrained by the same access flow policies. In general this is not the case. Therefore, an alternative view is taken here, based on SPE protection state replication for each right.

Taking the access matrix for a single right as a point of departure, two approaches present themselves to extend the access matrix to accommodate different generic rights. First, the columns can be replicated such that each column contains a single right. In the proposed SPE simulation this means that multiple regions are introduced, one for each right in the generic right set. The subject regions are all made part of the contents of these regions.

A second approach is to consider a three dimensional matrix in which the matrix elements contain controlling rights only. Then each user/object slice in the matrix can be associated with a different non-controlling right. In the traditional 'read, write, execute' case three slices would be needed. The prime advantage of the three dimensional approach, as opposed to the use of a single access matrix (with replicated columns), is a reduction of information considered during protection analysis, because the access rights are grouped by non-controlling right. Moreover, objects and users are not necessarily represented in all slices, as it depends on the semantics. For example, read-only objects need not be represented in the slice associated with 'write'. The second advantage of a three dimensional view is that the infrastructure for access flow and the commands to change the protection state can be grouped per slice. That is, the policy to change the protection state is strictly separated from the protection policy associated with the different operations on the objects modeled.

The notion of slices indicates a third method for access matrix representation. Instead of storing the access rights with the objects, i.e. access control lists, or in the user domain, i.e. capabilities, the access rights can be administered by the functional processor, i.e. server. This approach is of particular interest when both the number of users and the set of objects are large, while the access decisions are simple.

For example, consider a database of documents classified into top-secret, secret, classified, and unclassified, which are accessed by a large group of users. The access control list method requires a long list of permissible users to be associated with the individual documents. By contrast, the capability-based approach requires long lists of capabilities to be distributed to all the users. In the right oriented approach this information is associated with each document class. This approach not only reduces the amount of storage, but provides a means to control the physical location and properties of the system more easily as well.

Viewing the access matrix as a three dimensional matrix does not generate radically new security problems. Instead, partitioning the protection state simplifies a number of security questions due to a reduction of complexity. The following problems are new and should be addressed separately for a given system:

1) The slices; what is the flow of access within a given slice ?
2) Inter slice; what interference can be expected when two slices model different policies ?
3) The objects; what role does an object play in the different slices ?
4) The users; what role does a user play in the different slices ?

The different roles rights play has been given limited attention in the literature. As noted before, in [Weyuker78] an attempt is made to distinguish the roles of rights in the HRU model and it is used to analyze its general safety question. Another example, the Take-grant model, to be discussed shortly, models the flow of access permissions with two rights, *take* and *grant*. In Lipton and Snyder [Bishop79] the interplay between these two controlling rights and the flow of information extracted from documents is analyzed, i.e. an example of the inter-slice protection.

5.6.6. Mapping the three-dimensional matrix to SPE concepts

An SPE realization of the three-dimensional matrix model repeatedly uses the SPE state depicted in Figure 5.6, i.e. the mapping of the two dimensional matrix with one non-controlling right. However, in the new situation each object is represented by a set of SPE objects each representing a different access right. Each SPE object behaves like a 'ticket' in terms of a capability based approach with the observation that the ticket contains a reference to a single right and

object. An example 3-right protection state is shown in Figure 5.19.
Modeling rights in SPE as objects makes the introduction and manipulation of new rights easely. For example, three SPE programs suffice to selectively extend the new slice with users and objects, as shown in Figure 5.18. The command addslice can be applied by any user to create a new slice with owner 'rightowner'. The latter subject is introduced solely to avoid a clash between any real user and the fictive users known within the model. This framework can be extended to enforce new policies dealing with the interplay of rights, such as enforcing that users having execute permission to a file do not have write permission.

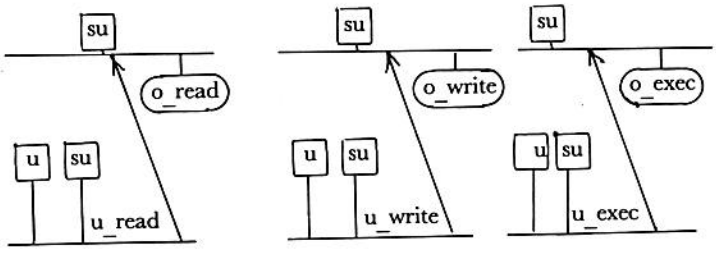


Figure 5.18 Access matrix representation with three slices.

```
command addslice(user, rightname, rightowner)
begin
        add_region(user, rightname)
        add_owner(user, rightname, rightowner)
        del_owner(rightowner, rightname, user)
end
command addusertoslice(user, userregion, rightowner, rightregion)
begin
        add_region(rightowner, userregion)
        add_struct(rightowner, userregion, rightregion)
        add_owner(rightowner, userregion, user)
end
command addobjecttoslice(object, rightname, rightowner)
begin
        add_object(rightowner, rightname, object)
end
```

Figure 5.19 Slice extensions.

5.6.7. The Take-Grant model

The second model simulated is the Take-Grant model introduced by Jones [Jones76] which uses a directed graph to represent the protection state, the nodes of which denote subjects or objects, edge labels denote access rights. Two controlling rights *take* and *grant* are defined, which represent the ability to rewrite the protection graph as illustrated in Figure 2.4.2. The relationship between Take-Grant and HRU has been analyzed by Weyuker [Weyuker78], who gave a HRU simulation of the Take-Grant model as well. As HRU can be simulated with the SPE, we know that Take-Grant can be simulated too with one extra level of SPE programs. In this thesis we also show how this model is directly simulated with the SPE model.

Similar to the SPE, the Take-Grant model distinguishes passive from active entities and associates rights to describe their relationship. Unlike the HRU model rights can be defined between objects as well, which makes it possible to model directories in file systems. The authorization policy of Take-Grant limits the modification of the protection state to active entities, i.e. the elaboration of a controlling right is restricted to users. Consequently, *take* and *grant* rights associated with edges between objects are passive, and can only be the object of transfer by subjects.

These properties imply that objects should be mapped to a combination of region and object in the SPE model. Likewise active entities are represented by a SPE user and region combination. The rights modeled are restricted to the set {*take*, *grant*, Alpha}, the latter standing for a non-controlling right. The region architecture for the simulation is shown in Figure 5.14. Three regions are used as containers for SPE objects to represent the rights. Moreover, each user and object is represented as a region. It is a focal point for access rights available to the user (or object), that is, they represent the nodes of the protection graph. To simplify the description of the SPE commands we use the naming convention <region>/<object>, which stands for the name of a single object called <object> in the region <region>. Each object (and user) is represented by three SPE objects: take/object, grant/object, and alpha/object. Moreover, names for regions, objects and users may be identical, the context being derived from its use. An owner of region X having *take* right to object Y is represented by accessibility of the object take/Y in the region X. All state transformations are issued by the fictive user 'su'. The users are only used for authorization purposes, i.e. a graph rewriting can be requested by active users only. Figure 5.15 shows the mapping of the Take-Grant instructions.

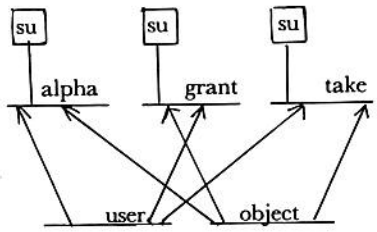


Figure 5.14 Take and Grant SPE state description

```
create_obj(o)        command create_obj(o)
                     begin
                           add_region('su',o)
                           add_struct('su',o,'alpha')
                           add_struct('su',o,'take')
                           add_struct('su',o,'grant')
                           add_object('su','alpha','alpha'/o)
                           add_object('su','take', 'alpha'/o)
                           add_object('su','grant','alpha'/o)
                     end
create_usr(u)        command create_usr(u)
                     begin
                           add_region('su',u)
                           add_owner('su',u,u)
                           add_struct('su',u,'alpha')
                           add_struct('su',u,'take')
                           add_struct('su',u,'grant')
                           add_object('su','alpha','alpha'/u)
                           add_object('su','take', 'alpha'/u)
                           add_object('su','grant','alpha'/u)
                     end
destroy_obj(o)       command destroy_obj(o)
                     begin
                           {compensation actions of create_obj }
                     end
destroy_usr(u)       command destroy_usr(u)
                     begin
                           {compensation actions of create_usr }
                     end
take_alpha(A,B,O)    command take_alpha(A,B,O)
                     auth (A,A)∈OWN
                     pre ('take'/B,A)∈IMP ∧ ('alpha'/O,B)∈IMP
                     begin
                           add_import(A,A,'alpha'/O)
```

```
                    end
take-take(A,B,C)    command take_take(A,B,C)
                    auth (A,A)∈OWN
                    pre ('take'/B,A)∈IMP ∧ ('take'/C,B)∈IMP
                    begin
                        add_import(A,A,'take'/C)
                    end
take-grant(A,B,C)   command take_grant(A,B,C)
                    auth (A,A)∈OWN
                    pre ('take'/B,A)∈IMP ∧ ('grant'/C,B)∈IMP
                    begin
                        add_import(A,A,'grant'/C)
                    end
grant-alpha(A,B,C)  command grant_alpha(A,B,C)
                    auth (A,A)∈OWN
                    pre ('grant'/B,A)∈IMP ∧ ('alpha'/C,A)∈IMP
                    begin
                        add_import(A,B,'alpha'/C)
                    end
grant-take(A,B,C)   command grant_take(A,B,C)
                    auth (A,A)∈OWN
                    pre ('grant'/B,A)∈IMP ∧ ('take'/C,A)∈IMP
                    begin
                        add_import(A,B,'take'/C)
                    end
grant-grant(A,B,C)  command grant_grant(A,B,C)
                    auth (A,A)∈OWN
                    pre ('grant'/B,A)∈IMP ∧ ('grant'/C,A)∈IMP
                    begin
                        add_import(A,B,'grant'/C)
                    end
```

Figure 5.14 Mapping TG instructions to SPE commands

To illustrate, the take-alpha right rewriting rule, which is applied to nodes A,B,and C with *take* labeling the edge (A,B) and alpha labeling (B,C) is simulated with the SPE instructions add_import and the object naming convention. The authorization condition ensures that a user is requesting the state transformation, the command condition checks for the proper TG rights.

The security questions analyzed within the context of the Take-Grant model are limited in scope. The prime question considered is whether a finite sequence of graph rewritings exists such that a subject u gains access to an object o. This security question has been shown to be decidable in linear time in the size of the protection graph. In fact, this security question is analogous to path finding in an undirected graph.

The result of this simulation that it is indeed possible to build multi-level security systems with different protection policies using SPE. In particular, in this example any user represented in the SPE state is able to obtain any access right using add_import. However, as all actions are encapsulated by commands this weakness in protection is avoided. Similarly, it illustrates how a system owner can be used to regulate the creation and removal of users(objects) without taking part in the access flow.

The Take-Grant model definitions [Jones76] do not provide an abstraction mechanism. In [Jones78] an attempt is made to extend the model with behavioral properties, i.e. procedure mechanism. This machinery can be used to simulate the SPE model, because it allows for testing . However, it requires a single universal node having *take* and *grant* access to all users, regions, and object nodes, as well as the SPE specific rights on all of these. (In the Take-Grant model disconnected components can not be combined using the graph rewritting rules.) Then access relations between two nodes can be described by edges labeled with the relevant SPE relation names, while the applicable procedures can be controlled by the behavior rules. The effect is that the distance of edges in the simulated SPE model is at most two, for the 'universal' node connects all. No theory has been developed around the procedural concept in Take-Grant to derive protection properties for a group of procedures.

5.7. Summary

In this chapter we have introduced a sample instruction set for the SPE model and showed that it satisfies the properties set forth in the previous chapter on instruction sets, i.e. the `set` is *well-defined.* In Section 5.4 the compensation property is extended to cover the notion of revocation. Revocation prescribes a policy to undo actions in general. Two algorithms were presented to realize the revocation sequence with different objectives and implementation costs. Subsequently, instruction set partitions were studied to realize protection states governed by derivable security policies. In particular, a set of predicates parameterized by initial state and instruction set characterize reachable states. In that context the notion of stealing has been given a formal definition and an algorithm to determine potential theft and the conspirators has been indicated.

In the last part of this chapter we have indicated the potentials for a multi-level security system based on the SPE model and used this approach to simulate two well-known theoretical protection models. The first simulation, the Harrison-Ruzzo-Ullman model, emphasized the approach taken by the SPE model in viewing access control protection as a state transformation invariance property rather than a state property or a state reachability property. Moreover, it is shown that both protection systems are equally powerful by simulating the

Turing machine. The simulation of the Take-Grant model showed how the SPE model should be extended to accommodate semantic rights.

6

AN SPE PROGRAMMING ENVIRONMENT

In the previous chapters we have introduced the SPE model and instruction set. The protection model defines an abstract machine and has been show to be a good basis for multi-level protection systems. In particular, its emphasis on behavioral properties made it a vehicle for the description and analysis of access control problems in computer systems.
The SPE model as a theoretical tool has limited use without an indication of how it can effectively be applied to the construction of machines, how it relates to trends in their design, and how the model realizes protection issues in a programming environment. These topics are addressed in this chapter.

First, a sketch of a (virtual) machine architecture based on the SPE model is presented. In the design of this machine we ignore the large collection of details to be dealt with in reality. Instead we concentrate on its global characteristics and rely on the large body of literature in the area of operating systems design to fill-in the details. The approach taken is based on a distributed architecture of loosely coupled virtual processors. Each processor provides a security kernel implementing a reference monitor for the SPE model.

Second, the SPE model is compared with the static scope rules of the language PLAIN. It is shown that access control within SPE forms a framework for enforcing visibility rules in the programming language. The protection model is used to extend the scope rule mechanism to define and enforce access control at compile time in collections of PLAIN programs. This leads to a consideration of a module interconnection facility based on the protection model which encompasses a framework for project and version management.

The last section of this chapter describes access control enforcement in the PLAIN programming environment as achieved by a symbiosis of the SPE

machine architecture with a PLAIN interpreter. In that context procedure calling, variable declarations, and variable sharing are analyzed. This leads to rules for a secure compiler/interpreter SPE implementation.

6.1. The architecture of an SPE machine

6.1.1. Region managers as basic building block

The architecture of the SPE machine is based on an aggregation of virtual processors. One such processor is associated with each region in the protection state and referred to as Region Managers. A region manager cannot be equivalent to a single hardware processor, because this would make dynamic extension and reduction of the system, as provided for in the protection model, impossible.

Each region manager can be broken down into three functional components: a data type processing function, an SPE processing function, and a network processing function. The network processing function provides facilities for region managers to communicate using (various) communication media and protocols. The SPE processor is dedicated to the administration and enforcement of the SPE protection policies. The data type processing function provides object class manager(s). Each functional component presumably has exclusive access to hardware components, such as cpu and memory banks. The tasks and options of the machine components are discussed separately.

network processor(s) <----> [message queues, protocols]

SPE processor <-----> [SPE protection information]

Type processor(s) <-----> [process text and data]

Figure 6.1 SPE machine components

The choice for a functional decomposition of the SPE processor stems from concerns about protection *separation* and *mediation*. Cluttering functions dealing with protection administration and processing has long been recognized as highly insecure. Early attempts to resolve the protection problem led to separation of activities in a few different processor states, i.e. kernel-system-user states. Unfortunately, operating system software soon became too complex to guarantee reliability and enforcement of the protection policy in the multi-state processor systems. In turn this led to a new design criterion to separate, as much as possible, access control enforcement from normal processing functions.

All access rights, whether related to IO devices or memory, are encoded in special (hardware supported) objects called capabilities, and handled by a *reference monitor*. The reference monitor is the sole system function able to interpret the capability objects and is normally implemented by a hardware/software mechanism, called a *security kernel*. The reference monitor has been used as a basis for many secure systems. For a survey see [Landwehr83, Levy84]. The smaller size of the protection code in a reference monitor makes verification attainable as shown by Popek [Popek80], yet full scale application of this technique is not likely going to happen for some time .

The SPE machine architecture considered, based multiple processors coupled through communication media into a computer network, aggravates the security problems. The mediation on protection issues, as implemented by a security kernel in the single site system, becomes more complex, because protection information is spread among multiple sites. Moreover, access control requires cooperation and trust among sites, while the communication media impose their own security problems, such as piggy backing, wire tapping, etc.

In the area of secure distributed operating systems design emphasis is placed on the network processing activities and reliable exchange of information. The communication aspects has led to several communication models and protocols. One example is the ISO/OSI model [Zimmerman80], a multi-layered description of inter-site communication. With respect to reliable and private communication, the model relies on message encryption to ensure a secure communication channel between different user processes. A public key encryption scheme can be used for this purpose [Diffie76].

The ISO/OSI model does not explicitly address the access control aspects implied by the architectural choices. It assumes that entities at the same level in the hierarchy can establish a communication path. This, however, requires peer authentication and an authorization mechanism between the different communication layers. Unfortunately, the model explicitly allows for layer bypassing, which may result in overloading the communication net with messages not intended to reach the net at all, or the ability of higher levels in the ISO/OSI model for illegally tapping the message streams.
A security kernel approach can be used to improve security in an ISO/OSI environment as well. For example, the presentation layer can be extended to include authentication functions to establish reliable application level communication. Each message sent to the application is authorized first. Conversely the presentation layer authorizes all requests for network facilities, reducing contamination of the communication media (Assuming that no bypassing of the presentation layer is allowed). This implies that layers should not be able to decode and use the contents of the message other than required for the task associated with that level.

The designer of a distributed system should address the security properties of the individual sites as well, since the insecurity of a single site results in insecure

communication paths. To circumvent these problems two approaches have been proposed in the literature:

a) explicitly allow for insecure components and

b) use a distributed capability object type.

The former approach has resulted in the design and implementation of a multi-level security system around a number of UNIX machines [Rushby83], where individual sites may be insecure. The network functions only guarantee that inter-site transfer obeys a multi-level security policy. Approaches taken by designers of capability based systems can be classified further into:

1) make forging capabilities a probabilistic problem [Tanenbaum81] or

2) ensure that distributed hardware is trustworthy in the management of system wide capability objects [Wulf74].

The SPE machine architecture sketch is based on the premises that each node in the network can be made secure and that reliable communication is achievable. That is, we assume that the hardware is fail safe or has an acceptable low error rate. The design guidelines raise technical problems of naming, network topology, type management, authentication, and object administration, each of which is now addressed separately.

6.1.2. A naming problem

The first problem to be solved for the SPE machine is its naming policy. The SPE protection model requires unique names, which in turn requires a solution for the distributed environment envisioned. System wide naming has been a problem to be solved in each distributed system and therefore it suffices to choose an efficient and reliable one. Different strategies have been adopted, which may be classified as follows:

- Centralized naming
- Decentralized naming
- Context naming
- Probabilistic naming

Centralized naming in an SPE machine requires a single region manager, called the name space manager, to be in control of the name space. Whenever a new name is required this manager is consulted. The advantage of the centralized naming mechanism is compactness in both name representation and new name generation. The prime disadvantage of the centralized naming mechanism stems from reliance on the proper working of the communication network. When the name space manager is disconnected, all work comes to a halt.

The disadvantage of centralized naming is solved by generating unique names based on local information only, i.e. using distributed counters. One way to handle this is by using a system clock (or counter) extended with the processor's serial number [Wulf74] or in generating system wide unique transaction numbers [Reed79].

Context or relative naming differs from previous methods by dropping the name uniqueness requirement. Instead, names are bound by the context in which the are used. A good example of context naming is a hierarchical file system, like the UNIX file system. File names may be re-used in different directories, name decoding takes place in the current working directory or an explicitly defined search path through directories. Another example is the use of identifiers in scoped programming languages, global definitions are shielded by local definitions.

Probabilistic naming is used in distributed systems based on the paradigm that access control is a probabilistic problem. Whenever a new name is needed, one generates a bit pattern of sufficient length at random. Uniqueness can not be guaranteed, but the probability that a names clashes with one already in use can (under the assumption that drawing random numbers in a distributed system can be realized).

In the subsequent use of the SPE model implementations we assume that all names are unique within a given context. Wherever it is necessary to distinguish them, they are prepended with the name of the context, i.e. the defining or enclosing region.

6.1.3. The network processor

The top level component of the SPE machine is formed by the network processor, which isolates the communication aspects from the functional aspects of the SPE machine. Its sole task is to establish reliable and secure communication links with other network processors. Reliable in the sense that all messages sent reach their destination unaltered and, conversely, all messages addressed to the region manager are delivered to its security kernel. The communication link established should be made secure by making provisions against wire tapping, piggy backing, masquerading etc. Moreover, the security kernel has to safeguard the contents of the messages against tampering and it has to perform remote security kernel authentication.

The separation of transport activities from access control makes it easier to verify the functionality of the network processor, because one does not have to prove that the buffering schemes, routing, temporary storage, and message logging facilities used by the network processor are secure as well. Moreover, moving authentication to the security kernel dissociates the 'site' aspects from the communication software. Altogether, the network processor can be considered as providing the SPE processor facilities at the presentation layer level in the

ISO/OSI model.

For the design of the SPE machine the topology considerations do not differ from other distributed operating system designs. One aspect of concern is that in the subsequent use of the SPE machine model we assume a fast local area network, although this is not a prerequisite for a proper working of the system.

6.1.4. The type manager

The lowest level of the SPE machine provides the conventional operating system functions, i.e. a limited class of objects, types and operations such as devices, semaphores, processes and files. Rather then considering a single piece of software providing all facilities, resulting in a baroque and complex system, we assume that the operating system is a collection of type managers or abstract data types. A type manager is a piece of software (hardware), which supports the creation/deletion and manipulation of a particular class of objects. The notion of type managers comes directly from the area of software engineering and has been introduced in the area of operating system design by Jones [Jones73].

From a protection point of view a type manager denotes the boundaries considered relevant for access control. Authorization takes place during initialization of a procedure call. Subsequent execution of the procedure is assumed to be free of any access violations. Realization requires hardware support, i.e. memory segmentation, as available on commercially available systems.

To illustrate the concept in the context of the SPE machine, one of the type managers available within each region manager is a segmented virtual memory data type. This data type splits a virtual address space into a number of distinct segments and attaches read, write, and execute capabilities. The operations made available are besides reading, writing, and execution of the segment, as well as its creation and destruction. Segment creation delivers the name of the segment for subsequent references.

The segment type facilities are used by user defined type managers in the same region manager to realize, for example, a database of records. The database type manager uses one or more segment names provided by the segment manager for record representation. Incidentally note that these type managers view a piece of memory from two different perspectives. The SPE machine should provide facilities to extend the class of type managers for this purpose and can therefore be considered an open system. However, to simplify the subsequent discussion we assume that a single user defined abstract data type is associated with a given region.

6.1.5. The SPE kernel

The central component of the virtual SPE processor is the security kernel, which implements a reference monitor. This means that all requests to execute an operation provided by one of the type managers or to access an object locally or remotely is checked against the protection policy administration. For this purpose, the security kernel contains part of the system wide SPE protection state. In particular, it contains the SPE names for all owners associated with the region being managed, the objects defined within the region, objects exported and imported from the region, and the structure relations with regions in both its environment and contents. Replication requires access control information to be consistently stored in all sites and that all security kernels should trust one another after proper authentication. Moreover, it implies that the resulting system should be homogeneous at the security kernel level.

The tasks assigned to the security processor can be further divided into: (local and remote) procedure execution, maintenance of the protection state information, mapping instances of the abstract data types and operations to SPE names, and authenticating users of the region manager. Each of which is discussed shortly.

6.1.5.1. Procedure execution

A procedure invocation is considered local when its definition, i.e. the corresponding SPE object, resides in the same region manager as from which the request originates. Otherwise the call is considered remote. To illustrate the actions taken when a local call is made, assume a program named Job issues a request to execute the procedure Insert supplied by the type manager Dbm. The actions performed by the different SPE machine processing functions are:

a) Job sends the message Insert with parameters and user held responsible to the security processor.

b) The SPE processor checks the SPE protection information, i.e. is the procedure Dbm/Insert being called defined as a local SPE object?

c) If authorization succeeds then the message is sent to the object Dbm/Insert for further processing. Otherwise notify denial.

d) The type manager procedure Dbm/Insert decodes the parameters, executes the procedure definition, and notifies success to caller.

In this scheme mapping object names and access control enforcement take place within the SPE processor. The representation of objects, however, is hidden by the type managers. All communication between procedures is done in terms of SPE names, which is equivalent to the approach taken in capability oriented systems. In a sense the SPE security kernel can be seen as a capability directory manager.

6.1.5.2. Remote execution

In the SPE machine, procedure execution is limited to the context of their definition, rather than their call. This implies that necessary access rights should be transferred to the remote region safely. The alternative approach is addressed in the next section. For the moment it suffices to note that such a scheme ensures that private access rights of an operation are kept private. Remote execution differs from local execution in the involvement of the network processor to transport messages.

For example, assume that the type Dbm is defined remotely, i.e. in another region manager. Then calling the routine Dbm/Insert from Job results in the following actions:

a) Job sends the message to the security processor of the region manager it belongs to for authorization and execution.

b) The SPE processor checks the local protection information, i.e. is the procedure Dbm/Insert accessible as an SPE object and is the user an owner of the region?

c) If authorization succeeds, the message is given to the network processor for transport to the region manager where the procedure is defined as an SPE object. Otherwise the caller is notified of denial.

d) Upon receipt of the message by the region manager for Dbm, the message is propagated to the type manager.

e) The type manager procedure Dbmtype/Insert decodes parameters, executes the operation, and returns information.

Note that distribution of the SPE access control information means that local inspection to authorize requests is sufficient. Provided though that information distribution can be done in a secure way. This represents a departure from capability-based systems where the type manager receiving the capability authorizes a request.

6.1.5.3. Mapping names

The execution templates above ignore the mapping problems introduced by the separation of the security kernel from the type manager. Each has its own memory and they are assumed to cooperate. The kernel, upon receiving a request to execute a procedure, should be able to determine the name of this procedure within the type manager name space (i.e. its segment and entry point). On the other hand a type manager should be able to determine the name of an SPE program to request its execution or to determine the name of an SPE object to access it. Both situations call for a directory, binding the name spaces of the two subsystems.

As message flow is funneled through the security kernel, the directory is best placed under control of the kernel software. Selection of an object (or an SPE user) is triggered by a type manager by sending an alias for the SPE object to the kernel. Conversely, procedure parameters received by the kernel are mapped to their aliases before being shipped to the type manager.

The advantage of this approach is that it limits type managers in naming valid SPE objects (and users), they can only name objects (users) accessible within their security kernel. Moreover, using this approach makes it possible to define additional aliases for the same object. Often they reflect an address in the (permanent) virtual memory associated with the type managers' process. Although a type manager is free in the construction of the aliases, actual use is limited by the directory structure kept in the kernel. An actual implementation should address problems like how both kernel and type manager become active upon receiving a message, and the size and structure constraints imposed on the type manager aliases. In summary, instruction invocation in this architecture is seen as sending a message with the appropriate access rights to an object, in this respect it is similar to systems like [Wulf74, Goldberg] or to a communication **port** in [Tanenbaum81].

6.1.6. Changing the protection state

Changing the protection state administration is not different from calling routines provided by any type manager. When a change is requested a message is constructed for one of the SPE programs with SPE parameter names. However, the message is interpreted directly by the security kernel. The instructions add_owner, add_object, add_region, and add_export can be handled locally. Communication with the region managers in the environment is sufficient for handling add_struct and add_import instructions. Other SPE instructions, such as the deletion instructions, require conversation with the region managers in the environment or beyond. (chapter 4 and 5) The actions are illustrated with a few examples.

Whenever a new object is added to a region the security kernel generates a unique SPE name and constructs an entry in the directory to enable the type manager to denote this object in the future. Export of SPE objects are handled as specified in the SPE model, the user issuing the request is checked as one of the owners associated with the region manager, access of the object being exported is checked, and finally the EXP relation is updated.

The instruction add_region requires support by the underlying machine, because a new region manager should be constructed. One possible implementation is to copy the network processing functions and built-in type managers of its ancestor into the new region manager and assign the user responsible for the call as sole owner.

When access rights are imported or when structure relations are created conversation with other region managers is required. The former requires the object to be accessible in one of the regions in the environment from which the call arises, and the latter requires that the invoker of the call owns both regions involved. For example, consider importation of an object, i.e. add_import(u,R,o), used within a program running in the region manager R. First, the security kernel obtains the region names of those defined in its environment. Each of these regions is sent the request add_import(u,R,o). Say region manager S receives it, then its security processor recognizes the message as an inter region manager request (looking at R), which means that no authorization of the user is needed. Instead, the region manager R requests access permission to an object accessible in S based on the availability of a structure relationship with S. Accessibility of the object in S and the (R,S) structure relationship is checked to ensure that indeed R is allowed to obtain access. An acknowledge message is returned indicating success or denial. When R receives a successful reply, it modifies the protection area to indicate that the object has been imported. Upon denial other regions are checked or the action is sent a denial or access violation.

Another example is the making of a structure relation, which requires modification of the protection area in both security processors involved. Assume that security processor for R tries to establish a structure relation with security processor S on behalf of user u. The first action of R is to authorize u, then the message add_struct(u,R,S) is sent to the processor for S, which authorizes the request and changes its protection area accordingly. Upon receiving a successful reply R can change the protection area as well.

6.1.7. Authentication in the SPE machine

In the SPE model extensive use is made of the concept of user entities. Each state transformation requires the user responsible for the request to own the regions affected. The representation of users within the SPE security kernel requires a solution for naming, authentication, and introduction of new users as well. In essence, the naming problem for users is not different from the naming problem of objects. Each new user should be represented by a (globally) unique SPE user name within the kernel. Moreover, to allow type managers to construct messages for protection state management, the user names should be supplemented with an alias.

Authentication involves a process where the identity of a user is established in a 'dialogue'. In essence the authentication process has access to the list of user name aliases and, upon receiving proper information (passwords) from the user (at the terminal), changes the activator associated with the command interpreter or starts a new interpreter process for the user. One approach is to dedicate a single region manager RM to handle all authentication requests. The

advantage is simplified authentication information administration. The disadvantage is that all region managers should trust the authentication manager. The role of this RM can be compared with the role of a notary in reality. An example of such a scheme in a PLAIN programming environment is given in [Riet80]. An authentication scheme using encryption within a distributed environment is described in [Needham78].

A small problem arises during the authentication process itself, especially when conversation is required with other RMs. During this process, an SPE user name is required as invoker of the actions. Clearly, the user being authenticated can not be this invoker. The solution to these and similar problems is supported in SPE by considering co-owners of a region equally powerful. This property can be used to introduce an SPE user name at creation time of the region manager, which is unique to the region. When this SPE user is not augmented with authentication information, no changes with respect to the protection policies result. This way the authentication process never derives it. However, it can be used to represent actions taken by the region manager on its own initiative or as a substitute for the user for whom it is interpreting the command.

6.1.8. Summary

In summary, the SPE model can be used to direct the design of a distributed system using existing hardware/software technology. In resulting system necessary protection information is available on the spot, i.e. the security kernels, which makes access authorization cheap. Moreover, separation of the security aspects from the type managers and communication protocols leads to a more orthogonal system design. Type managers (i.e. servers) are freed from the storage, decoding, and maintenance of the access rights (if they so desire). The communication media are shielded from the type managers using the same protection structure, avoiding malicious communication software and type managers gaining access to the interkernel communication.

As mentioned in the introduction of this section, the description of a complete SPE based machine architecture necessarily requires attention to many more detail, such as multi-programming a region manager, concurrency conflicts in access authorization and revocation, etc..

To summarize the global design of the SPE machine, we compare it with the design constraints given by Saltzer [Saltzer75] for the construction of secure systems:

1) Processes on an SPE machine execute in **small protection domains** by definition.
2) The number of instructions used to implement access control is small, i.e. the SPE instructions.

3) Every access is checked against the most current policy by the SPE security kernel.

4) The design is open and new type managers can be introduced readily to describe and enforce new protection policies.

5) More than one condition determines access permission, i.e. local administration and invoker identity.

6) Access control flow channels between procedures are explicitly defined and limited.

Therefore, the architecture presented here around the SPE protection model provides the necessary infrastructure to accommodate a large class of protection policies, which may include insecure data type processors as components as well.

6.2. Visibility in high-level programming languages

Central issues in the design of high-level algorithmic programming languages are the description of its program text structuring primitives, the visibility of identifiers at various points in the program text, and classification of objects by data types. The structuring primitives, like block structure, modularization, and visibility rules give the programmer precise control over the name space management: the set of names a programmer may define and use at any point in his program. Name space management results in economy of name space usage by resolving naming ambiguity by context using the visibility rules.

Primitives for text structuring and visibility rules can be traced back to the early days of High Level Languages (HLL). For example, Fortran allows for grouping of code into separately compiled routines. Objects are declared local to a routine using both explicit and implicit typing. The prime name space management primitive is the (labeled) common data area, which, besides the parameter mechanism, is the only means for routines to communicate within the language framework.

The late fifties mark the introduction of block structured languages [Wijngaarden69], where two units are written in the context of the same declarative part if they are to share the visibility of some common outer objects. Its successor, Algol 68, is a significant mark in the development of block structured languages, because it generalizes the typing mechanism and visibility rules of ALGOL 60. Its definition is rich enough to describe formally visibility rules as well [Wijngaarden69].

Languages developed in the seventies extended and refined the structuring and visibility rules in two directions: encapsulation, originating with the class definition in SIMULA, and control of name visibility. Encapsulation is used to describe not only the structural aspects of objects in a data type definition, but include its behavioral aspects as well. Encapsulated data types are manifested as

textual units and are divided into an interface specification and implementation body, also called modules or abstract data types. In addition, the programmer is given tools to explicitly state the transfer of name visibility between different lexical groups in the form of import/export lists [Wirth80, Liskov81, Lampson77] or restrict/use clause [Ichbiah79].

6.2.1. Relation between access control and visibility

Object visibility in programs has long been related with access control. An early example is the extension of COBOL with access classes to restrict the use of record fields [Conway72b]. Morris [Morris73] claims that types in HLL should be interpreted as a language mechanism to implement authentication and access authorization. Authentication ensures that any value supplied to an operation is consistent with the type expected. Access authorization in this view means that any operation applied to a value is meaningful for that type. Jones and Liskov [Liskov78] take a different approach by extending a HLL with facilities to restrict the behavior of an object when it is declared to support a subset of the operations defined for the type. Their approach is based on mapping a capability-based protection scheme into a high-level programming language, using generic procedures to handle objects with different access behavior.

The main benefit of enhancing a language with explicit access control information is enhanced software reliability and manageability of the development environment. Software reliability is gained through the ability to state the intended use of objects in advance and to control/restrict the propagation of object visibility. These language based control primitives, compared with operating system access control, results in finer protection granularity. Moreover, a language providing separate compilation augmented with access control information results in a better software engineering environment. CLU [Liskov77], PLAIN [Wasserman81] and Ada [Ichbiah79] are examples of such languages.

Inclusion of access control primitives in a HLL alone does not solve all protection problems encountered. For one thing, a model to study the range of protection is needed and this model should be independent of the semantics of the rest of language. For example, in [Liskov78] the Take-Grant model is used in a language framework, addressing assignment and parameter passing only.

The access control issues in a HLL are studied in this thesis using the programming language PLAIN. PLAIN was designed in the late seventies to simplify the construction of interactive information systems, a goal similar to languages like ASTRAL [Bratbergsengen79] and Pascal-R [Schmidt77]. It contains most features of contemporary high-level programming languages. In addition, it incorporates full fledged relational data base facilities and pattern matching mechanism, to ease the construction of dialogues. PLAIN follows the trend in providing powerful data structures and control primitives. Being

designed late, it could rely on experiences gained in the 'use' of the languages [Wirth80, Liskov81, Lampson77]. After a short introduction of the lexical language elements, the SPE model is used to obtain a more formal basis of the access control issues in this language. A prototype compiler has been implemented at the University of California San Francisco.

6.2.2. Scope rules in PLAIN

The syntactic objects dealt with in a PLAIN program are:

program units (program, procedures, functions, handlers)
type definitions (simple, structured, database, modules)
variable declarations
exception definitions
pattern definitions
constant definitions

Syntactic objects can be referred to by the identifier assigned to it as part of its declaration/definition. The region of text where an identifier or name is known with a single meaning is called its scope, which is delimited by the closing bracket of the program section in which it is declared or defined. A scope is associated with an identifier and not with keywords of the language. Consequently each introduction of a name can be considered starting a new scope. A procedure name is both known within the procedure definition and in the body of directly enclosing program units.
Economy of name space allows the same identifier to be used for identifying different objects, thereby shielding its previous definition i.e. disallowing access. Renaming is allowed for objects defined at an outer definition or declaration section only.

Aside from hiding definitions by identifier reuse, the programmer is given tools to control the visibility of identifiers using a refinement of the scope mechanism. PLAIN distinguishes two scope types: open- and closed- scopes. A program, a module type and a program unit definition (procedure, function, or handler) form closed scopes; the others are open. Open and closed scopes affect the visibility of identifiers defined in the enclosing scopes as follows:

In a closed scope, an identifier is accessible if it is:
- declared/defined in that scope
- accessible within the enclosing scope, and imported into the scope
- a formal parameter of the type or program unit definition in which the identifier appears, or
- declared as being an external object in the main program in which the scope is enclosed

In an open scope, an identifier is accessible if it is:
- declared/defined within that scope, or

- accessible within the enclosing scope.

Basically, a closed scope makes all identifiers known within the enclosing scopes inaccessible unless explicitly requested through an import clause.

6.2.2.1. Pervasiveness

From the standpoint of programming practice straightforward application of the open/closed scope rules is cumbersome, because it requires each procedure (type,program) to enumerate all identifiers used in advance. Recognition of this effect led to the definition of pervasive objects in EUCLID [Lampson77]. Pervasive objects are not affected by the scope type, they are accessible in all inner scopes without an explicit import request. Non-pervasive objects should be made accessible within the inner scopes by means of the **imports** statement.

The question still remains as to which language constructs should be made pervasive and which not. In PLAIN all names associated with definitions, i.e. constants, patterns, and exceptions are pervasive, while all others, i.e. files, variables, and program units, are non-pervasive. This separation is based on the premise that pervasive objects are not "mutable". Mutable objects are those objects in the PLAIN program which can be changed using assignment or implicitly assigned in a parameter list. (Unfortunately, the latter makes program units "mutable" as well)

6.2.2.2. Visibility restrictions

The closed scope in combination with the **import** statement, allows the PLAIN programmer to describe the use of objects in the context of their use, i.e. "what object is used here". The complementary primitive to limit the context where a mutable object can be made visible, the **restricted to** clause, embodies the "where an object may be used" concept. The **restricted to** clause provides a means to restrict the number of closed scopes in which an object can successfully be imported, but controls the first of a nested sequence of imports only. For example, the variable a in Figure 6.2 is restricted to the procedure walk. Thus a can not be used in the procedure print, although a is defined globally. The procedure walk can import a, as can all of the procedures defined locally to walk.

```
program prog1;

procedure search;
var b:integer;
    a: integer restricted to walk;
    procedure walk;
    imports a:readonly;
    var c,d:integer;
    begin
      ...a...c..d
    end
    procedure print;
    begin
      ...
    end
begin
  ...
end

begin
  ... body of program ...
end.
```

Figure 6.2 Example PLAIN program structure

To complete the introduction of the access control primitives in PLAIN we should mention the ability to limit the use of mutable objects upon importation to either **modify**, **readonly**, or **invoked** access. Procedures can be imported as **invoked**, variables as either **readonly**, or **modified**. A modifiable object can be restricted to readonly, which restricts further import to readonly as well.

Access control primitives in PLAIN are limited to compile time issues only, i.e. static properties are controlled. The qualification **invoked** does not limit the use of an imported procedure, because the single operator defined for a procedure is invocation. The use of **readonly/modify** primitives for access control is limited too, because it is restricted to lexical objects rather than an attribute of a variable. Furthermore, the access attributes are restricted to entire variables and cannot be applied to structure components.

6.2.3. SPE as a framework for PLAIN object visibility

In this section we show how the visibility aspects of algorithmic languages like PLAIN can be translated into SPE terminology, which makes the protection model a reference point of compiler implementation and verification.

The regions in the SPE model are repositories of objects and provide boundaries on the accessibility of objects, which is best modeled by associating a different region with each closed scope in the language. That is, each program unit definition, program and (module) type introduces a new region. The nesting of closed scopes in PLAIN is mapped to the structure relation of SPE, which results in a hierarchical region structure.

The next step is to associate each PLAIN object definition with an SPE object. The SPE object is assigned to the region associated with the scope in which the PLAIN object is defined/declared. Note that a procedure definition is both represented as a region and as an SPE object. The former represents the scope, the latter the entry point. Applying these actions to the example program of Figure 6.3 (ignoring the **restricted to** clause) gives the following graphical representation. To resolve ambiguity in naming, regions are named after their PLAIN occurrence, enclosed in <> brackets.

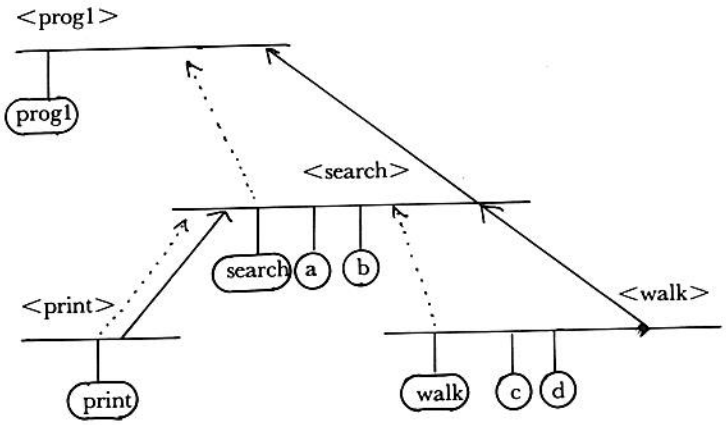


Figure 6.3 SPE state description of prog1

A region is associated with each closed scope, depicted by a line and the name of the scope. Within each closed scope, i.e. region, objects representing the variables and the body of the unit are defined. The name of a procedure is exported to its environment to make it visible to other procedures defined at the same lexical level. The contents of the SPE objects is shown below.

Object	Contents
prog1	**program** prog1; **begin** .... body of program ... **end**
search	**procedure** search; **begin** ... **end**
a	a:integer
b	b:integer
walk	**procedure** walk; **begin** ...b...c...d **end** **end**
c	c:integer
d	d:integer

The region/object structure obtained from this mapping defines the visibility relationships only. We should show how a compiler interprets the various scope rules and statements in the context of the protection model. That is, it is necessary to define the meaning of identifier hiding, importation, pervasiveness, and restricted access.

Identifier hiding is solved in SPE by using context naming, as assumed in section 6.1.2. Importation of mutable objects is interpreted as issuing an import, which succeeds if and only if the object imported is accessible within the environment. Export of an object, as available within module types, models the concept of making 'globally' known objects defined. The semantics of pervasive objects is simple. For each subregion of the object's defining region an import is generated recursively. Since the structure relation forms a hierarchy, pervasiveness makes the objects known within the complete subtree. The **restricted to** clause divides the collection of subregions of the object's defining region into classes with similar access permissions: those where the object may be imported and those where the object may not imported. This situation can be modeled within SPE using *filter* regions. For example, consider the declarations:

var a: integer;
b: integer **restricted to** proc2;

This leads to a graphical notation as shown in Figure 6.4, where two regions are introduced to control the environment of procedures proc1 and proc2. Procedure <proc2> is limited to object a, while <proc1> can import both object a and b.

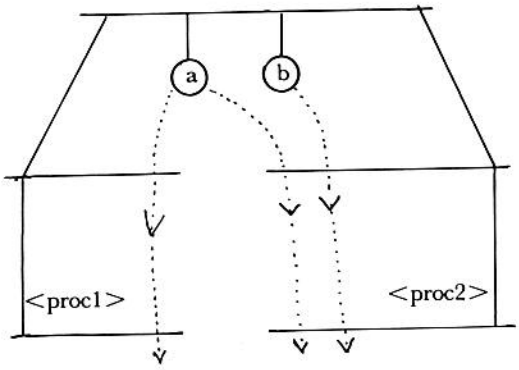


Figure 6.4 Mapping restricted to access

Note that a different filter region is required for each access group involved and that the environment is defined by the structure relations.

6.2.4. Summary

Mapping PLAIN visibility aspects to the SPE model equates identifier visibility with access control. The model describes the actions taken by the compiler upon the interpretation of the scope related actions and can serve as a basis for implementation verification. Moreover, the SPE model suggests a number of improvements and clarifications to the informal language definition [Wasserman81] in the areas:

a) export of mutable objects
c) exception identifiers
d) **restricted to** for program units
e) the role of externals.

The export directive allows operations defined for a module type to be exported. However, the language report requires all operators exported to be defined in the operator definition section (Sec 6.5). This requirement precludes export of operations of module types defined local to the module type considered. A generalization is gained by considering all these operators to be defined implicitly in the operator section as well. For example, assume that the module stack in Figure 6.4 defines the concept of a stack element locally. Then all operations defined on stack elements should be exported to the environment of stack.

Exception identifiers don't follow the identifier hiding policy. In every closed scope where they are is used, they should be defined. In terms of SPE this means that multiple objects are used to represent them, but this makes the resulting SPE state unacceptable. In particular, it is very difficult to analyze and control the scope of exception identifiers. A solution would be to consider

exception definition equivalent to either variable declarations or constant definitions. Then, however, they should follow the same importation rules.

The **restricted to** clause can be thought of as implementing SPE filter regions. Considering program units as "mutable", subject to import manipulation and thus filtering, requires that the **restricted to** clause to be applicable to program units as well. Moreover, to benefit from the filtering mechanism at all levels in the program, the **import** clause should be extended with the **restricted to** clause as well.

External definitions can be seen as a specification of the environment of the program's region, or the program's prologue. All mutable objects should be named in both the **external** clause and a global **import** statement. However, the latter is superfluous.

In the mixture of (non) pervasive identifiers, **restricted to** clauses, and the two scope variants, PLAIN differs from similar languages like Pascal and Ada. In Pascal all scopes are open, while in Ada all scopes are open unless explicitly closed and importation is necessary for modules only. The rationale for using closed scopes is founded in the application of "the need to know privilege" from protection methodologies. Phrased alternatively, a program unit should not gain access to objects unless explicitly granted.

Observe that in the SPE model the protection issues are related to the actions of users as they are associated with regions, that is, users can affect regions when they own the region only. Normally when compiling a program there will be one user. Thus the security questions addressed by the SPE model, such as "who can gain access to an object", translates to cross reference information on the program, i.e. in what scopes is an identifier visible. Moreover, the use of these features supports the construction of reliable code by making visible more of the intended behavior.

6.3. Building information systems with PLAIN

Construction of large information systems requires more than the programming language PLAIN. A well-integrated environment is needed, which incorporates both tools for the construction of systems, and a methodology for their analysis and design, supported by automated tools. A requirement specification for such a programming environment has been described for the language Ada [Buxton80], which includes a compiler, syntax driven editors, (symbolic) debuggers, cross reference tools, and linkage editors. The methodological consequences of Ada has been given limited attention in the literature so far.

By contrast, the programming language PLAIN was developed in parallel with User Software Engineering methodology [Wasserman82b, Wasserman79a]. The advantage of such an approach is a better integration of the software tools with

the techniques to apply them to the construction of interactive information systems. The automated tools to support the methodological aspects are under development at the University of California, Berkeley [Rubin84], which uses the relational database system Troll developed at the Vrije Universiteit, Amsterdam [Kersten81b, Kersten], to store the system structure information and to drive the different software specification and analysis tools.
In this section we illustrate two aspects of the PLAIN programming environment: a module interconnection facility in the development phase and the realization of the programming environment using an SPE machine.

Going from PLAIN programs to the construction of large scale information systems poses three problems not covered by the language semantics: incremental system construction, integration and versioning, and project management. Incremental system construction is a technique to build systems through gradual extension of a family of application programs. Incremental system construction is often based on separate compilation, a technique to compile programs components delaying variable binding to the linkage editing phase or execution time (dynamic linkage). For this purpose the compiler maintains a database of (un)resolved names and the interface definitions of the program constituents. Example high-level programming languages providing separate compilation and tight control over the interfaces are CLU [Liskov81] , Euclid [Lampson77] , and AdaIchbiah79.

The SPE model can function as an architectural framework for the construction of a separate compilation facility for PLAIN, because it describes the infrastructure and the access relation between the system parts. The model does not restrict the semantic issues involved in linkage, such as text/ data/ bss memory allocation scheme. However, the model constrains execution of separately compiled PLAIN code. That is, a PLAIN program can only be executed if the corresponding SPE state is secure, i.e. all access references are resolved.

6.3.1. Module interconnection languages

Incremental system construction, supported by separate compilation, can be seen as a bottom up approach to system development. It does not enforce any decomposition or hierarchical system structure in advance. On the other hand, system structure and access relations may be derived from the design process and enforced by the compilation system. This technique was first proposed by DeRemer and Kron [DeRemer76] They stipulate that the construction of complex systems should primarily constitute programming in the large, knitting (large) pieces together, rather than programming in the small, writing low level code. Their suggestions have triggered research on models and tools for capturing the interconnections between program components (modules) at design stage, which resulted in several Module Interconnection Systems [Cooprider79,

Thomas76].

A module interconnection language (MIL) can be considered as a design language, because it states how modules of a specific system fit together to implement the system's function. This is architectural design information. The basic functions of a MIL are:

a) Description of the system structure
b) Establishing static inter-module connections
c) Provide different kinds of access to resources
d) Manage version control and system families.

MILs are not concerned with what the system does (specification information), what the major parts of the system are and how they are embedded in the organization (analysis information), or how the individual modules implement their function [Pietro-Diaz83]. In particular, a MIL does not address the following compilation properties:

a) Loading
b) Functional specification
c) Type specification
d) Embedded link-edit specification.

The trend in MIL development is to keep the domain of the MILs well defined so that stand-alone MILs can be defined and then integrated as part of a software development environment, such as Gandalf [Habermann79]

The SPE model is a good architectural framework for developing a MIL as well. System structure, import, export, and inheritance properties as used in [Thomas76] can be mapped into structure relation and accessibility directly. Module and (sub)system map to object and region respectively. The version concept can be mapped to user entities in the protection model, which then models the access relationships of versions and the objects composing a system version.

6.3.2. Module interconnection and project management

During the software development phase each program (module) can be (and should be) assigned a person principally responsible for its creation. The extend to which a user can obtain or use other pieces, that is, software owned by other users, depends on the provision of access rights (explicitly) granted. The obvious way this is achieved in existing systems is to maintain one or more library files (maintained by a librarian), to reflect the access rights by a particular group of users. The disadvantage of this method is inflexibility in the dissemination of the components needed by programmers to complete their task. The file system protection deals with files as units and provides limited facilities to distribute access permissions on an file component basis.

Designing a Module Interconnection Language using the SPE protection model as a basis makes it possible to address these project management issues.

In particular, an SPE based MIL can be used to control the communication paths between the programming teams. A sketch of this approach follows.

6.3.3. A PLAIN Programming Environment

The SPE model is used as an infrastructure for a programming environment for PLAIN, hereafter referred to as PLAIN*. In the description of this programming environment we restrict ourselves to the basic relations among programs and processes. We ignore the large collection of semantics specific to editors, library management, etc. The infrastructure of PLAIN* is based on the extension of the scope concept in the language itself, for which we have seen that the SPE model primitives are applicable as well. Figure 6.5 shows some commands available to the user of PLAIN*, partly based on the commands available in the User Software Engineering Control Systems [Rubin84].
Presumably these commands are recognized by a command interpreter and supported by the proper SPE kernel programs. Most commands are directly obtained from the SPE instruction set. The command change(name,region) permits users to switch between different regions they own without the necessity of re-authentication by the system.

```
addowner(region-name, user-name)
addregion(region-name)
addstructure(region-name, environment-region-name)
addimport(object-name, region-name)
addexport(object-name, region-name)

delowner(region-name, user-name)
delregion(region-name)
delstructure(region-name, environment-region-name)
delimport(object-name, region-name)
delexport(object-name, region-name)

runprogram(objectname,region-name)
newsystem(username,region-name)
newversion(username,region-name,version-name)
compile(file-name, region-name)
change(username,region-name)
```

Figure 6.5 PLAIN* commands

PLAIN* deals with relations between users, system versions, and PLAIN lexical objects. This means that PLAIN* users should be recognized, i.e. authenticated, and an alias should be determined corresponding to the SPE user entity. The methods to realize this within the SPE machine have been presented earlier (Section 6.1.7).

Initialization of a new system or project is implemented by the command newsystem, which takes the alias for the system manager, say FooMgr, to create a region and assigns ownership. In addition all PLAIN* commands are assumed to be accessible to the system manager. Subsequently one or more projects are defined by installing region managers and project managers as the sole owner. By setting up a structure relation between the system region and the project regions, using addstructure, PLAIN* commands can be imported by the project managers using addimport. The project managers recursively define sub-projects for their programmers. The resulting infrastructure is shown in Figure 6.6.

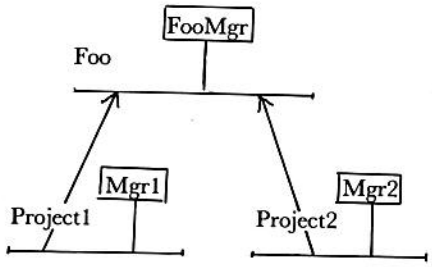


Figure 6.6 Project infra structure.

Construction and installation of software now becomes a two-phase process. First, the programmer prepares a PLAIN object definition, a PLAIN program or part thereof, using one of the editors built (or included) in the region manager. Second, it presents the file to PLAIN* using compile(file-name, region-name). The compile command performs a syntax and semantic check and copies the text file into a region/object structure as illustrated in the previous section. The net result is an extended protection state reflecting the specifications of the program.

Note that the decomposition of a program into smaller components and its representation in region/object/structure relations provides the means to selectively replace or extend the program definition using an enhanced (syntax driven) editor. Moreover, the access sharing commands, addimport and addexport, can by applied directly to the protection state.

6.3.4. Object sharing

Building large systems is teamwork, which implies that a programming environment should provide tools for selective communication as well as 'broadcast' communication among the team workers. Sharing through a common parent region, the use of filters, and the construction of private communication channels are presented to illustrate this.

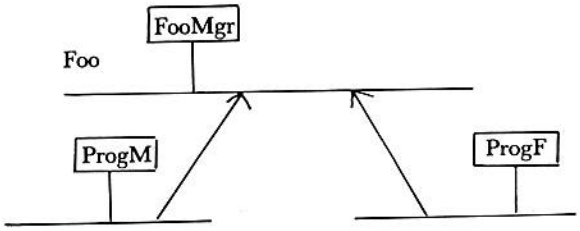


Figure 6.7 Project structure

Consider two programmers, ProgM and ProgF, working on the same project Foo and managed by FooMgr. The region structure of this project is illustrated in Figure 6.7. The authorization policy of SPE and thus for PLAIN* allows both programmers to install software in their regions. Moreover, the protection model supports selective transport of access rights in four different ways:

a) local trust communication
b) mutual trust communication
c) third party communication
d) mutual distrust communication

Local trust communication allows both participants to communicate through a common ancestor region. In our example, both ProgM and ProgF can import all objects installed by FooMgr through the addimport command directly and exchange access rights by exportation of objects. However, each export makes the object accessible to FooMgr and all other programmers represented by subregions of Foo. Note that both programmers exchange objects without active support of FooMgr, yet are restricted in communication with the rest of the 'world'.

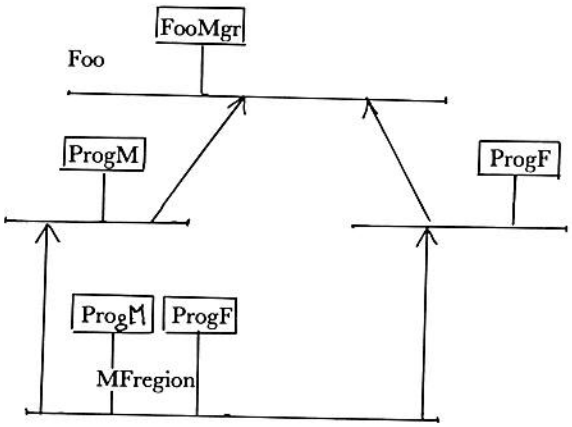


Figure 6.7 Project structure

A more private communication channel is the construction of a mutual trust channel, illustrated in Figure 6.7. Both programmers end up with unlimited access to objects in the neighbors' region. Provided, of course, that the primitive SPE operators are available. The construction of the channel requires both programmers to cooperate, because the SPE authorization requires the invoker to own both regions involved. After this channel has been installed, both ProgM and ProgF can import objects into the common region and export it to the region of the other.

The lack of selectivity in access flow is handled either through the introduction of an independent third party or a filter region. A third party is of use when access flow is known in advance and prescribed by a contract, which is guaranteed by the third party. The introduction of a trusted user between ProgM and ProgF to regulate access is shown in Figure 6.8.

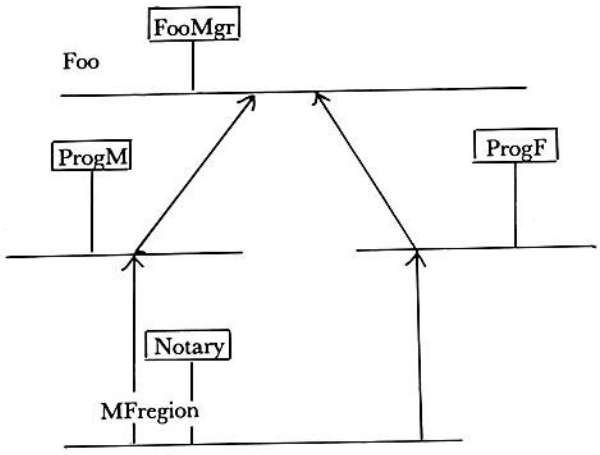


Figure 6.8 Third party communication

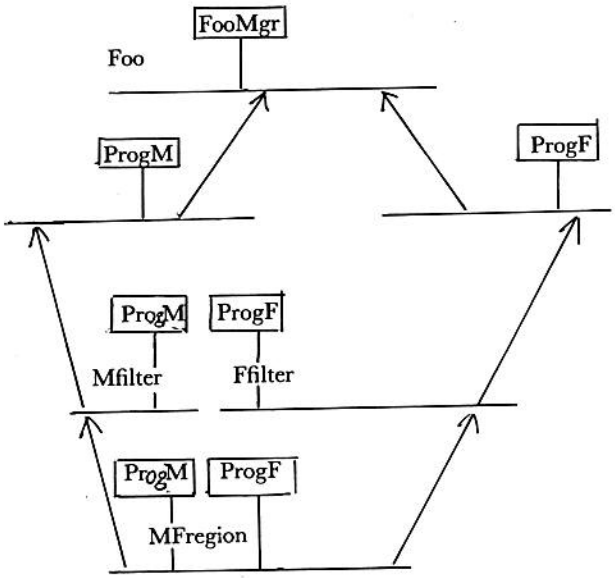


Figure 6.9 Restricted third party communication

To reduce access permissions of the third party a filter region can be used. The final situation is shown in Figure 6.9. The owner of the filter region selectively imports the objects accessible to the other programmer. The result of the actions is that both programmers can share selectively objects. Gaining access to an object requires the active cooperation of the other, to make the

object accessible within the filter region.

The manager can, of course, use the concept of a filter as well. Placing a filter between the Foo region and the programmer's regions disables the use of the system region as a *mailbox* between ProgM and ProgF. Exchange access rights now requires active cooperation of FooMgr. Moreover, FooMgr can selectively provide both programmers access to his objects.

6.3.5. Summary

The global description of PLAIN* shows that indeed the protection model can function as a backbone for structuring the programming environment imposing the protection infrastructure. The realization of PLAIN*, including a database management system, editors, debuggers, graphical design tools, etc., is a major time investment and its description beyond the scope of this thesis.

The SPE model provides a clear picture on the relationship between pieces of the program without going into too much detail. It focuses on the access permissions, that is, can a user use parts of the software in a given context? An example of our approach is published in [Riet83].

The SPE model invariants ensure that the relations between the components are consistent. In particular, it places a constraint on compilable programs. That is, a program can be compiled when its region/object structure represents a secure state only.

The SPE model shows a way to decentralize authorization. Each user owns part of the system and can freely communicate with his "twins" using a common parent region as a mailbox. Moreover, the PLAIN commands can be regarded as objects and thus their invocation becomes subject to SPE authorization. That is, the system manager can withhold SPE primitives or provide parameterized versions. For example, the system manager can, using a filter, withhold project managers to define new users or provide new command to do this in a new way.

Theorems derived for the SPE model indicate the potentially dangerous situations where access rights may leak to non-trusted users. For example, the private channel of Figure 6.11 is only partly secure, because it allows both programmers to import all objects from the other's region. In other words, ProgM can steal objects from ProgF without any conspirator. Using the filter mechanism to restrict the commands available to the FooMgr, project leaders can be forbidden to enter new programmers. This corresponds to the situation that a limited instruction set is available, characteristics of which have been analyzed in section 5.4.

6.4. Dynamic behavior

In the previous sections we addressed the lexical aspects of PLAIN programs both at a microscopic level, in a single program, and the macroscopic level, in the controlled construction of (interactive) systems. It showed that the SPE model can be used as a framework for access control enforcement and system structure in both situations. However, it relies on the proper incorporation of the model in both the compiler and its environment. A machine architecture has been introduced for SPE as well and it is worthwhile to investigate the consequence of this architecture on the compiler construction and the run time environment, such that the symbiosis of compiler and SPE machine guarantees the protection properties of PLAIN. All protection decisions are made by the security kernel and the compiler generates only proper parameterized calls to the security kernel. As a consequence, the compiler can not rely on isolation of the object manipulated in the address space of the process to determine access violations.

First, we show how procedures in PLAIN can be modeled in terms of SPE programs. Second, the mapping of variable declaration in PLAIN to SPE machine objects is discussed using the declaration and subsequent use of abstract data types. Constants, files, and patterns can be considered variations on this theme. Third, we show how parameters are treated. Finally, extended access control and database representations are discussed. Studying the protection issues in this context raises problems due to concurrency and multi-processing, which are ignored in the compiler based approach discussed here.

6.4.1. Procedure invocation

In the introduction of the SPE machine we proposed a scheme for operator invocation based on message passing. Each message is scrutinized by the security kernel and sent to the region manager where the procedure being called is defined. In the presentation of the protocol we ignored the choices underlying the protection policies, which can be classified as follows:

a) who may invoke a procedure,
b) where is the procedure being executed,
c) what are the access rights provided to the process*.

The algorithmic properties defined by the procedure body are ignored as far as they do not involve any transfer of access rights or (in)directly change the protection state. Thus, procedure calls, parameters, and object declarations are retained for analysis only.

Issue a). There are limited options, because SPE requires that at least one user

* A process is the execution of a procedure.

should be associated with each process. Moreover, the authorization policy requires that this user is an owner of the region manager in which the procedure is called or executed. For this purpose, the user is represented within the process by an alias, sent to the security kernel for authorization purposes. Moreover, the protection model implies that a procedure can only be invoked when the corresponding SPE object is accessible within the region manager of the call, because that is the only way to name it. it can not be named.

Issue b). Procedure execution requires a new name space to identify the local variables uniquely. In terms of SPE, this means that execution is coupled with a new region and that we should use a relative naming scheme. Isolation is needed both to avoid name clashes and to avoid interference of concurrently running processes in the same context.

Issue c). Access rights needed by a process are both determined statically, in the procedure definition, and dynamically, through the use of parameters. This implies that, during procedure initialization, access rights should be transferred between region managers using existing structure relations or by building a private communication channel between caller and callee. The former requires the compiler to trust users to cooperate along the structure paths. This cannot be enforced in all situations. Therefore, access transfer should take place using a private channel, i.e. a structure relation between the defining region and the region from which it is being called.

In conclusion, a procedure call results in the creation of a region with structure relations to both the caller region manager and the region manager in which the definition resides. The former path is used to obtain access rights for actual parameters, the latter to obtain access rights based on the lexical structure of the PLAIN program.

The region manager in which execution takes place requires an owner, the choice of which greatly impacts security, because it determines the access rights available during the call. Three different authorization policies can be distinguished:

a) Caller based authorization

The execution region is owned by the caller, i.e. the access rights of the invoked procedure are determined by the user responsible for its invocation.

b) Callee based authorization

The execution region is owned by the callee, i.e. the access rights of the invoked procedure are determined by the user responsible for the procedure definition.

c) Third-party authorization

Under a Third-party based policy both the caller and callee provide access rights, but have no control over its use during procedure execution. The activator of the state transformation is a (unique) fictive user or notary.

Each policy is illustrated in detail, showing the structure of the SPE programs derived by the PLAIN compiler. We use an example type T with two operators crt and proc, the type T operators are used in the region R. The static structure as it results from a PLAIN program definition is shown in Figure 6.10. The operators defined for the type T are represented by an SPE object within the region associated with the static definition of T. Note that both operators are made accessible within the region R through appropriate SPE primitives and that we have introduced a fictive owner of the type T, called Tu.

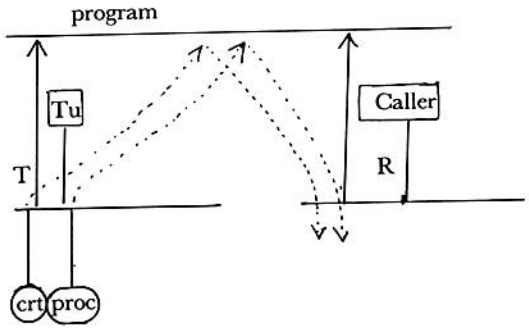


Figure 6.10

6.4.2. Caller/callee based access control

Under a caller based access control policy the SPE process execution is bound by the PLAIN compiler to one user; the user responsible for the procedure activation. The code generated by the compiler includes references to designate this user. Variable user names are replaced upon procedure activation by the actual user name, similar to the association of UIDs in operating systems.

The process prologue and epilogue for this policy is represented in Figure 6.11. The command condition contains two access control constraints; the procedure is called by a user having access to the procedure definition and the user is owner of the execution region. The result of calling exec1(Caller,R,T/proc,Newcontext) on the SPE state is shown in Figure 6.12.

```
command execl(caller, context, proc, newcontext)
if Visible(caller, proc) and Owner(caller, context) and newcontext ∉R
begin
        add_region(caller, newcontext);
        add_struct(caller, newcontext, context);
        add_owner(caller, newcontext, reg_own(reg_obj(proc)));
        add_struct(reg_own(reg_obj(proc)), newcontext, reg_obj(proc));
        del_owner(caller, newcontext, reg_own(reg_obj(proc)) );
        {import global objects}
        {execution of body }
        {revoke import global objects}
        revoke( add_owner(caller, newcontext, reg_own(reg_obj(proc)));
        revoke( add_struct(caller, newcontext, context));
        revoke( add_region(caller, newcontext))
end
```

Figure 6.11 Caller based policy

The construct reg_obj(proc) evaluates to the name of the defining region and is a protection state inquire operator. Similarly, reg_own(reg_obj(proc)) returns the name of an owner of the procedure. Both have no effect on the protection state and could have been represented in the parameter list as well. In this case the caller can be determined from the context of the call also.
The epilogue of the procedure invocation removes all objects necessary for its working. Note that this involves a revocation process, instead of calling the corresponding decremental operators. This way imports and exports of access rights which result from the execution of the procedure body are undone as well.

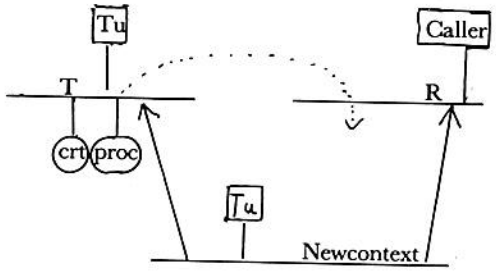


Figure 6.12 The SPE state after the invocation of T/proc.

The shortcoming of this approach is its lack of selectivity, because access rights granted to the Caller are increased during procedure execution. More precisely, the SPE model shows that access rights private to the type manager are subject to stealing by the Caller. Therefore, the command does not provide real information hiding. During the interpretation of T/proc the caller is owner of

the region newcontext and thus objects created/manipulated are subject to stealing too.

A secure implementation of the procedure call mechanism requires that at least two parties should be considered: the Caller and the Callee. Both are autonomous users in the SPE machine, which means that one user can not force the other to initiate actions. However, the cooperation required between the users can be partly prescribed by the compiler and modeled in SPE. To achieve a secure implementation the compiler should take into account the different protection requirements:

a) Caller provides Callee with a set of access rights and expects Callee to be unable to acquire more,

b) Callee may need private access rights for its task and should ensure that the Caller does not acquire any of these.

The latter is referred to in the literature as the problem of "rights amplification," the Callee needs more rights then the Caller can supply. In that case of abstract data types, the Callee needs access to the internal representation of the abstract object and doesn't want the caller ever to be able to gain access to the representation. Amplification of rights occurs as a side-effect of name visibility in scoped languages. This poses no protection problems in their implementation, because access control is then fully enforced by the compiler. However, a compiler based on separate compilation should take special precautions. Especially when object binding is achieved at run time.

These requirements are easily fulfilled using the modeling capabilities of SPE. First, to avoid the Callee to gain access to the objects in the region Newcontext a filter region is constructed to shield Newcontext. The second problem is solved by removing the Caller from Newcontext before the actual interpretation of T/proc starts. To safely transport access rights, a structure relation is constructed between Newcontext and T. The resulting SPE command is shown in Figure 6.13 and the SPE state at the moment of body execution is shown in 6.14.

```
command exec2(caller, context, proc, newcontext)
if Visible(caller, proc) and Owner(caller, context) and newcontext ∉ R
begin
        add_region(caller, filter);
        add_struct(caller, filter, context);
        {import objects needed into filter region}
        add_region(caller, newcontext);
        add_struct(caller, newcontext, filter);
        {add access rights to newcontext}
        add_owner(caller, newcontext, reg_own(reg_obj(proc)));
        del_ownerer(caller, newcontext, caller);
        {link new context with reg_obj(proc) region}
        add_struct(reg_own(reg_obj(proc)), newcontext, reg_obj(proc));
        {ready to interpret the procedure}
        {execution of body}
        {revoke all protection statements in this command }
        {not spelled out in detail}
end
```

Figure 6.13 Restricted caller based procedure invocation

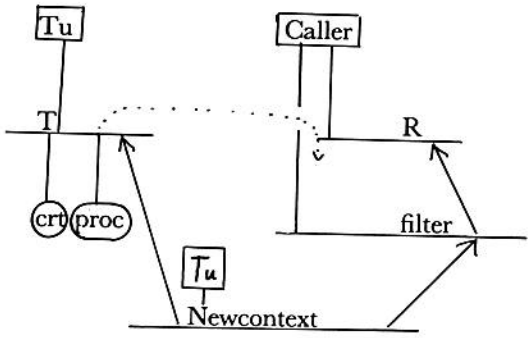


Figure 6.14 SPE state after restricted call

In this procedure we assume that the compiler is able to generate a locally unique name for the filter region. Of course, this name can be replaced by an actual parameter or an SPE primitive operation. Moreover, the construction of the SPE intermediate state is assumed to be revocable, which has been previously addressed.

Whenever the compiler uses this command as a basis for generating code for the PLAIN procedure call, the two protection questions are guaranteed to hold. In fact, the type manager and routine can be compiled with binding delayed to run time. Moreover, the type manager can make additional dynamic protection

decisions and does not need to rely on the proper working of the compiler in the enforcement of such constraints. For example, read/write access to objects may be managed by the type manager.

6.4.3. Third-party procedure invocation

The distinction between Caller and Callee processing provides a natural solution to the confinement of abstract data type object representations and rights amplification. However, in the previous scheme we implicitly assumed that Tu, the owner of the type T, could be trusted. The owner of T/proc has full access to all internal representations and is able to disclose information to different Callers either through import/export operations within the context T or by using shared representation objects. Moreover, information embodied by the actual parameters can be retained by the type manager and passed on. Although these problems fall under the category of confinement, the SPE model provides the hooks to partially overcome them.

A method of procedure invocation where Caller and Callee are mutually suspicious is solved by a mutually trusted third party. An SPE user is introduced to mediate between Caller and Callee for each procedure invocation and becomes the activator of the SPE primitives specified in the execution region. This method is similar to the concept of a notary office in real world. The corresponding SPE execution framework is shown in Figure 6.15 and the result of the invocation of T/proc is graphically displayed in Figure 6.16. The process runs as follows. First the caller creates a filter region and imports the objects needed for processing. Similarly, the type owner makes a filter region with the necessary access rights. Finally, the notary creates the isolated environment for procedure execution. Note that we can't make the notary owner of the context and type region, because this implies that he gains access to the environment of both as well. In the command definition we assume that filter and newcontext names are generated dynamically.

```
command exec3(caller, context, proc)
if Visible(caller, proc) and Owner(caller, context)
begin
        add_region(caller, filter1);
        add_struct(caller, filter1, context);
        add_owner(caller, filter1, Notary);
        {import objects from caller context into filter region}

        add_region(reg_own(reg_obj(proc)), filter2);
        add_struct(reg_own(reg_obj(proc)), filter2, reg_obj(proc));
        add_owner(reg_own(reg_obj(proc)), filter2, Notary);
        {import objects from type context into filter region}

        add_region(Notary, newcontext);
        add_struct(Notary, newcontext, filter1);
        add_struct(Notary, newcontext, filter2);
        {add access rights to newcontext}

        {ready to interpret the proc}
        {execution of body}
        {revoke import/exports from newcontext}
        {not described in detail}
end
```

Figure 15 Third-party command

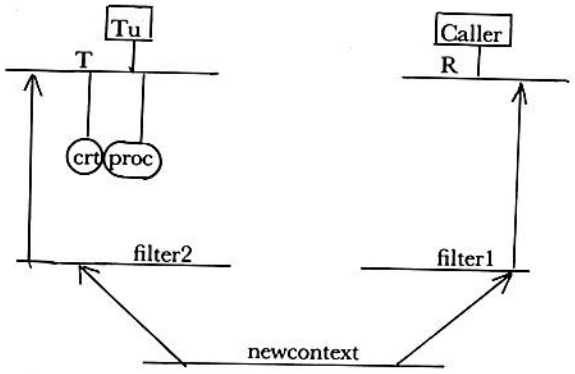


Figure 6.16 Calling T/proc with third-party policy

6.4.4. Variable declarations

PLAIN variable declarations differ from procedure calls by changing the protection state 'permanently'. This is reflected in the mapping of PLAIN to the SPE machine architecture in two problem areas: handling variable declarations and variable usage. We discuss variable declarations here and take up variable usage later, in the context of parameter usage.

The two prime aspects of variable declaration are its type and representation mechanism. The type of a variable determines the applicable operations and reserves and manages space for its representation. In particular, it hides the internal representation from usage outside control of the type procedures. Both typing and information hiding can be considered as access control measures normally encoded in and enforced by the compiler.

As with to the analysis in the previous section, we should consider the protection issues raised and analyze the means to provide this protection within the context of an SPE machine. That is, what are the requirements on and possibilities for the PLAIN compiler given an SPE machine to enforce type checking and information hiding.

The declaration of variables, i.e. abstract data type objects, raises four protection issues:

a) the Caller should be unable to manipulate the variable representation,

b) the Caller should be able to share access to the variable,

c) the type operators have access to the internal representation,

d) no other operator can gain access to the internal representation without consent of the type owner,

e) no other user can use the variable without consent of the owner.

Rules a), b), and c) prescribe the ability to use the objects, while d) and e) limit their use.

In mapping PLAIN to the SPE machine concepts, the declaration of variables can be considered a procedure call with a modified epilogue phase. For example, the declaration of an object o of type T can be thought of as handled by an operation T/crt(o) using one of the invocations schemes but with omission of the revocation phase. That is, after the procedure finishes, one or more SPE objects, regions, and structure relations remain in existence to represent the variable and its contents. The result, using the third-party scheme, is shown in Figure 6.17. In the region newcontext the SPE object o represents the variable and access to this object is exported to the declarer and type manager via the filters.

Interpretation of a variable declaration as a suspended procedure call implies that the SPE objects used for the variable representation are not created in the context of the call, but in the execution region.

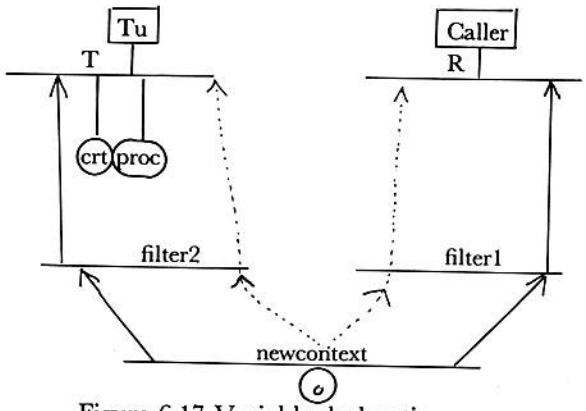


Figure 6.17 Variable declaration

In most cases the representation of a type T variable is one or more variables of type T', the representation of which should be accessible within the context of T only. In general, the SPE objects representing the T objects can not be given to the declarer of the variable, for it would invalidate information hidding. Then the T objects could be manipulated directly by the T' operators.

One approach to overcome this access problem is to use a single SPE object to represent the variables of type T and to use this object for no other purpose then as a variable alias. One may think of it as a capability or ticket (It does not represent specific access rights). The mapping of a variable-alias to the internal representation objects should be coded into the type operators, which are permitted access to the internal representation anyway. Yet this precludes a user to obtain access to the T' objects, unless explicitly granted by a T operator.

For example, when an variable o of type T is declared, which uses a type T' for representation, the SPE structure of Figure 6.18 is generated. This SPE structure provides R access to o within Newcontext, but does not provide direct access to its internal representation o'. Moreover, owners of R can share access to the abstract object by sharing access to the variable-alias using the import/export mechanism.

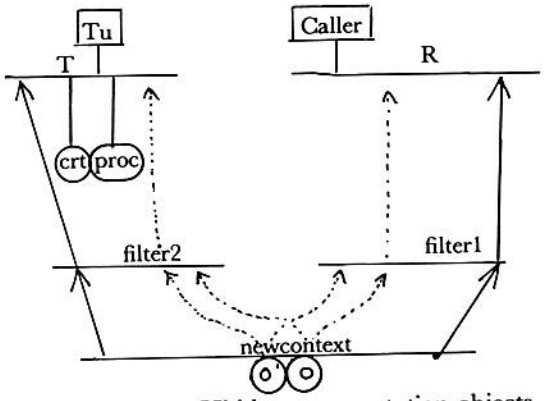


Figure 6.18 Hidden representation objects

The operators defined on T use the object o' instead. Thus, the SPE command to create an instance of an abstract PLAIN variable becomes:

```
command declare(caller, context, variable, type)
if Owner(caller, context) and Visible(caller, type)
begin
        { set up region structure using Notary }
        add_region(caller, filter1);
        add_struct(caller, filter1, context);
        add_owner(caller, filter1, Notary);
        {import objects from caller context into filter region}

        add_region(reg_own(reg_obj(proc)), filter2);
        add_struct(reg_own(reg_obj(proc)), filter2, reg_obj(proc));
        add_owner(reg_own(reg_obj(proc)), filter2, Notary);
        {import objects from type context into filter region}

        add_region(notary, newcontext);
        add_struct(notary, newcontext, filter1);
        add_struct(notary, newcontext, filter2);

        { create the variable-alias and give it to caller}
        add_obj(notary, variable, newcontext);
        add_export(notary, newcontext, variable);
        add_export(notary, filter1, variable);

        { suspend procedure until end of variable scope}
        { revoke all actions}
end
```

Figure 6.19

Note that we assume that the variable name is unknown in the state before the execution of this command. In the protection state derived from this command, the caller has no access to the variable representation and thus access rights on it cannot be stolen.

6.4.5. Variable usage

Variable semantics in PLAIN require the compiler and SPE machine to guarantee both type correctness and information hiding. In particular, we must define what access control enforcement should take place upon invocation of T.proc(o) and how the parameter o is interpreted.

For the caller of the procedure, two access issues are implied by name visibility:

- the caller should have access rights to the procedure T/proc, and

- the variable alias is accessible to the caller.

From the standpoint of the callee, two access constraints are in effect as well:

- the object o should be of type T and
- all references to o within the procedure T/proc should be mapped to o', its representation.

Access control for the caller is handled similarly to the procedure mechanism described earlier. An extension can be included to import the variable-alias into the execution region of the procedure call. Under callee based procedure processing, type compatibility enforcement is realized using a combination of SPE features. Type correctness means that the parameter o is a variable alias owned by the type T. This constraint can be modeled in the pre-condition of T/proc. Thus, the pre-condition for each operator defined for T should include:

Access(caller, o) and
Visible(caller, T/proc) and
Owner(o) = callee

If the pre-condition succeeds, the internal representation of o, i.e. o', can be made accessible in the execution region through the construction of a safe communication path. All references to the internal representation of o in the definition of T/proc can safely be replaced by references to o' and each call to an operation belonging to the class defined by T' is considered activated by T. Thus, to provide type enforcement and information hiding we can extend the procedure invocation prologue with a simple test.

Parameter passing

Given the mapping for operations on variables using a single parameter to designate the object, we can simply extend this scheme to handle the more general procedure parameter mechanism. Note that each routine in PLAIN is defined by the following structure:

routine(O1:T1;...;On:Tn)

This heading prescribes the order, type, and formal name of the actual parameter. Protection issues involved in its use are:

a) does the caller have permission to use the parameters
b) does the callee have permission to use the parameters
c) is the parameter of the correct type

Unlike variable declarations, procedure definitions in PLAIN differ in number of parameters and parameter types. In the mapping to an SPE machine we use a single command to describe the actions for passing a single parameter. Multiple parameters can be handled by repetitive application of this command.

Figure 6.20 describes the SPE command to enforce the three access constraints on a single parameter. Note, that we used variable aliases and type owners to

guarantee type correctness.

```
command parameter(caller, callee, newcontext, variablealias, typeregion)
if Visible(caller, variablealias) and Visible(callee, newcontext)
   Owner(variablealias)= Owner(typeregion)
begin
      add_import(callee, newcontext, variablealias)
end
```

Figure 6.20 Parameter handling

6.4.6. Extended access control

The parameter mechanism in PLAIN is more involved then illustrated above. In particular, semantic access rights are associated with the parameters, i.e. readonly or modify rights. Clearly, these notions are not provided by the SPE model directly, which concentrates on flow non-interpreted access rights. To enforce these access rights within a dynamically interpreted SPE environment we should specify for each parameter what operators are available. This in turn, requires the class of operators defined for a type to be separable, or the ability to define multiple views on the same variable, differing in the class of applicable operations.

We illustrate the extended access control using multiple views. Assume that we have to distinguish two different access categories, as in PLAIN. Then, instead of using one variable alias, we use two variable aliases to represent access rights to the object and associate with each alias access rights to an operator class. The association between operator and alias is realized using filter regions for each access rights. Next, extended access control can be enforced by requiring that each operator usage is represented by access rights in the region where the corresponding alias is defined. The internal representation of the object is hidden in a separate region.

For example, consider a type T with operators readobj and writeobj, and assume that we wish to restrict the use of the operators for the object o to either read or write. Then for the declaration of o we construct an SPE state as shown in Figure 6.21. The region r' and r" are given access rights to the read and write operator only, while both have access to the internal representation of the object. To guarantee proper access we extend the access control statements above to include the condition that in order to use an operator the alias presented should be defined in the region where the operator is accessible. This way, the read operator can only be used in the read-class context. Notice that we used filter regions to separate access properties.

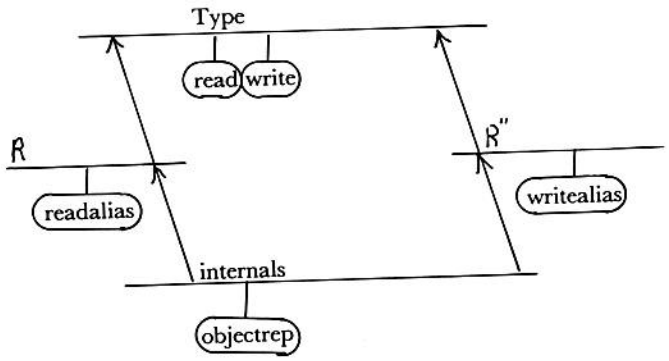


Figure 6.21 Readonly and Modify rights on the same object

A zone representation

An alternative scheme for variables of a certain type is to use a single region for their representation, rather than dispersed in the protection graph. The prime benefit of this approach in an implementation is the ease of removing all objects whenever the type becomes obsolete and the provision of operators on all the existing objects, like garbage collection, database facilities, and statistical purposes. The programming language EUCLID uses this scheme in the notion of a zone.

This scheme can be described in the SPE terminology as well. For, consider subordinate to each type definition, a region called Zone, were all the representations are stored. Then the remaining protection issue is whether privacy of the object representation can be guaranteed to its owner (=creator). Stated alternatively, if the operator T/proc can be applied by the owner of the object only (or those he has granted access), how should the SPE graph be constructed and what authorization should be enforced.

To avoid naming conflicts we should store the object aliases and their representations not within the Zone region directly, instead we represent each object by a region subordinate to Zone. Thus, the execution region structure used previously is moved to the Zone region. Unlike the previous scheme we can not build a structure relation between the variable representation and the context of the caller directly. For, an export of the variable alias would make it accessible to other callers having defined a T-object as well.

The solution is to construct an agent between the hidden region and the caller context, owned by the type. This scheme is similar to the secure communication path defined in section 4, i.e. the caller is given exclusive access to the variable-alias. Part of the SPE structure for the type T and two instances is shown in Figure 6.22.

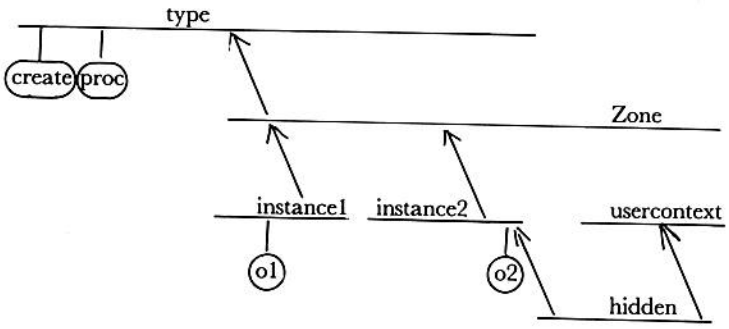


Figure 6.22 Database oriented ADT implementation

6.4.7. Summary

The scheme presented to combine PLAIN with an SPE machine architecture may seem artificial at first. We claim it is not, in fact it models what happens in many so-called capability oriented systems using type-managers. The representation of the abstract object by a primitive SPE object is identical to the representation of the object by a capability object. The capability object is given to the declarer, which can share it with others. However, our scheme goes a step further. It can model situations where access to objects is shared with others, which do not have access rights to the operations defined upon the type, because the applicable operators are not present in the capability.

For example, in a capability oriented system like Hydra [Wulf74] or AMOEBA [Tanenbaum81], a line-printer spooler inherently obtains read access permission to the files to be printed, because the file manager does not distinguishes the rights associated with the objects from the rights to use certain of its operations. It examines the capability objects and executes the task. The example use of SPE in restricting access to variables and its operations shows that to ensure that the spooler does not read any file. In fact, each action on a file requires two capability objects, one to indicate the access to a particular file and one capability to indicate the right to perform the required file operation. This way denying the spooler read access to the files being spooled leads to a more secure system. The authorization mechanism for such a two-edged approach becomes

Visible(caller, operator) and Visible(caller,object)

That is, the operator should be accessible to the user as well as the object. Where ObjRegion(operator) stands for Newcontext in our example.

The extended access control scheme discussed for the PLAIN parameter mechanism indicates a solution to a major protection problem in the databases as well. Whenever a database is defined by an abstract data type, as proposed in [Wasserman79b, Riet83] using conventional programming semantics, its internal representation becomes inaccessible for separately compiled programs or results in an insecure system. Within a database environment where database objects are persistent and shared by many application program under various views, this constraint is too severe, rather opaqueness of the representation is required. Our scheme illustrates how multiple views can be defined on a single representation, using a classification of the applicable operators.

On the other hand, the schemes discussed do not directly model the situation where the declarer binds an abstract object with a subset of operations, as in a capability system. Two alternative solutions exists, either a new abstract data type is defined to represent this object or for each operator an alias object is stored in the Newcontext region and made accessible to the owner of the object, which in turn can selectively grant rights. We favor the first, because one can consider the reduction of the operator class on an object as a change of its semantics, which naturally leads to the definition of a new abstract data type (or package, because the representation may be shared).

7

SUMMARY AND FUTURE RESEARCH

7.1. Summary

In this thesis we have developed an access control model for a secure programming environment. Although access control addresses only a small part of security in computer systems, it is an important aspect of data security, where emphasize is placed on the data stored and manipulated in a computer. Evenly important areas of computer security not addressed in this thesis are physical, operational and organizational security.

In Chapter 2 access control protection is placed in a broader perspective by surveying the work done in this area. This survey shows that much of the work is pragmatic; many techniques have been introduced without a proper formal model and security analysis or without a proper separation of the protection concepts, such as authentication, authorization and the semantics of the objects being protected. Moreover, the lack of a formal basis makes integration of access control in operating systems, programming environments, and programming languages difficult to achieve.

Chapter 3 informally introduces the SPE model as a unifying model to describe and analyze access control with emphasize on programming environments for the construction of interactive systems. The security axioms and security properties underlying the SPE model are explained and a rationale is given. Moreover, a few examples illustrate the applicability of the model in real-world situations. Furthermore, some variations of the model are given to illustrate the effect on expressiveness and ability to analyze security issues.

In Chapter 4 we have introduced the SPE model by formally specifying the static and dynamic properties of SPE protection states. An SPE protection state is considered secure when it satisfies the independent properties consistency, acceptability, and validity, which model the security properties introduced in Section 3.3. Following, the security properties of SPE states have been related with directed graphs, which gave a handle on their algorithmic complexity.
The dynamic properties of the SPE model are cast into state transformation invariants, which thereby describe the secure behavior of potential SPE instruction sets. In Section 4.4 we have indicated how authorization policies enter the scene as a separate dimension on access control. A particular authorization policy for the SPE model has been introduced, which models the rule that each user changing the protection state should own (or be responsible for) the protection domains where the changes take place. Finally, desirable properties of SPE instruction sets, i.e. minimality criterion, compensation criterion and completeness criterion, have been pointed out as forming the basis for *well-defined* instruction sets. Although no algorithm exists that can check the well-definedness property handed an arbitrary SPE instruction set, a given set may well be proved to satisfy this criterion.

In Chapter 5 we have introduced a sample instruction set for the SPE model and showed that it is *well-defined.* In Section 5.4 the compensation property is extended to cover the notion of revocation. Revocation prescribes a policy to undo actions in general. The *chronological* algorithm and the *goal-seeking* algorithm were presented to realize a revocation sequence. They differ in objectives and implementation costs. Subsequently, instruction set partitions were studied to limit derivable protection states. This way alternative security policies can be implemented. In that context the notion of stealing access rights has been given a formal definition and an algorithm to determine potential theft and the conspirators has been indicated.
In Section 5.6 we have indicated the potentials of a multi-level security system based on the SPE model and used this approach to simulate two well-known theoretical protection models. The simulation of the Harrison-Ruzzo-Ullman model emphasized the approach taken by the SPE model in viewing access control protection as a state transformation invariance property rather than a state property or a state reachability property. Moreover, it has been shown that both protection systems are equally powerful by simulating the Turing machine. The simulation of the Take-Grant model showed how the SPE model can be extended to accommodate semantic rights.

Chapter 6 discusses the practical implications of the SPE model. First, a machine organization centered around loosely coupled processors is presented. It shows that a security kernel approach with SPE as the underlying model for this kernel is feasible. Next the SPE model is used to analyze visibility rules in the high-level programming language PLAIN, which shows some omissions and inconsistencies in its definition. Extending the visibility rules to include the

environment of PLAIN programs using the SPE model provides a basis for a secure programming environment. Finally, the PLAIN programs and the machine organization are combined to study the dynamic behavior of program execution.

7.2. Future research

7.2.1. Theoretical issues

The approach taken in this thesis to formalize the access control problem using set and graph theory and using *auth-*, *pre-* and *post-* conditions in the description of state transformations is just one approach among many formalisms. An interesting theoretical question is whether other such formalisms are more effective then ours in the description and analysis of the security problems. Some formalisms to consider are shortly indicated; an introductory work for language oriented formalisms is [Pagan81].

In section 2.4.3 we introduced the grammatical approaches to access control. They are useful to define the protection state structure and simple state rewriting rules. The major drawback of grammars is the limitation in modeling complex authorization constraints. Simple authorization schemess as presented in Chapter 4 result in a large set of context-sensitive grammatical rules.

An alternative formalism is the Vienna Definition Language (VDL) [Wegner72], which, similar to our approach, results in an operational specification of the protection system. Unlike the grammatical approaches, VDL makes it possible to formalize both the structure (syntax) and the semantics of pogramming languages. Therefore, VDL potentially has the capability to formalize access control issues. It is unknown whether such an approach is more effective in the analysis of the SPE protection issues than our approach or whether VDL should be considered as a convenient vehicle for implementation specification.

Another formalism advocated by programming language designers is the *denotational* approach [Stoy77], where the meaning of a construct is taken to be some abstract mathematical function. The prime use of this formalism is the rigorous analysis of program computation sequences, e.g. conditional SPE programs. There is no explicit concept of a machine or computation sequence, but the protection state is still present in the abstract form.

The aforementioned formalisms, including our approach, are limited by the fact that they assume nonconcurrent execution of programs. Unfortunately, this assumption need not hold in reality. Potentially interesting formalisms to tackle security problems inherent in concurrent execution of SPE programs and ensuring the protection policies prescribed are modal and temporal logics [Manna81]. Both logics enable one to describe the detailed execution of a

(concurrent) program(s) and not just the function it (they) computes. Relevant concepts in this context to consider are:

- *Invariance properties*, stating that some condition holds continuously throughout the computation, namely the security invariants.
- *Eventuality properties*, stating that under some initial conditions, a certain event (e.q. stealing a right) is eventually realized.
- *Precedence properties*, stating that a certain event must precede some other (e.g. in revocation sequences).

7.2.2. Machine architecture issues

Most systems designed for access control protection are based on capability protection. A detailed study of how the SPE model can be matched to capability-based machine architectures therefore seems in place. Although the protection aims of both approaches have a lot in common, they differ on essential points:

- SPE does not enforce tight coupling between access to objects and their semantics.
- SPE restricts flow of access rights.
- SPE considers access to the object orthogonal to the ability to apply an operator to that object.
- SPE addresses protection issues found in both programming languages and filing systems in a uniform manner.
- The SPE model associates protection with the explicit maintenance of invariants in (concurrently) running programs.

The SPE model and capability-based addressing bear the following similarities:

- The SPE model and capability-based protection rely on typing as a means for data-dependent protection.

Capabilities are nothing but context-independent protected addresses, while in SPE context and flow restrictions form the basis of access control.

The SPE model can benefit from a hardware-supported capability scheme in its implementation of the security kernel. For example, the Intel 432 [Pollack82, Pollack81] microprocessor and operating system is one of the most sophisticated architectures in existence for capability-based protection and provides many high-level features to realize an SPE kernel. In particular, the interpretation of procedures as objects with access rights checked both statically and dynamically comes close to the proposal sketched in Chapter 6. However, the (static) structure relations among environments are not modeled within their system, which makes access flow constraints difficult to enforce. One way to extend or modify the Intel 432 architecture is to change the semantics for object tables and

object table directory as follows. Conceivably, each SPE region structure can be simulated with a single object table containing access descriptors both to objects and to other object tables. Alternatively, a region manager object type can be defined which uses a single segment to administer the objects, users and relationships. In any case, to arrive at a secure implementation it should be possible to limit the use of access descriptors by procedures. This step may require splitting the access descriptor index for segments into two sections, one to administer the locally stored objects and one to administer remote access rights. In conclusion, the Intel 432 is a good basis for a building an SPE machine.

An method adhered to by many software developers is modularization and classification of objects by abstract data types. A similar attitude has been used in the design of the SPE machine and is the use of PLAIN in the construction of interactive systems. As mentioned before, many implementation details must be addressed before an SPE-based system is obtained. At the Vrije Universiteit, Amsterdam, a prototype implementation of the class concept in the C-language using separate processes, has been established [Klauw84]. This prototype indicates that an ADT-like approach in combination with the region manager concept is feasible and leads to a nearly provably secure programming environment.

7.2.3. Software issues

In Chapter 6 we have illustrated the use of the protection model to highlight and comment on language design choices in PLAIN. This language is a traditional high-level programming language. The open question is whether the SPE model is useful in the context of very high-level languages such as PROLOG [Clocksin81] and SETL [Dewar79, Dewar82]. Protection issues have not been addressed for the latter languages in great depth, which should be ascribed to the limited use of both languages for the development of information systems.

A central theme in software development research is the architecture of integrated programming environments. Most researchers in this area address technical problems of human interfacing, ignoring to a large extent the protection issues of a multi-user software development team. For example, the Ada programming environment requirements [Buxton80] leave decisions about the protection to the designer of the tools, but assume that they fits the Ada language framework. A negative aspect of this requirement is that the visibility model within Ada -everything is visible within a program or module unless explicitly forbidden- is less constraining than in PLAIN. In fact, it is not based on the "need to know" principle used in most protection systems.

7.2.4. The future of access control

The SPE model introduced in this thesis provides a framework to compare and analyze access control features of new hardware and software systems. In this respect it covers only part of the security problems. Equally important areas are the authorization policy (Section 4.4), information flow, and the ability to infer confidential information. However, these aspects rely on the facilities provided by an access control mechanism. Formalization of such problems within the context of a specific access control policy has been given limited attention in the literature. Notable exceptions in these areas are the multi-level security model [Bell74], its extensions by [Feiertag79], and the extension of the Take-Grant model with *de-facto* and *de-jure* rules by [Bishop79]. Further exploration of the formal aspects of these areas based on the SPE model may lead to a better understanding of the applicability and effectiveness of either one.

An open area for access control research is its form and function in expert systems and knowledge base management systems. In such systems blocking access to pieces of knowledge, facts and rules, may drastically affect the decision making process or compromise (governmental) policy decisions. For example, consider a social welfare organization as complex as the Dutch system. Then it is clear that an expert system is needed to allow individual social workers to help their clients. How access control policies and techniques can aid in the regulation of the knowledge base usage is unknown and therefore privacy of individuals remains threatened.

References

[Anderson79]
Anderson, R.B., *Proving Programs Correct.* John Wiley & Sons, 1979.

[Andrews80]
Andrews, G.R. and Reitman, R., " "An axiomatic approach to information flow in programs"," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp.56-76, Jan 1980.

[Ardity78]
Ardity, Joel and Zukovsky, Eli, "An Authorization Mechanism for a Data-Base," pp. 193-214 in Database: Improving Performance and Responsiveness, ed. Ben Shneiderman (1978).

[Astrahan76]
Astrahan, M.M. et al., "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems*, vol. 1, no. 2, pp.97-137, June 1976.

[Bell74]
Bell, D.E. and LaPadula, L.J., *Secure computer systems: Mathematical foundations 1,2,3.* MITRE Corp, Nov 1973 / Apr 1974.

[Berson79]
Berson, T.A. and Barksdale, G.L., "KSOS- The Design of a Secure Operating System," *Proceedings NCC*, vol. 48, pp.365-371, 1979, AFIPS Press.

[Bishop79]
Bishop, M. and Snyder, L., "The Transfer of Information and Authority in a Protection System," *Proc. 7-th Symposium on Operating Systems Principles, ACM Operating Systems Review*, pp.45-54, December 1979.

[Bishop81]
Bishop, M., "Hierarchical Take-Grant Protection Systems," *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5, pp.109-122, 1981.

[Borr81]
Borr, A.J., "Transaction monitoring in ENCOMPASS: reliable distributed transaction processing," *Proc. 7-th Int. Conference on Very Large Data Bases*, pp.155-165, September 1981.

[Bratbergsengen79]
Bratbergsengen, K., Risnes, O., and Amble, Tore, "ASTRAL- A Structured Relational Applications Language," *Technical Report 6/79*, June 1979, Division of Computing Sciences, University of Trondheim.

[Budd77]
Budd, T.A. and Lipton, R.J., "Inert Rights and Conspirators in the Take-Grant System," *Research Report #126*, 1977, Dep. of Computer Science, Yale University.

[Budd80]
Budd, T.A., "Safety in Grammatical Protection Systems", TR80-22, Dep. of Computer Science, Univ. of Arizona, August 1980.

[Bussolati80]
Bussolati, U. and Martella, G., "On designing a security management system for distributed databases," *Proc. IEEE Fourth Int. COMPSAC*, pp.288-294, October 1980.

[Bussolati81a]
Bussolati, U. and Martella, G., "Managing Data Privacy in Database Management Systems," *Convention Informatique Latine*, pp.216-230, 1981.

[Bussolati81b]
Bussolati, U. and Martella, G., "A Database Approach to Modelling and Managing Security Information," *Proc. 7-th Int. Conference on Very Large Data Bases*, pp.532-542, September 1981.

[Buxton80]
Buxton, J.N., *Requirements for Ada programming support environments.* Department of Defense, Feb 1980.

[Chambers78]
Chambers, A.D., "Computer Fraud and Abuse," *The Computer Journal*, vol. 21, no. 3, pp.194-198, 1978, British Computer Society.

[Cheheyl81]
Cheheyl, M.H., Gasser, M., Huff, G.A., and Millen, J.K., "Verifying Security," *ACM Computing Surveys*, vol. 13, no. 3, pp.279-340, Sep 1981.

[Clocksin81]
Clocksin, W.F. and Mellish, C.S., *Programming in Prolog.* Berlin:Springer Verlag, 1981.

[CODASYL78]
CODASYL, "Report of the Codasyl Data Description Language Committee," *Information Systems*, vol. 3, 1978.

[Conway72a]
Conway, R.W., Maxwell, W.L., and Morgan, H.L., "On the Implementation of Security Measures in Information Systems," *Communications ACM*, vol. 15, no. 4, pp.85-103, Apr 1972.

[Conway72b]
Conway, R.W., Maxwell, W.L., and Morgan, H.L., "Selective security capabilities in ASAP- A file management system," *Proc. AFIPS Spring Joint Computer Conference*, 1972.

[Cooprider79]
Cooprider, L.W., "The Representation of Families of Software Systems", CMU-CS-79-116, Carnegie-Mellon University, Apr 1979.

[Date77]
Date, C.J., *An Introduction to Database Systems.* Addison-Wesley, 1977.

[Davida80]
Davida, G.I. and Turn, R., *Proceedings of the 1980 Symposium on Security and Privacy.* Oakland, USA:Technical Committee on Security and Privacy IEEE Computer Society, April 1980.

[Denning77]
Denning, D.E. and Denning, P.J., "Certification of Programs for Secure Information Flow," *Communications ACM*, vol. 20, no. 7, pp.504-512, July 1977.

[Denning76]
Denning, D.E.R., "A Lattice Model of Secure Information Flow," *Communications ACM*, vol. 19, no. 5, pp.236-242, May 1976.

[Denning80]
Denning, D.E.R. and Schlorer, J., "A Fast Procedure for Finding a Tracker in a Statistical Database," *ACM Transactions on Database Systems*, vol. 5, no. 1, pp.88-102, Mar 1980.

[Denning82]
Denning, D.E.R., *Cryptography and Data Security*. Addison-Wesley Publ. Company, 1982.

[Dennis66]
Dennis, J.B. and VanHorn, E.C., "Programming Semantics for Multiprogrammed Computations," *Communications ACM*, vol. 9, no. 3, pp.143-155, Mar. 1966.

[DeRemer76]
DeRemer, F. and Kron, H., "Programming-in-the-large versus Programming-in-the-small," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, June 1976.

[DES77]
DES, "Data Encryption Standard", Fed. Inf. Process. Stand. Publ. 46, National Bureau of Standards, Jan. 1977.

[Dewar79]
Dewar, R.B.K., Grand, A., Liu, S.C., Schwartz, J.T., and Schonberg, E., "Programming by Refinement, as exemplified by the SETL Representation Sublanguage," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp.27-49, July 1979.

[Dewar82]
Dewar, R.B.K., Schonberg, E., and Schwartz, J.T., *High-Level Programming - An introduction to the Programming Language SETL*. New York:Courant Institute of Mathematical Sciences, July 1982.

[Diffie76]
Diffie, W. and Hellman, M., "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp.644-654, November 1976.

[Dijkstra66]
Dijkstra, E.W., "The Structure of the 'THE'- Multiprogramming System," *Communications ACM*, vol. 11, no. 5, pp.341-346, Mar. 1966.

[Downs77]
Downs, D. and Popek, G.J., "A Kernel Design for a Secure Database Management System," *Proc. 3rd Int. Conference on Very Large Data Bases*, pp.507-514, October 1977.

[Downs80]
Downs, D., *Security in Database Management Systems.* Los Angeles:Ph.D. Thesis University of California, 1980.

[Egge80]
Egge, P.R., "Overview of the 'Ina Jo' specification language," *Tech. Rep. SP-4082*, Oct 1980, System Development Corporation.

[England72]
England, D.M., "Operating System of System 250," in Infotech State of the Art Report on Operating Systems (1972).

[Fabry74]
Fabry, R., "Capability-Based-Addressing," *Communications ACM*, vol. 17, no. 7, pp.403-412, July 1974.

[Fagin77]
Fagin, R., "Multivalued Dependencies and a New Normal Form for Relational Databases," *ACM Transactions on Database Systems*, vol. 2, no. 3, pp.262-278, September 1977.

[Fak83]
Fak, V.A.(editor), *Security, IFIP/sec '83.* North-Holland, 1983.

[Feiertag79]
Feiertag, R.J. and Neumann, P.G., "The Foundations of a Provably Secure Operating system," *AFIPS*, vol. 48, pp.329-334, 1979, SRI International.

[Fenton74]
Fenton, F.C., "Memoryless Subsystems," *Computer Journal*, vol. 17, no. 2, pp.143-147, May 1974.

[Fernandez75]
Fernandez, E.B., Summers, R.C., and Lang, T., "Definition and Evaluation of Access Rules in Data Management Systems," *Proc. First Int. Conference on Very Large Data Bases*, pp.268-285, 1975.

[Fernandez81]
Fernandez, E.B., Summers, R.C., and Wood, C., *Database Security and Integrity.* Addison-Wesley, 1981.

[Goldberg]
Goldberg, A., "Introducing the Smalltalk-80 System," *BYTE*, pp.14-26, August 1981 .

[Goldsmith81]
Goldsmith, L.H., "Dynamic Protection of Objects in a Computer Utility," *Technical Report CSRG-130*, Apr 1981, University of Toronto.

[Graham72]
Graham, G.S. and Denning, P.J., "Protection-Principles and Practice," *Proc. Spring Joint Computer Conference*, vol. 40, 1972, AFIPS Press.

[Graham68]
Graham, R.M., "Protection in an Information Processing Utility," *Communications ACM*, vol. 11, no. 5, pp.365-369, May 1968.

[Griffiths76]
Griffiths, P.P. and Wade, B.W., "An Authorization Mechanism for a Relational Data Base System," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp.242-255, September 1976.

[Habermann79]
Habermann, A.N., "A Software Development Control System", Carnegie-Mellon University, 1979.

[Harrison76]
Harrison, M.A., Ruzzo, W.L., and Ullman, J.D., "Protection in Operating Systems," *Communications ACM*, vol. 19, no. 8, pp.461-470, August 1976.

[Hartson75]
Hartson, H.R., *Languages for Specifying Protection Requirements in Data Base Systems- A Semantic Model.* Ph.D. Thesis Ohio State University, 1975.

[Hartson76]
Hartson, H.R. and Hsiao, D.K., "Full Protection Specifications in the Semantic Model for Database Protection Languages," *Proc. 1976 ACM Annual Conference*, pp.90-95, October 1976.

[Herschberg84]
Herschberg, I.S. and Paans, R., "Programmeren is kraken," *Informatie*, vol. 26, no. 9, pp.690-698, September 1984.

[Hilhorst83]
Hilhorst, G., Jonge, W. de, and Krijnen, B., "About the Take-Grant model", Informatica Rapport IR-84, Vrije Universiteit, Amsterdam, Apr. 1983.

[Hoffman69]
Hoffman, L.J., "Computers and Privacy: A Survey," *ACM Computing Surveys*, vol. 1, no. 2, June 1969.

[Hoffman70]
Hoffman, L.J., "The Formulary Model for Access Control and Privacy in Computer Systems", Rep No 117, Stanford University, California, May 1970.

[Hoffman77]
Hoffman, L.J., *Modern Methods for Computer Security and Privacy*. Englewood Cliffs, N.J.:Prentice Hall, 1977.

[Hsiao79]
Hsiao, D.K., Kerr, D.S., and Madnick, S.E., *Computer Security*. New York:Academic Press, Inc, 1979.

[Ichbiah79]
Ichbiah, J.D. (ed), "Preliminary Ada Reference Manual," *ACM SIGPLAN Notices*, vol. 14, no. 6 part A, June 1979.

[Iliffe62]
Iliffe, J.K. and Jodeit, J.G., "A Dynamic Storage Allocation Scheme," *Computer Journal*, vol. 5, no. 3, pp.200-209, October 1962.

[INGRES]
INGRES, *Reference Manual*. Berkeley, California:Relational Technology Incorporated.

[Jones73]
Jones, A., "Protection in programmed systems", PH-D Thesis, Carnegie-Mellon University, 1973.

[Jones75]
Jones, A.K. and Wulf, W.A, "Towards the Design of Secure Systems," *Software, Practice and Experience*, vol. 5, pp.321-336, 1975.

[Jones76]
Jones, A.K., Lipton, R.J., and Snyder, L., "A Linear Time Algorithm for Deciding Security," *Proc. 17-th Anual Symp. on Found. of Comp. Science*, 1976.

[Jones78]
Jones, A.K., "Protection Mechanism Models Their Usefulness," pp. 237-253 in Foundations of Secure Computation, ed. R.A. Demillo et al., Academic Press (1978).

[Jonge83]
Jonge, W. de, "Compromising Statistical Databases Responding to Queries about Means," *ACM Transactions on Database Systems*, vol. 8, no. 1, pp.60-80, 1983.

[Jonge85]
Jonge, W. de, *Security and privacy in information systems: some theoretical aspects*. Amsterdam:Ph.D. Thesis, Vrije Universiteit, June 1985.

[Kamer82]
Kamer, Tweede.ds [J 17207, nrs 1-2, "Wet op de persoonsregistraties. (Ontwerp van Wet)," *ACM Transactions on Database Systems*, Zitting 1981-1982.

[Kersten]
Kersten, M.L., Wasserman, A.I., and Riet, R.P. van de, *TROLL: Reference Manual.* Vrije Universiteit.

[Kersten81a]
Kersten, M.L., Riet, R.P. van de, and Jonge, W. de, "Privacy and Security in Distributed Data Base Systems," pp. 229-242 in Distributed Data Sharing Systems, ed. W. Litwin, North-Holland, Amsterdam (June 1981).

[Kersten81b]
Kersten, M.L. and Wasserman, A.I., "The Architecture of the PLAIN Data Base Handler," *Software, Practice & Experience*, vol. 11, pp.175-186, Feb. 1981.

[Klauw84]
Klauw, G. v.d. and Meer, D. v.d., *Data Protection and Abstract Datatypes in the C-Programming Language.* Amsterdam:Ms. Thesis, Vrije Universiteit, May 1984.

[Krauss79]
Krauss, L.I. and Macgahan, A., *Computer Fraud and Counter Measures.* Prentice-Hall, 1979.

[Kreissig80]
Kreissig, G., "A Model to Describe Protection Problems," *Proc. IEEE Symposium on Security and Privacy*, pp.9-17, April 1980.

[Kuitenbrouwer79]
Kuitenbrouwer, F., "Het privacyreglement en andere beleidsaspekten van gegevensregulering," *Informatie*, pp.571-582, Oct 1979.

[Lampson69]
Lampson, B.W., "Dynamic Protection Structures," *Proc. Fall Joint Computer Conference*, vol. 35, pp.27-38, 1969, AFIPS Press.

[Lampson73]
Lampson, B.W., "A Note on Confinement," *Communications ACM*, vol. 16, no. 10, pp.613-615, October 1973.

[Lampson77]
Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J., "Report on the Programming Language Euclid," *ACM SIGPLAN Notices*, February 1977.

[Lampson81]
Lampson, L., "Password Authentication with Insecure Communication," *Communications ACM*, vol. 24, no. 11, pp.770-772, November 1981.

[Landwehr81]
Landwehr, C.E., "Formal Models for Computer Security," *ACM Computing Surveys*, vol. 13, no. 3, pp.247-279, Sep 1981.

[Landwehr83]
Landwehr, C.E., "The Best Available Technologies for Computer Security," *Computer*, vol. 16, no. 7, pp.86-100, July 1983.

[Leerkamp82]
Leerkamp, N., "Hoofdlijnen van de wet op de persoonsregistratie," *Informatie*, vol. 24, no. 7/8, pp.372-377, Jul 1982.

[Levitt79]
Levitt, K.N., Robinson, L., and Silverberg, B.A., *The HDM handbook.* Menlo Park Cal.:Computer Science Lab SRI, June 1979.

[Levy84]
Levy, H.M., *Capability-Based Computer Systems.* DIGITAL Press, 1984.

[Linden76]
Linden, T.A., "Operating System Structures to Support Security and Reliable Software," *ACM Computing Surveys*, vol. 8, no. 4, pp.409-445, December 1976.

[Lipton78]
Lipton, R.J. and Budd, T.A., "On Classes of Protection Systems," pp. 281-296 in Foundations of Secure Computation, ed. R.A. Demillo et al., Academic Press (1978).

[Liskov81]
Liskov, B. and others, "CLU Reference Manual," *Lecture Notes in Computer Science*, vol. 114, 1981, Springer Verlag.

[Liskov76]
Liskov, B.H. and Jones, A.K., "A Language Extension Mechanism for Controlling Access to Shared Data," *Proc. 2nd Int. Conference on Software Engineering*, pp.62-68, 1976.

[Liskov77]
Liskov, B.H., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU," *Communications ACM*, vol. 20, no. 8, pp.564-576, August 1977.

[Liskov78]
Liskov, B.H. and Jones, A.K., "A Language Extension for Expressing Constraints on Data Access," *Communications ACM*, vol. 21, no. 5, pp.358-367, May 1978.

[Lockman81]
Lockman, A. and Minsky, N., "Unidirectional Transport of Rights and Take-Grant Control", LCSR-TR-13, Rutgers University., New

Brunswick, May 1981.

[Manna81]
Manna, Z. and Pnueli, A., "Verification of Concurrent Programs: the Temporal Framework," pp. 215-273 in The Correctness Problem in Computer Science, ed. J. Strother Moore, Academic Press (1981).

[Manola75]
Manola, F.A. and Wilson, S.H., "Data Security Implications of an Extended Subschema Concept," *Proceedings Second USA-JAPAN Computer Conference*, pp.481-487, 1975.

[Martin73]
Martin, J., *Security, Accuracy, and Privacy in Computer Systems.* Englewood Cliffs, N.J.:Prentice-Hall, 1973.

[McCauley79]
McCauley, E.J. and Drongowski, P.J., "KSOS- The design of a Secure Operating System," *Proc. NCC* , vol. 48, pp.345-353, 1979, AFIPS.

[McLean85]
McLean, J., "A comment on the 'basic security theorem' of Bell and LaPadula," *Information Processing Letters*, vol. 20, no. 5, pp.67-70, February 1985.

[Minsky76]
Minsky, N., "Intentional Resolution of Privacy Protection in Database Systems," *Communications ACM*, vol. 19, no. 3, pp.148-159, Mar 1976.

[Minsky77]
Minsky, N., "Cooperative Authorization in Computer Systems," *Proceedings of the COMPSAC-77*, pp.729-733, November 1977.

[Minsky78a]
Minsky, N., "The Principle of Attenuation of Priviliges and its Ramifications," pp. 255-276 in Foundations of Secure Computation, ed. R.J. Lipton, Academic Press (1978).

[Minsky78b]
Minsky, N., "An Operation-Control Scheme for Authorization in Computer Systems," *Int.Jour of Comp.and Inf. Sci.*, vol. 7, no. 2, pp.157-191, 1978.

[Minsky81]
Minsky, N.H., "Locally Controlled Transport of Priviliges", LCSR-TR-17, Rutgers University, New Brunswick, June 1981.

[Morris73]
Morris, J.H., "Protection in Programming Languagse," *Communications ACM*, vol. 16, no. 1, pp.15-21, January 1973.

[Morris79]
Morris, R. and Thompson, K., "Password Security: A Case History," *Communications ACM*, vol. 22, no. 11, pp.594-598, November 1979.

[Needham78]
Needham, R.M. and Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," *Communications ACM*, vol. 21, no. 12, pp.993-999, December 1978.

[Organick72]
Organick, E.I., *The MULTICS System: An Examination of its Structure.* Cambridge Mass.:The MIT Press, 1972.

[Pagan81]
Pagan, F.G., in Formal Specification of Programming Languages, Prentice-Hall, Inc. (1981).

[Parker76]
Parker, D.B., *Crime by Computer.* New York:Charles Scribner's Sons, 1976.

[Parnas72]
Parnas, D.L., "A Technique for Module Specification with Examples," *Communications ACM*, vol. 15, no. 5, pp.330-336, May 1972.

[Pietro-Diaz83]
Pietro-Diaz, R. and Neighbors, J.M., "Module Interconnection Languages", Comp. Sci. Dep. University of Irvine, Cal., June 1983.

[Pollack81]
Pollack, F.J., Kahn, K.C., and Wilkinson, R.M., "The iMAX-432 Object Filing System," *Proceedings of the 8-th Symposium on Operating Systems Principles*, pp.137-147, December 1981.

[Pollack82]
Pollack, F.J., "Supporting Ada Memory Management in the iAPX-432," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.117-131, Mar 1-3 1982.

[Popek78]
Popek, G.J. and Farber, D.A., "A Model for Verification of Data Security in Operating Systems," *Communications ACM*, vol. 21, no. 9, pp.737-749, Sep 1978.

[Popek80]
Popek, G.J., Walker, B.J., and Kemmerer, R.A., "Specification and Verification of the UCLA Unix Security Kernel," *Communications ACM*, vol. 23, no. 2, pp.118-131, Februari 1980.

[Reed79]
Reed, D.P., "Implementing Atomic Actions on Decentralized Data," *Proc. Seventh ACM/SIGOPS Symposium on Operating Systems Principles*, 1979.

[Riet80]
Riet, R.P. van de, "Databanken, enkele onderzoeksaspecten," in Colloquium Databankorganisatie, Math.Centrum, Amsterdam (1980).

[Riet81]
Riet, R.P. van de, Kersten, M.L., and Wasserman, A.I., "A Module Definition Facility for Access Control in Distributed Data Base Systems," pp. 255-272 in Distributed Data Sharing Systems, ed. W. Litwin, North-Holland, Amsterdam (June 1981).

[Riet83]
Riet, R.P. Van de, Kersten, M.L., Jonge, W. de, and Wasserman, A.I., "Privacy and Security in Information Systems using Programming Language Features," *Information Systems*, vol. 8, no. 2, pp.95-103, 1983.

[Robinson80]
Robinson, D.F. and Foulds, L.R., *Digraphs: Theory and Practice*. Gordon and Breach Science Publishers, 1980.

[Rubin84]
Rubin, H.I., *Integrated Development Environment*. Ms Thesis, University of California, Berkeley, 1984.

[Rushby83]
Rushby, J. and Randell, B., "A Distributed Secure System," *Computer*, vol. 16, no. 7, pp.55-67, July 1983.

[Saltzer75]
Saltzer, J.H. and Schroeder, M.D., "The Protection of Information Systems," *Proc. IEEE* , vol. 63, no. 9, pp.1278-1308, September 1975.

[Schmidt77]
Schmidt, J.W., *Some High-level Language Constructs for Data of Type Relation*. ACM Transactions on Database Systems, September 1977.

[Schroeder77]
Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel design project," *ACM SIGOPS Operating Systems Review*, vol. 11, no. 5, pp.43-56, November 1977.

[Snyder79]
Snyder, L., "Theft and Conspiracy in the TAKE-GRANT model," *CSD-TR 361*, 1979, Dep. of Comp.Sci. Purdue University.

[Snyder81]
Snyder, L., "Formal Models of Capability-Based Protection," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp.172-181, Mar 1981.

[Stepoway81]
Stepoway, S.L. and Silberschatz, A., "An extension to the Language-Based Access-Control Mechanism of Jones and Liskov," *ACM SIGPLAN*

Notices, vol. 16, no. 5, pp.54-58, May 1981.

[Stonebraker74]
Stonebraker, M. and Wong, W., "Access Control in a Relational Data Base Management System by Query Modification," *Proceedings ACM 1974 National Conference*, pp.180-186, November 1974.

[Stoy77]
Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, 1977.

[Tanenbaum81]
Tanenbaum, A.S. and Mullender, S.J., "An Overview of the AMOEBA Distributed Operating System," *ACM SIGOPS Operating Systems Review*, vol. 15, no. 3, pp.51-64, July 1981.

[Thomas76]
Thomas, J.W., *Module Interconnection in Programming Systems Supporting Abstraction.* PhD. Thesis, University of Utah, June 1976.

[Thompson81]
Thompson, D.H. and Erickson, R.W., *AFFIRM System Documentation.* Marina Del Rey:USC Information Sciences Institute, Februari 1981.

[Tichy79]
Tichy, W.F., "Software Development Control Based on Module Interconnection," pp. 29-41 in Proc. 4-th Int. Conference on Software Engineering (September 1979).

[Turn80]
Turn, R., "An Overview of Transborder Data Flow Issues," *Proc. Symposium on Security and Privacy*, pp.3-8, April 1980, IEEE Computer Society.

[Ullman80]
Ullman, J.D., *Principles of Database Systems.* Computer Science Press, 1980.

[Ware67]
Ware, W.H., "Security and Privacy in Computer Systems," *Proc. AFIPS Spring Joint Computer Conference*, pp.279-282, 1967.

[Wasserman79a]
Wasserman, A.I., "USE: a Methodology for the Design and Development of Interactive Information Systems," pp. 31-50 in Formal models and Practical Tools in Inf. Sys. Design, ed. Schneider,H.J., North Holland, Amsterdam (1979).

[Wasserman79b]
Wasserman, A. I., "Modularity in Data Base System Design: A Software Engineering View of Data Base Management," in Issues in Data Base Management, ed. H. Weber, North Holland (1979).

[Wasserman81]
Wasserman, A.I., Sherertz, D.D., Kersten, M.L., Riet, R.P. van de, and Dippe, M. , "Revised Report on the Programming Language PLAIN," *ACM SIGPLAN Notices*, vol. 16, no. 5, pp.59-80, May 1981.

[Wasserman82a]
Wasserman, A.I., Riet, R.P. van de, Kersten, M.L., and Leveson, N.G., "A Formal, Integrated Approach to Data and Usage Integrity in Health Information Systems," *IFIP Conf. on Data Protection in Health Information Systems*, pp.103-118, 1982, North-Holland.

[Wasserman82b]
Wasserman, A.I., "The User Software Engineering Methodology: an Overview," pp. 581-629 in Information Systems Design Methodology, ed. Verrijn Stuart, North Holland, Amsterdam (1982).

[Wegner72]
Wegner, P., "The Vienna Definition Language," *ACM Computing Surveys*, vol. 4, no. 1, pp.5-63, 1972.

[Weissman73]
Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," in Security and Privacy in Computer Systems, ed. Lance J. Hoffman, Melville Publ. Co., Los Angeles (1973).

[Weyuker78]
Weyuker, E.J., "Security in Operating Systems: separating the roles of rigths", Technical Report 003, Courant Institute New York University, 1978.

[Wijngaarden69]
Wijngaarden, A. van, Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A., *Report on the Algorithmic Language ALGOL 68*. Amsterdam:Mathematisch Centrum, 1969.

[Wirth80]
Wirth, N., *MODULA-2, Reference Manual.* ETH Zurich, Mar. 1980.

[Wood79]
Wood, C. and Fernandez, E.B., "Decentralized Authorization in a Database System," *Proc. 5-th Int. Conference on Very Large Data Bases*, pp.352-359, 1979, IBM Corp, Los Angeles Sci. Center.

[Wood77]
Wood, H.W., "The Use of Passwords for Controlling Access to Remote Computer Systems and Services," *AFIPS NCC* , vol. 46, pp.27-33, 1977.

[Wulf74]
Wulf, W.A., Cohen, E., Corwin, W., Jones, A.K., Levin,, R., Pierson, C., and Pollack, R., "HYDRA: The Kernel of a multiprocessor operating

system," *Communications ACM*, vol. 17, no. 6, pp.337-345., 1974.

[Zimmerman80]
Zimmerman, H., "OSI Reference Model- The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communication*, vol. 28, pp.425-432, Apr 1980.

1. Bij het reglementeren van het benaderen van objecten in een computersysteem dient onderscheid gemaakt te worden tussen het toegangsrecht en het gebruiksrecht.

(Hooofdstuk 4 van dit proefchrift)

2. Een formeel 'access-control' model is noodzakelijk voor het ontwerp van een veilige programmeertaal en zijn omgeving.

(Hoofdstuk 6 van dit proefschrift)

3. De combinatie van 'information hiding' en 'closed scopes' in PLAIN vormt een goede basis voor de constructie van veilige informatiesystemen.

Lit: R.P. Van de Riet, M.L. Kersten, W. de Jonge and A.I. Wasserman, "Privacy and Security in Information Systems Using Programming Language Features," Information Systems, Vol. 8 nr. 2, pg. 95-103, Feb. 1983

4. De aanname van McLean dat 'security' een universeel begrip is en dat daardoor het Basic Security Theorem van Bell-LaPadula van beperkte waarde zou zijn, is onjuist.

Lit: J. McLean, "A comment on the 'Basic Security Theorem' of Bell and LaPadula," Information Processing Letters, Vol. 20 nr. 2, pg. 67-70, Feb. 1985.

5. Een 'optimistic concurrency control' techniek is ongeschikt voor database systemen die ten behoeve van prestatieverbetering geaggregeerde informatie bijhouden.

Lit: M.L. Kersten and H. Tebra, "Application of an Optimistic Concurrency Control Method," Software, Practice & Experience, Vol. 14 nr. 2, pg. 153-168, Feb. 1983.

6. Een 'zoom' mechanisme vormt een essentieel onderdeel van een gebruikersvriendelijke vraagtaal voor (relationele) databanken.

Lit: A.I. Wasserman and M.L. Kersten, "A TBE Tutorial," Vrije Universiteit, Dep. of Mathematics and Computer Science, Amsterdam, May 1984.

7. Het gebruik van trigrammen in plaats van woorden voor indicering van documenten is wat betreft het aantal referenties in de index pas een verbetering bij een gemiddelde documentgrootte van meer dan 40 paginas.

Lit: J. Chudáček, "Niet-grammaticale verwerking van natuurlijke talen in computers," Informatie, Jaargang 26 nr. 8, pg 594-600, juli/aug. 1984.

8. Het 'marking' concept in de programmeertaal PLAIN kan efficient geimplementeerd worden.

Lit: R.P. van de Riet, A.I. Wasserman, M.L. Kersten and W. de Jonge, "High-Level Programming Features for Improving the Efficiency of a Relational Database System," ACM Transactions on Database Systems, Vol. 6 nr. 3 pg. 464-485 Sep. 1981.

9. Het exploratief prototypen van informatiesystemen kan het best geschieden door gebruik te maken van software op microcomputers.

   Lit: C. Floyd et al., "A Systematic Look At Prototyping," Proceedings Workshop on Approaches to Prototyping, Namen (Belgie) pg 1-19, Oct 1983.

10. De wettelijke verplichting om een kind binnen drie dagen na de geboorte bij de burgerlijke stand aan te geven is voor de meeste moeders een niet haalbare termijn en dus een beperking in haar recht het kind zelf aan te geven.

    Lit: Nieuw Burgerlijk Wetboek art. 18.

11. Het gebruik van andermans naam en password voor het zich met bedriegelijk oogmerk toegang verschaffen tot computers kan beschouwd worden als het aannemen van andermans identiteit en zou dienovereenkomstig strafbaar gesteld moeten worden.

    Lit: Wetboek van Strafrecht art. 225 en art. 326.

12. Uit oogpunt van bezuinigingen bij het onderwijs is te verwachten dat de term student vervangen zal worden door leerling.