AN ACCESS CONTROL MODEL BASED ON TIME AND EVENTS

By

FELIX P. JAGGI

Dipl. Ing. ETH, Swiss Federal Institute of Technology Zurich, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1990

15

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date July 23, 1990

Abstract

A new access control model incorporating the notion of time and events is introduced. It allows the specification of fine-grained and flexible security policies which are sensitive to the operating environment. The system constraints, expressed in terms of access windows and obligations, are stored in extended access control lists. The addition of a capability mechanism gives another dimension of protection and added flexibility, so that the flexibility and expressive power of the system constraints is fully supported by the underlying mechanism. The approach is compared to several existing models and its expressive power is demonstrated by showing the new model can be used to specify different existing security models as well as some special problems. The model is then adapted to work in a distributed environment.

Table of Contents

List of Tables

List of Figures

Acknowledgement

First, I would like to thank my supervisor, Sam Chanson, for his guidance throughout my studies and for helping to clarify my ideas.

I extend my thanks to Son Vuong for reading the final draft.

However, the highlight of my stay in Vancouver was the hospitality and friendship I experienced, especially in this department. I will miss the good times I had here and will never forget them.

Finally, I would like to thank the Faculty of Graduate Studies and the Department of Computer Science for their financial support.

1 Introduction

In this chapter we first show the need for computer security and outline the different fields in this area. A more detailed view of the work in a specific field - access control - is then given and general problems of existing approaches are discussed. The objectives and motivations of our research are presented followed by a brief review of related work. A short overview of the thesis concludes this chapter.

1.1 The Need and Structure of Computer Security

Computer security became an important issue in the 60's when multi-user, time-sharing systems were first developed. For the first time resources could be used concurrently by more than one user. Therefore all misuse of information stored in a computer system had to be prevented, i.e., data had to be protected from unauthorized access and from being passed through unauthorized channels, and users had to prove their identities to the system. With the appearance of computer networks in the 70's and 80's an additional dimension was added to the security problems.

Computer security is a wide area encompassing many related fields, including access control, information flow, inference control and cryptographic control [Denning 79]. Cryptographic controls are used to prevent accidental or malicious disclosure of data and are also of importance for authentication; inference control deals with the problem of analyzing and drawing conclusions from declassified and statistical information that is derived from more classified and more detailed information (mainly in databases). Information flow control is used to define channels along which information is allowed to propagate. It is somewhat related to access control, which deals with the protection of data, i.e., the regulation of access to objects. Another field that is of importance for computer security is verification, i.e., the control that specifications (of security-relevant programs) are implemented correctly.

1.2 Motivation

The need for flexibility and fine-grain control was initially motivated by the requirements for a protection system for the complex environment of general purpose distributed computing systems, but can be equally useful for other applications as illustrated in chapter 4. As shown below, none of the existing access control models are suitable for a wide variety of applications and thus none of them satisfies our requirements.

In this thesis we present a new access control model which accommodates these requirements by incorporating the notion of time and events into access control, i.e., pre-conditions and post-conditions for using an access right can be expressed by time and events. These conditions allow a more fine-grained and flexible specification of access control policies than that provided by existing systems. To support the flexibility and generality of the model a highly flexible and fine-grain underlying mechanism which consists of a combination of capabilities and access control lists is used.

1.2.1 Approaches to Access Control

Existing systems can be roughly classified into three categories of access control models:

- discretionary access control
- multilevel security
- commercial access control

1.2.1.1 Discretionary Access Control

Discretionary access control puts the decision on who may access an object into the hands of the owner (creator) of the object who may then pass access rights (including owner right) to other subjects. It is normally based on some variation of the access matrix which includes access control lists, capability-lists, and individual capabilities which can be obtained by splitting up an access matrix column-wise, row-wise, or entry-wise respectively (see below).

The notion of an access matrix was introduced in [Lampson 74]. An access matrix is a two-dimensional array with one row for every subject and one column for every object in the system. Each entry in the matrix contains the rights a subject (determined by the row) has on an object (determined by the column). Together with the access matrix rules are specified on how the matrix may be changed (e.g. transport of access rights to other subjects, creation of objects). However these rules are usually not considered to be part of the model and therefore not stated explicitly.

A practical problem with access matrices is their sparsity, as there are (especially for larger systems) no rights specified for most combinations of subjects and objects and thus most of the entries in the matrices are empty. Furthermore, access matrices are not well suited for distributed environments, as they are a centralized concept. In most implementations, the access matrix is therefore spilt up either into access control lists (one per object containing a list of subjects and the rights they possess), capability lists (one per subject containing a list of objects and the rights on them) or capabilities (each capability describes an access right for an object; a subject must possess the capability to be able to access the given object in the manner specified in the capability). The concepts of access control lists and capabilities already existed before access matrices were introduced (capability-lists were introduced as a way to implement the capability-concept). A capability-like concept was introduced in [Dennis 66]. Capability-based systems (supporting either capability-lists or capabilities) are suited for small protection domains (e.g. protected subsystems) and provide a simple means to distribute access rights. Access control lists, which are normally stored close to the object, appeared around the same time [Wilkes 68], and provide a simple database of the subjects that can access an object and allow simple revocation of access rights. There are also systems that used both the access control lists and capabilities; we will discuss them later.

However by breaking up the access matrix into many pieces (access control lists or capabilities) and distributing these pieces, some information (e.g. who may access a given object (in a capability system), or which objects can be accessed by a given subject (in systems using access control lists)) is more difficult to obtain and therefore new problems, such as revocation and review of capabilities are introduced and have to be solved.

A general problem for models based on access matrices is that they are not sensitive to their environment or to events that have occurred earlier. While this has the virtue of simplicity it also prevents the definition of more sophisticated security policies such as application-dependent access control policies.

1.2.1.2 Multilevel Security

Multilevel secure systems protect information of different classifications (e.g. classified, secret) from users with different clearances in the same system.

One of the first models incorporating multilevel security is the Bell/LaPadula model [Bell 76]. It uses an access matrix for the discretionary part of the access control but adds a clearance level and category to every subject and a classification and category to every object. A ‹clearance/classification, category›-pair dominates another one if its clearance/classification is higher than or equal to that of the second and if its category includes that of the second pair. A read access is only permitted if the subject has the right in the access matrix and the subject's ‹clearance, category›-pair dominates the object's ‹classification, category›-pair (this is known as the simple security-property). Also a domination of the subject's ‹clearance, category›-pair by the object's ‹classification, category›-pair is necessary for write access except for trusted subjects (known as the *-property).

Another multilevel security model for military message systems was defined in [Landwehr 84]. Besides having clearance and classification levels, an access set (access control list) is associated with every object. Each element of the access set contains a subject, an operation and an index number, indicating as which parameter the specified subject can use the associated object with the operation. This model also supports a multilevel object structure, i.e., objects can reside within others but the classification of the outer object must be at least as high as the classification of all objects inside it. There are two ways objects residing within another object can be accessed. In one scheme, objects can be accessed when the requester has the appropriate clearance. In the second scheme, not only the clearance for the object inside is needed, but the clearance for all the objects it resides in is also required. Furthermore, information copied from an object inherits the classification of that object.

The military message system model partially solves some of the problems of the Bell/LaPadula model since it allows the specification of application-dependent access control policies. However, it is customized for military message systems and therefore less useful for other environments such as commercial, research, development or educational ones.

1.2.1.3 Commercial Access Control

In commercial access control more emphasis is placed on the integrity of objects (and the information stored within them) and the way data are changed.

Lipner formulated an access control policy for commercial environments based on the Bell/LaPadula model by putting the system-management in a different security level than the rest of the system and by putting different departments and functions of a general data processing environment (production data, development, system development and tools) into different categories [Lipner 82]. Different types of users are assigned different clearance levels and different sets of categories. However this approach primarily distinguishes different classes of users or programs, but not individual members within a class, i.e., it does not really allow application-dependent access control.

Clark and Wilson formulated an integrity model for the commercial environment [Clark 87]. All objects have to be in a valid state, and may only be changed by certified transactions which move a object from a valid state to another valid state. Furthermore, there is an access control list defining which user may use which transactions with which parameters. The model also emphasizes that certification of any of the transactions mentioned above by a security officer before the installation into the system, as well as the static separation of duty between certifiers and users, and the enforcement by the system are crucial for the security of the system. The problem of the Clark/Wilson model is that while it is well suited for a production environment it is also very inconvenient for research, development and educational environments. This is because in such environments the exact definitions of who may use which data and programs and in what way would create a large overhead without there being a need for this.

Other approaches to separation of duty can be found in [Brewer 89] (dynamic separation of duty) and [Nash 90] (object-based separation of duty).

1.2.2 General Problems with Existing Models

A major problem of existing security models is that they are either designed to solve some specific problems (e.g. military message systems) or meet some special security requirements (e.g. Orange Book [DOD 83]) or are kept simple if they are designed to be general (e.g. capabilities) and can therefore only address basic problems but not more sophisticated ones. Therefore, none of these models can be used to satisfy the needs of general purpose, distributed computing systems. This means that we can either separate the security mechanisms (and models) from the operating system, and thus choose add-on security mechanisms depending on the actual needs of a system or develop a more flexible mechanism that is part of the operating system and can satisfy the needs of different applications. We chose to follow the second path.

There also exist security problems that are not addressed by existings models or are too specialized that they are difficult to be solved by them (e.g. n-person rules). Furthermore, few of the existing models allow conditions (pre- or post-conditions) to be attached to access rights. Although there exist a few approaches that may support these facilities (see section 1.3) they are usually mentioned as possibilities and no details are given.

1.3 Review of Related Work

Our model uses four major concepts to achieve its goals:

- combining access control lists and capabilities
- adding conditions to access rights
- the use of obligations
- using events to define security

A brief review of these concepts in related work follows.

1.3.1 Combining Access Control Lists and Capabilities

Combining access control lists and capabilities to provide a flexible, fine-grain access control model has been done very early. Lampson described such a system (using keys instead of capabilities) for the BCC computer [Lampson 69].

Other systems combine both approaches to achieve a highly flexible, fine-grain distribution of access rights (e.g. Multics [Organick 72]). In both Multics and the BCC computer, segment descriptors, which are similar to capabilities but not transferable, are used to access virtual memory. The segment are also protected by access control lists.

Gligor looked at combinations of capabilities and access control lists with respect to the review and revocation of capabilities and found them only partially useful [Gligor 79].

Karger and Herbert added access control lists to their capability architecture [Karger 84]. Their goal was to add support for multilevel security and the traceability offered by access control lists to a capability system while preserving its benefits (non-hierarchical protection domains, sealed objects etc.). When a capability is presented to the system, the access control list is checked, and access is granted only if it is permitted by the access control list.

They also mentioned a similar approach taken by IBM System/38 which supports authorized and unauthorized pointers (capabilities), where unauthorized pointers cannot be used without first checking an access control list for permission.

1.3.2 Adding Conditions to Access Rights

Ekanadham and Bernstein extended conventional capabilities into conditional ones [Ekanadham 79]. Each capability is associated with a set of locks which can only be opened by the corresponding condition keys, i.e., keys with conditions associated. A subject can access an object only if it has the keys to all the locks attached to the capability and if the associated conditions are fulfilled for all the keys. No details were given about the type of conditions that can be imposed.

Another approach to add conditions to access control based on predicates is described in [Hartson 84] for a database environment. In this scheme, an authorizer may define a predicate as the condition under which any member in a given set of users may execute any operation in a given set of operations on data items in a given set of data. The access conditions can come from three categories: system-dependent, query-dependent and data-dependent. The ones that are of interest to us are the system-dependent ones whose values can be determined from the general system state.

The Send-Receive mechanism [Minsky 84] attaches conditions to the transport of tickets (access privileges or capabilities). To successfully pass a ticket, not only does the sender have to send the ticket, but the receiver has to accept it by executing a receive operation. To execute either the send or the receive operation, a subject must have the privilege to do so. These privileges are expressed by can-rules. These rules allow a subject to send (receive) a ticket of a certain type to (from) another subject only if a precondition is fulfilled. The scope of the arguments of these conditions is limited to the information stored in the ticket, the object associated with the ticket, the receiver (sender), and the system parameters that are easily available (e.g. time). The can-rules are permanently assigned to subjects.

While all the conditions discussed here were conditions that have to be fulfilled before the access is granted, conditions that must be fulfilled after an access has occurred are discussed in section 1.3.4.

1.3.3 Using Events to Define Security

There exist some security models that use events for their definition of security. However, they are not used primarily to define conditions, but to restrict information flow. A brief description of some of them is given below.

Goguen and Meseguer based their model of non-interference [Goguen 84] on events. The basic idea of this model is to reduce the flow of information by giving a set of non-interference assertions. Such an assertion states that what one group of users does has no affect on what another group sees. Assertions are extended by conditions, based on events, which have to be satisfied for the assertion to be valid.

Johnson and Thayer proposed a more general description of the approach taken by Goguen and Meseguer [Johnson 88]. Instead of attaching conditions to the non-interference assertions, they define a set of event sequences which may be seen by subjects.

1.3.4 The Use of Obligations

Minsky and Lockman [Minsky 85] added obligations to permissions. In contrast to the conditions mentioned above which are pre-conditions for allowing access, obligations are conditions that have to be fulfilled after the access has occurred, i.e., within a time period[1] of using an access right, a subject must (or must not) execute a set of operations, otherwise a specified sanction will come into effect. While the sanction for violating pre-conditions is simple, namely the denial of access, sanctions for obligation violation can be quite complex. These may include

[1] Time period may be specified by the occurrence of some events.

execution of a correction procedure, roll-back of a transaction, an emergency measure, imposing another obligation, or revocation of some rights.

Obligations allow a user to violate integrity constraints for a limited time (until the obligation is fulfilled) and thus makes it possible to build constructs similar to the concept of transactions in database systems. Obligations also provide the means to give users access under the condition that they will not misuse this privilege later.

Jacob mentions systems that guarantee punishment for illegal acts as an area of future research [Jacob 89].

1.4 Outline of the Thesis

The rest of the thesis is organized as follow: Chapter 2 contains a description of the proposed model, including justification of the approach of combining capabilities and access control lists in the underlying mechanism. Examples illustrating the power and flexibility of the model are presented in chapter 3. Chapter 4 deals with the adaptation of the model for a distributed environment and chapter 5 summarizes the thesis and lists future research. In the appendices the model introduced in chapter 2 is presented in a more formal manner and some more details of the the model are given.

2 Model Description

In the last chapter we showed the need for an access control model that allows the specification of fine-grained and flexible access control policies. In this chapter we first give an overview of our model and define a group structure for the subjects. Then each component of the model including the enforcement mechanism is described in detail. This is followed by a description of the way initial access control lists are assigned in our model and the basic model operations such as passing an access right and deleting an object. The chapter concludes by giving a justification for the use of both the capability and the access control lists in the underlying mechanism.

2.1 Components of the Model

Our security mechanism consists of three major components:

- an access control list for every object,
- capabilities (rights on objects),
- event and time manager (enforcement mechanism).

A capability contains an object identifier and an access right to an object. However, for a subject to gain access to an object, in addition to possessing the appropriate capability, the conditions for access as specified by the associated access window must also be satisfied. Furthermore, any valid obligations must be fulfilled within a specified period of time.

The access control list also serves as a database for the access windows and the obligations (we shall refer to these as system constraints). An entry in the access control list of an object contains the access window and the obligations associated with a given access right for a given user.

The capabilities provide flexible management of the distribution of the access rights, and also provide an additional dimension of security to the model.

The last element of the security mechanism is the time and event manager which is responsible for the enforcement of the system constraints. Part of its functions – the audit of all events – are usually included in more sophisticated security systems. In our model, however, this information is also used to make dynamic access control decisions.

Each of these components is discussed in detail below, but first we shall describe how subjects can be grouped, and the relationship between programs, processes and users.

2.2 Grouping of Subjects and the Relation between Programs and Processes

Subject groups are used to group subjects with similar functionality, be it users in the same section within an organization (e.g. members of a department or a project team) or programs that perform similar system functions (e.g. compilers, editors or system programs in general) or different instances of a program.

The reasons for introducing subjects groups in our model are:

- the use of groups reduces the number of explicit entries in the access control lists
- it offers more convenient security policy definition in that it is no longer necessary to know all subjects in advance, especially for defining system constraints
- it allows easier distribution of access rights as access rights can be passed to groups instead of individual subjects

2.2.1 Group Structure

The group structure in our model is based on a tree structure, i.e., all nodes below a certain node belong directly or indirectly to the group represented by that node. We did not consider more sophisticated group structures, such as combinations of trees and inverted trees, as the grouping is only of minor importance in our model. When different types of group structures are used, the merging of system constraints when access rights are passed would have to be adjusted (see section 2.8.4). The nodes in the group tree may be of the following types:

- group: a group node is a collection of other groups, programs, users and processes (see Figure 2.1). There is also a special group ALL, which is the root of the tree and includes all subjects (see Figure 2.1)
- program: a program node may only,have processes as members (for description of program see section 2.2.2), i.e., a program node without any process instance must be a leaf
- user: a user node represents a human user, and must be a leaf in the group tree
- process: a process node must be a leaf in the group tree (see section 2.2.2)

An example of a group tree is shown in Figure 2.1.

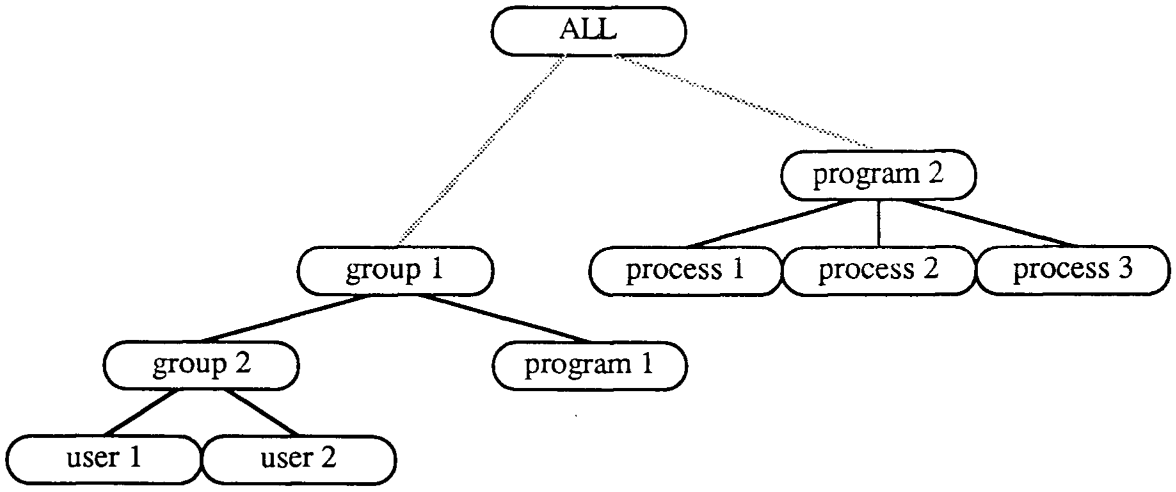

Figure 2.1: Example of a Group Tree

2.2.2 Programs / Processes

Programs have two roles to play. On the one hand they are objects, but on the other hand they are also nodes in the group tree with the created process instances as members. Due to their special role, programs can possess capabilities which, like the system constraints, are inherited by every process that is an incarnation of that program.

Processes do not run in the protection domain of the invoking user. This allows processes and their invokers to have different access rights. A user who invokes a program thereby creating a process has to pass access rights to the process (or to the corresponding program) if he wishes to give some specific access rights to the process.

2.2.3 Group Management

The creation and management of groups and users (e.g. who may join a group) are organizational matters and are therefore not considered further. On the other hand, processes automatically belong to the group of the associated program upon creation. Also every group and subject belongs to the group ALL if it does not belong to another group.

Groups can only be removed if they have no members, i.e., only leaves of the group tree can be removed. The reason for this is the complications that will arise when system constraints are defined using groups.

2.3 Event and Time Manager

As mentioned before, access windows and obligations are both defined by events and time. While the use and enforcement of time is conceptually relatively simple, the introduction of events complicates the enforcement of access windows and obligations. On the other hand, the notion of obligations is more naturally related to events, and access windows would lose much of their expressive power if they had to be defined solely in time.

The fundamental task of the time and event manager is the collection of relevant events. The sequence of such events forms the history of the system. The enforcement issue is discussed in section 2.6.

Representation of Time and Events

Time can be represented by any kind of regular continuous enumeration (e.g. clock ticks, date and time), but we will assume date and time from now on. The granularity of time has to be chosen appropriately to satisfy the needs of the system.

When stating a security policy it is necessary to specify events (called description events). These events are compared with the events in the history that actually occurred (real events) to enforce the security policy. As description events and real events serve different functions in our model, they are represented slightly differently.

A real event consists of a subject, an action and associated parameters (which can be subjects, objects or capabilities) and the time the action occurred.

The actions that are considered important for security include logging in, logging out, executing a program, the start of the corresponding process, the end or abortion of a process, reading, writing or deleting an object, passing of access rights (consisting of the actions to initiate the pass, update the access control list, and receive capability, all of which can also occur independently in other circumstances such as creating an object), and discarding of capabilities. Creating an object is considered executing a specific program, and hence not itself a security-relevant action.

Description events consist of a subject (which may be a group), an action and its parameters (again subjects (which may be a group), objects or capabilities), but do not contain a timestamp. To describe events more conveniently, we have added special values for subjects and parameters.

The additional values for subjects are SELF, representing the subject that is acting; and OTHER meaning any subject except the acting one. SELF is used to describe events related to the acting subject, yet maintaining the flexibility of the description when it is transferred to another subject[2] (see section 2.8.4); and OTHER is just the complement of SELF.

The special parameters are ANY, indicating any object or capability, SELF indicating the acting subject, and OTHER being again the complement of SELF.

2 When passing a system constraint containing events expressed in terms of SELF or OTHER from one subject to another, it is important to realize that the time when the real events corresponding to these description events occur may be different for the two subjects, as the events are relative to the acting subject. It is shown in appendix C how one can ensure that the absolute size of the access windows are not increased, and the obligations are not reduced when passing access windows and obligations containing SELF or OTHER.

2.4 Capabilities

The following rights are defined on an object: read, write, delete, and execute if the object is a program. This basic list can be extended to include more sophisticated rights, but is sufficient for the purpose of describing our model.

The capabilities in our model are unforgeable and permanent, i.e., they can only be created by the system and they do not disappear unless explicitly deleted by the holder of a capability. Therefore capabilities have be stored in non-volatile memory.

A subject can read or delete its capabilities without restrictions. However, a subject must invoke the system to pass a capability to another subject. This is because an entry in the access control list of the associated object must be created or updated according to some rules (see section 2.8.4). The possession of the capability itself does not guarantee access.

2.5 Access Control List

Each object has an access control list. Conventionally, each entry in the list consists of a tuple ‹subject, access right› which describes how a given subject may access the object. In our scheme, the access control list is expanded so that it also contains the access window and the set of obligations connected to every ‹subject, access right›-pair for which an entry exists (see Figure 2.2).

subject	access right	access window	set of obligations

Figure 2.2: Structure of an Entry in the Extended Access Control List

Each single access window (component of access window; see section 2.5.1) and also each obligation has two flags indicating whether it is copiable and overwriteable. Those that are not overwriteable can only be defined by the authority of the object (see section 2.7.1).

When using the execute access right to an object, it is useful to specify obligations in terms of the process just created. For this purpose we allow the use of the special subject PROCESS in stating system constraints for programs, to indicate the process created by the instantiation of the program. Whenever a system constraint using PROCESS is used (e.g. access checking and passing of access right) every instance of PROCESS is replaced by the identifier of the process using it.

In this thesis we do not care about the way a process identifier is composed. But depending on the composition of this descriptor, one may also use the knowledge about this structure to express system constraints (e.g. if the invoker's identifier or the program's identifier are part of the process identifier).

2.5.1 Access Window

An access window consists of a set of single access windows each of which defines a period of time during which access to the object is allowed.

The single access window is defined by the following parameters:

- from_time,
- from_event,
- to_time,
- to_event, and
- base_time (time after which events are to be considered for fulfilling 'from_event' or 'to_event'). It can either be given an absolute value or relative to the time of the access request (e.g. only events that have occurred within the last 15 minutes preceding the access request are considered).

At least one of 'from_time' and 'from_event', and one of 'to_time' and 'to_event' have to be defined.

'from_event' and 'to_event' are sets consisting of ordered sets of description events.

For an access attempt to occur within a single access window, each of the following conditions has to be fulfilled if the corresponding parameter is specified (see Figure 2.3):

- actualtime ≥ 'from_time'
- actualtime ≤ 'to_time'
- all events of any of the ordered sets in 'from_event' have occurred in the correct order since 'base_time'
- all events of any of the ordered sets in 'to_event' have not occurred in the correct order since 'base_time'

Thus for an access to occur within a single access window, it must occur after 'from_time' and after 'from_event', and before 'to_time' and before 'to_event' if all four parameters are specified.

An access is successful if it occurs within any of the single access windows defined by the access window. It follows that if no access window is defined for a ‹subject, access right›-pair, the subject cannot access the object even if it holds the capability.

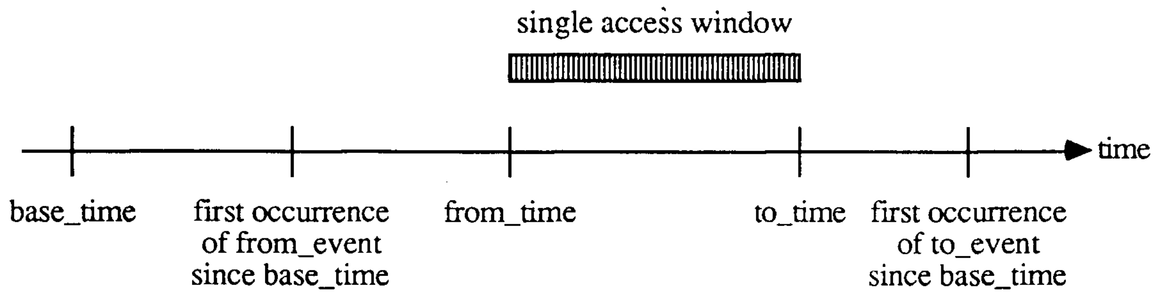
single access window
time
base_time
first occurrence of from_event since base_time
from_time
to_time
first occurrence of to_event since base_time

Figure 2.3: Example of a Single Access Window
(Note: the temporal order of 'base_time', the first occurrence of 'from_event', 'from_time', 'to_time', and the first occurrence of 'to_event' has been chosen randomly. The only ordering rule is that the first occurrence of 'from_event' and the first occurrence of 'to_event' have to occur after 'base_time')

2.5.2 Obligation

An entry in the access control list may contain zero, one or more obligations. An obligation consists of a set of obligation elements and a window (called obligation validity window) similar to the access window during which the obligation is in effect. In other words, an obligation will be triggered only if the access occurs within the associated obligation validity window. Every obligation is associated with a 'startevent' and a deadline, defining the time period within which the obligation has to be fulfilled (known simply as the obligation window), and sanctions which are taken if the obligation has not been respected.

We define two kinds of obligation elements: to do ... by ... (to-do obligation elements), and not to do ... until ... (not-to-do obligation elements).

An obligation element consists of a tuple - the first component indicates whether the element is a to-do element or a not-to-do element and the second component is an ordered set of description events. Each of these ordered sets specifies a series of events, and one of these series has to occur at or before the deadline (to-do element) or may not occur until after the deadline (not-to-do element).

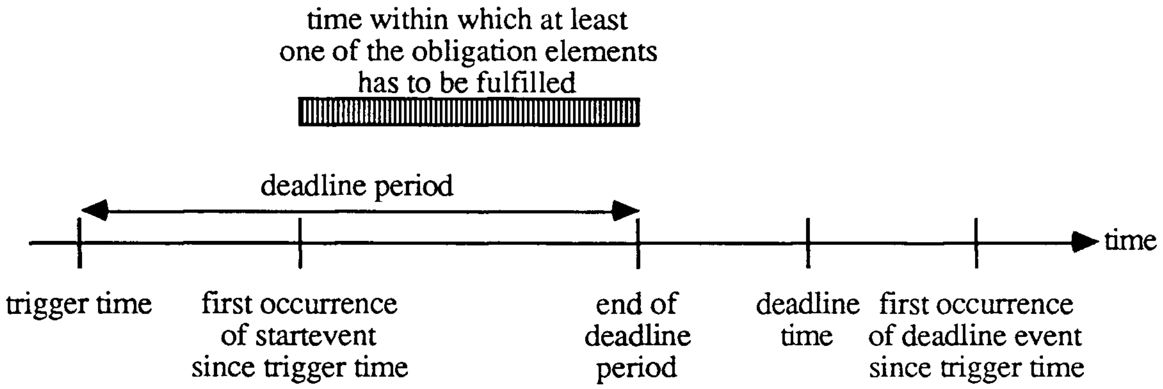
time within which at least
one of the obligation elements
has to be fulfilled
deadline period
time
trigger time
first occurrence of startevent since trigger time
end of deadline period
deadline time
first occurrence of deadline event since trigger time

Figure 2.4: Example of an Obligation Window
(Note: the temporal order of 'startevent', end of 'deadline period', 'deadline time', and 'deadline event' has been chosen randomly. The only ordering rule is that the first occurrence of 'startevent' and the first occurrence of 'deadline event' have to occur after 'trigger time')

An obligation is usually triggered by a successful access to an object (see section 2.6.1). If a 'startevent' (corresponds to 'from_event' of the single access window, with the time the obligation was triggered as the 'base_time') is specified then it marks the beginning of the obligation window. If 'startevent' is not specified, then the 'trigger time' starts the obligation window. The deadline marks the end of the obligation window and is defined to be the earliest occurrence of any of the following events that are specified (at least one must be specified, see Figure 2.4):

- 'deadline time' (corresponds to 'to_time' of the single access window)
- 'trigger time' plus a specified 'deadline period'
- 'deadline event' (corresponds to 'to_event' of the single access window)

An obligation is fulfilled if any of the obligation elements is fulfilled, i.e., events forced by a to-do element have occurred in the correct order within the obligation window, or a series of events prohibited by a not-to-do element have not occurred within the obligation window.

Sanctions may be specified in terms of other obligations (which do not need an obligation validity window as they are triggered as soon as the violation of the obligation is detected) or direct specific actions. The following types of penalty actions can be imposed:

- passing an access right (without restricting access windows and without additional obligations because these options are not suitable for automatic processing; the purpose of this is to force a subject to share its access right with other subjects),
- dropping a capability,
- executing a program (the program must be parameterless, as there is no mechanism of automatically passing parameters to a process; however, a process inherits the access rights from the associated program and may also receive access rights from other subjects),
- abort a process (only suitable for processes and programs),
- logging out (only applies for users, no effect otherwise),
- deleting an object.

Notice that log in, read and write accesses, receive a capability and update an access control list are not among the allowed penalty actions. While logging in and receiving a capability can hardly be seen as a penalty, the other actions would need further specifications (e.g. what value to write) so that they cannot stand by themselves but have to be used in some context, i.e., within a program.

Some restrictions must be placed on the subjects which execute the penalty actions. Consider the following situation:

Subject x creates an object a and passes the access right to itself adding the additional obligation that after it accesses object a, it has to access object b, otherwise subject y has to drop its capability to object c. If this could be done then x could prevent y from accessing any particular object.

This case shows that rules are needed on who may execute penalty actions and what kind of penalty actions can be defined in additional obligations that are added when passing access rights.

- initial access control list (see section 2.7.1) and manual updates:

  in initial access control lists and manual updates by an authority of the object (see section 2.7.1) penalty actions can be expressed for the following subjects:

  - SELF
  - CREATOR (only in entries for CREATOR and in the initial access control list)
  - any subject if it occurs in its own entry
  - PROCESS only if the entry is for a program
- additional obligation:

  penalty actions specified in additional obligations may only include the following subjects:

  - source subject
  - target subject, if target is not a group
  - SELF
  - PROCESS only if the target subject is a program

By limiting the subjects and special identifiers used for penalty actions, we can guarantee that only subjects which use the access right or were involved in the passing of that access right can be condemned to execute a penalty. In the initial access control lists, penalties can only be defined for subjects which access an object. Afterwards, only penalties for subjects sending or receiving an access right can be added. Thus no subjects can be penalized unless they are the ones defining the penalty themselves, or the penalty is defined on them in the initial access control list, or they are receivers of an access right.

2.5.3 Using Groups in System Constraints

Groups may be used in defining system constraints wherever a subject can be used except for the definition of penalty actions. We assume that any node below the group in the group tree can be used to replace the group in a real event.

The remaining problem is the change of group structure and group memberships during the period in which a sequence of description events occurs, though group changes will not be very common. When checking if an event has occurred we have to consider the group structure at the time the event occurred. This requires keeping a log of group structure changes. Some of these structure changes are already maintained for other purposes, i.e., the creation and deletion of processes. So we need only to keep track of subjects (excluding processes) joining and leaving groups, with the creation and deletion of these subjects as special cases. Group changes are typically not common events.

2.6 Checking of Access and Enforcement of Obligations

The tasks of the enforcement mechanism is to check all accesses, and to monitor the triggered obligations.

2.6.1 Access Checking

To attempt an access, a subject has to be in possession of the required capability. The access is granted if the request occurs within one of the associated single access windows.

If the access is granted, all obligations associated with this access right of the subject are also checked to see if the access occurred within an obligation validity window. If this is the case, the obligation is triggered by adding the access time and the identifier of the accessing subject to the obligation. The minimum of {'deadline time', 'trigger time' + 'deadline period'} is also calculated and attached to the obligation replacing the 'deadline time' and the 'deadline period' which are removed from the obligation. The obligation is then entered into the triggered obligation list (a dynamic database containing all triggered obligations). Once they are entered in this list, the obligation enforcement mechanism takes over.

The semantic of the group tree is as follows: when the system looks for the system constraints to an access right of a subject it checks to see if the ‹subject, access right›-entry is in the access control list. If this is not found, the system searches to use the system constraints of the next higher (i.e. parent) node in the group tree until it finds an entry or reaches the top of the tree (if there is no entry for ALL). In the latter case the access failed. The group system works such that the nodes further down the tree overrule the ones further up. This means that entries that are identical to the one of their parent node can be removed (see also appendix C).

2.6.2 Obligation Enforcement

The mechanism that enforces the obligations goes through the list of triggered obligations continuously and checks if an obligation has been fulfilled within the deadline, expired without the obligation being fulfilled, or if the obligation has been violated before the deadline.

Violations of an obligation element can occur in the following ways:

- an event or a series of events prohibited by a not-to-do obligation element has occurred
- an event or a series of events demanded by a to-do obligation element has not occurred

If all obligation elements (and thus the obligation) are violated then the sanctions are imposed and the corresponding entry in the list of triggered obligation is removed.

Obligations which have been fulfilled can also be removed from the list of triggered obligations.

2.7 Authority and Initial Access Control List

Authorities and initial access control lists are introduced to regulate the creation of objects and their protection.

2.7.1 Authority

Every object has an authority, determined at object creation time, which consists of a non-empty set of users. Every user is also associated with a non-empty set of users known as user-authority-set. The authority of an object is initialized to the user-authority-set of its creator if it is a user, or the user-authority-set of the user at the head of the invocation chain of the creating process. The management of authorities and user-authority-sets including membership criteria is an organizational task and is therefore not considered here.

The management of access control of an object in this model is thus distributed between

- the authority of the object, and
- the subjects that are in possession of the capabilities to the object.

The holders of the capabilities are responsible for distributing the access rights throughout the system. However the authority of an object will always have the final say as it can edit the contents of the access control list, or can prevent a user from getting access by specifying this in the initial access control lists (see section 2.7.2). The authority may also delete an object under its control without having to possess the delete capability (see section 2.7.2).

2.7.2 Initial Access Control List

As mentioned above, every user possesses a set of initial access control lists (managed by its user-authority-set) one of which is used for each object created by the user or on his behalf.

An initial access control list contains at most one entry per ‹subject, right›-pair. Initial access control lists also allow the specification of a special subject called CREATOR. This allows initial rights to be given to the creating subject without having to know its identity in advance.

When creating an object, the creating subject has to give (among other things) the object's initial access control list as a parameter. A user must be allowed to create an object with a certain initial access control list. Users are given a set of initial access control lists they can use. This set is managed by the user-authority-set of the user. As processes run on behalf of their invokers (namely, users at the head of the invocation chain) but not in their security domains, a process creating an object must use an initial access control list assigned to its invoker.

Once an object is assigned an initial access control list the special subject CREATOR in every entry should be replaced as follows: If there are entries for the creating subject, then they have to be merged with those for the CREATOR (see section 2.8.4), and the entries for the creating subject deleted. Finally, every occurrence of CREATOR is replaced by the identifier of the creating subject.

2.8 Operations

In this section we describe how the major operations related to our security model work.

2.8.1 Creating an Object

When creating an object we distinguish between two cases: creating a program (which is both a subject and an object), and creating a 'pure' object.

If a program is created by a user, the user-authority-set associated with the user becomes the authority of the object, otherwise the user-authority-set associated with the user at the head of the invocation chain of the creating program becomes the authority of the created object. An initial access control list is associated with the object according to the condition described in section 2.7.2. The creating subject also receives all applicable capabilities to the object created. The immediate creator can then also pass some of its access rights to the object, and in this way enables it to invoke other programs or access objects.

The case of creating a 'pure' object is similar except that the creator cannot pass any capabilities to the object and there is no 'execute' access right associated with the object.

2.8.2 Deleting a Subject

The deletion of a subject causes the following problem:

The last subject in possession of the delete-capability of an object may be deleted, thus leaving no subject with a delete capability to the object. This problem is dealt with by giving the authority of an object the right to delete the object whether or not it is in possession of the delete-capability.

This leads to the more tractable problem of guaranteeing that the authorities of all objects and the user-authority-sets of all users are non-empty. However, this is an organizational problem and is therefore not addressed in our model.

2.8.3 Accessing an Object

When a subject makes a request to access an object, the enforcement mechanism checks if there is an associated entry in the object's access control list. The access fails if there is no entry. However if the entry exists, the access request is checked to see if it occurs within any of the single access windows. If so the access is granted after every valid obligation is triggered. The access fails if no single access window allows it.

2.8.4 Passing of Access Rights

The passing of access rights from a source subject to a target subject can either occur explicitly as an individual operation or can be done as part of program invocation. As mentioned earlier, the passing of an access right also affects the associated system constraints of the target subject. Our model requires that the system constraints of the source and the target subjects be merged to form the new system constraints of the target subject in the following ways:

- RETAIN: keeps the system constraints of the target subject
- REPLACE: replaces the existing system constraints of the target subject (keeping those that are non-overwriteable) by the system constraints of the source subject (except those which are non-copiable)

- COMBINE: combines the system constraints of the source and the target subject (except non-copiable ones)

The merge modes for the access window and the obligations need not be the same, and are stated for each ‹subject, access right›-pair individually. When an access right is passed from a source subject to a target subject, the merge modes used are those defined at the target subject's end.

The impact of passing an access right from a source subject to a target subject on an access window (or on obligations) is described in Table 2.1. The columns 'source entry' and 'target entry' indicate if there is an explicit entry in the access control list for the source subject or the target subject.

A new entry is created if either the new access window or the obligations is different from that of the parent node in the group tree; the other system constraint is taken from the entry in the parent node.

Groups can receive but not pass access rights. They also cannot keep capabilities (except for programs, see section 2.2.2). Groups receiving an access right are treated like a regular subject (user or process) as far as creating and updating the access control list entry are concerned. In addition all existing access control list entries of the group members are updated as well (note that there is no need to create new entries for the group members). As passing an access right to a group consists of passing the access right to all members of the subtree, passing an access right to a group also fulfills description events requiring passing access rights to any member of this group; however none of the group members will receive a capability.

For all merge modes, the source subject can restrict (i.e., reduce) the access window it provides for the merging and can also add obligations in addition to its own. The ways to restrict an access window are described in appendix B.

source entry	target entry	merge mode[3]	action to be taken[4,5]
yes	yes	RETAIN	do not change entry of target subject
		REPLACE	replace entry of target subject by the system constraints passed by the source subject
		COMBINE	combine the system constraints passed by the source subject with entry of target subject
no	yes	RETAIN	do not change entry of target subject
		REPLACE	replace entry of target subject by the system constraints passed by the source subject (taken from entry of group closest to the source subject in the group tree)
		COMBINE	combine entry of target subject with the system constraints passed by the source subject (taken from entry of group closest to the source subject in the group tree)
yes	no	RETAIN	no entry for target subject is created
		REPLACE	create entry for target subject from the system constraints passed by the source subject
		COMBINE	combine system constraints passed by the source subject with entry of the group closest to the target subject in the group tree
no	no	RETAIN	no entry for target subject is created
		REPLACE	create target entry from system constraints passed by the source subject (taken from entry of group closest to the source subject in the group tree)
		COMBINE	combine system constraints passed by the source subject (taken from entry of group closest to the source subject in the group tree) with entry of the group closest to the target subject for new entry for the target subject

Table 2.1: Impact of Access Right Passing on Access Control List

3 If there is no entry for the target subject (target entry in table is 'no'), the merge mode is taken from the group closest to the target subject which has an entry. If no such group exists, no entry is created.

4 If there is no entry for the source subject (source entry in table is 'no') and no group in the group tree above the source subject for which an entry exists, we assume that there are no access window and no obligations to copy.

5 notice that non-copiable single access windows and obligations are not transferred to the target subject (REPLACE and COMBINE), and that non-overwriteable ones are not removed (REPLACE).

2.9 Justification of the Underlying Mechanism

In the following, we shall give reasons why both the capability and access control mechanisms are used in our model and to achieve the goals stated in section 1.2.

2.9.1 Why Are System Constraints Needed and Stored in Access Control Lists?

It is clear that the system constraints (access window and obligations) allow access control policies to be defined in terms of time and events that occur in the system, and thus provide for more dynamic and richer expressions. The separation of system constraints from capabilities allows the introduction of the concept of authority (over both objects and users; see section 2.7.1) in our model, but at the same time delegates some level of control to the owners of capabilities as well. It is obvious that this separation means it is inconvenient and inefficient to have access windows and obligations stored within a capability and thus requires a separate and more centralized database to store them. The separate storage of system constraints from the capabilities also allows to define these system constraints before accesses are permitted. We feel the access control list is the natural place to store the system constraints. This approach is compared to some other models below.

The concept of keys and locks, as used by Ekanadham and Bernstein [Ekanadham 79], does not support the philosophy of our model based on windows and obligations. The philosophy behind keys and locks is to have a capability with unlimited rights and then restrict the rights by adding locks, and with it conditions. Our philosophy is to limit access at the start and add rights later. Furthermore, by directly coupling the capabilities and the conditions, their model does not allow system constraints to be created independently from the capabilities; thus the concept of authority as exists in our model cannot be supported.

In MULTISAFE [Hartson 84], an implementation of the predicate-based model, all access rights are stored together in one relation with an element in every entry that contains either TRUE (no condition), FALSE (no access) or a condition identifier, which can be used to obtain the condition associated with the access right from another relation. Two things did not fit into our concept, namely the storage of all access rights in one spot and the storage of the access rights away from the associated conditions. We believe that both these solutions are not well suited for use in a distributed environment.

2.9.2 Why are Capabilities Useful?

When all the system constraints are stored in the access control lists, why do we still need capabilities?

The capabilities provide the following functionalities in our model:

- they allow a certain amount of distribution of authority, i.e., a subject has partial control of an object when it owns capabilities to the object, as it can pass its access rights to others,
- as mentioned earlier, the two-level approach (capabilities and access control lists) allows the definition of system constraints independently of giving a subject access to an object (e.g. using initial access control lists, editing of the access control lists). The subjects still need to receive the capability to be able to access the object. This feature works especially well in conjunction with the group concept, as system constraints can be imposed on groups (without giving the individual members access to the object),
- an initial access control list does not need to provide an entry for each subject, as new entries are generated only when the access right is passed. Therefore the addition of a new subject to the system does not require any change in an access control list until it receives some access rights. Additionally, no specific subject has to deal with the addition of a new subject unless it interacts with it, e.g., wants to pass some access right to it,
- our rules of merging system constraints allows access with enhanced rights by passing the capability to a program which has more access rights than the invoker,

- the dropping of a capability can be used as a sanction when an obligation is violated. This allows to deprive a subject from accessing an object until some other subject passes it the access right again.

Some but not all of the functionalities above could be expressed by other means or even somehow be integrated into the access control lists, but others cannot. Capabilities provide a powerful and simple mechanism to implement all the functionalities described above.

2.10 Summary

We have proposed a new access control model which allows the addition of pre-conditions and post-condition to access rights. These conditions are stated in terms of time and events. The pre-conditions are defined by access windows, i.e., periods of time during which the access rights are valid. The post-conditions are associated with a window during which they are valid, and specifies that some events must (or must not) occur during a specific period of time, otherwise a prespecified penalty action will be taken. These features are supported by an underlying mechanism which combines access control lists and capabilities. The model is completed by a simple group mechanism.

3 Examples

In this chapter we will give a demonstration of the expressive power of our model by several examples. We will show that our model can be used to express some of the existing models and solve some special problems.

3.1 Expressing other Security Models

3.1.1 Clark / Wilson Model

To demonstrate that the integrity model by Clark and Wilson [Clark 87] can be expressed by our model, we show that their enforcement rules can be specified in terms of our model.

The direct manipulation of objects by users can be prevented by providing an entry in the access control list only for programs, and with no access window for the group ALL.

To ensure that a user can invoke a program only with an allowed set of parameters, we can define access windows on objects as follows: an object can only be accessed by a process if a user invokes the corresponding program and then passes the access rights in the correct order to the process. This can be done by having the 'from_event' of a single access window specify the proper sequence of events, i.e., the invocation of the program and the passing of the parameters, that has to occur before access to the object is granted. A single access window is created for every allowed use, the 'base_time' for the access window is specified to be relative, single access windows non-copiable, and the merge mode of the access window RETAIN.

The static separation of duty is achieved by splitting the users into two categories: users who can create 'pure' objects and users who can create programs. The users in a category have the same user-authority-set but the user-authority-sets of the two categories are made disjoint. The authorities of the programs have to ensure that no member of the authorities of the objects may execute a program.

3.1.2 Access Control List

A conventional access control list for an object is obtained by setting the merge modes for access windows to RETAIN for all entries and making all single access windows non-copiable. The user-authority-set of all users includes the user himself.

We further need to make the capabilities freely available. This can be done by creating a capability server which possesses all the capabilities in the system and is willing to pass on to any subject requesting them. To guarantee that the capability server gets the capabilities of every newly created object, an obligation is attached to the 'create object' operation which requires the creator to pass the capabilities to the capability server.

3.1.3 Pure Capability Scheme

A pure capability scheme can be emulated very easily by supporting only one kind of initial access control list with entries for the group ALL and all access rights with an access window that is always open, i.e., an access window consisting of one single access window ranging from the smallest point in time expressible to the largest point in time expressible. The merge mode for all these entries is set to RETAIN. Setting up the access control list this way guarantees that no access is prohibited by the access control list. So access depends only on the possession of the capability which will be initially given to the creator of the object.

3.1.4 Multilevel Security

To achieve multilevel security, we start from an access control list. All the subjects belong (directly or indirectly) to a group representing specific clearance levels. There are entries in the access control list for the groups which are allowed to execute an access right. We have to define for all the rights an entry with an empty access window for the group ALL with the merge mode set to RETAIN. Further we set the access window merge mode of all the groups to RETAIN.

Users may only create objects using initial access control lists reflecting a classification equal to or higher than the subject's clearance. The user-authority-sets of the users ensure that subjects with a certain clearance level may only use appropriate initial access control lists.

3.1.5 Chinese Wall Security Policy

The Chinese Wall security policy [Brewer 89] can also be expressed by our model. This security policy requires that users who have access to one set of data may not have access to another set of data which belongs to the same class as this may cause a conflict of interest. We will call such data sets mutually exclusive.

Obligations like 'from_time', 'from_event', and 'to_time' of any single access window do not have any direct influence on mutual exclusion and therefore need not be treated differently.

The 'to_event' of any single access window for object o has to contain ‹SELF, acc, x› for every mutually exclusive object x except object o itself, where 'acc' stands for any type of access to x that would forbid the access to a mutually exclusive object later.

The 'base_time' has to be absolute and must be set to the earliest time expressible. If the 'base_time' is relative then the aging of information is considered, i.e., access to an object is only allowed if the subject has not had access to a mutually exclusive object recently.

A problem with this solution is that when another object is added to the set of mutually exclusive objects, all the access control list of the already existing objects have to be updated to include an entry in the 'to_events' for the new object as well.

3.1.6 Object Based Separation of Duty

Object based separation of duty [Nash 90] guarantees that a subject is authorized to perform only one transaction (within a given set of transactions) on a specific data item, and the subject has not performed any other transaction within the set on the data item.

The fact that only specific transactions can be performed on an object can be enforced the same way as in the Clark/Wilson model (see section 3.1.1) except that the base_time has to be absolute. To ensure that the subject has not yet performed another transaction on that data item the 'to_events' of the single access windows have to be amended such that the windows close if the the same user has already run another transaction of that object.

3.2 Expressing Special Cases

3.2.1 Exclusion of a Subject

A subject can be prevented from accessing an object with a certain right simply by setting the associated access window to an empty set and the merge mode for the access window to RETAIN, i.e., the access windows can not be changed.

3.2.2 Forcing a Subject to Access an Object through a Program

The subject is given a capability to access the object but is not assigned any access window. The program, however, is given a valid access window to the object with the merge mode set to RETAIN (or COMBINE), but no capability. To access the object, the subject has to invoke the program and pass the capability to the process created by the invocation. The process then accesses the object on behalf of the users. The program and thus the process must have the merge mode for the access window set to RETAIN or COMBINE.

3.2.3 Program Invocation Passing Minimal Access Rights

As a subject may not wish to pass too much rights to a process it has created, it may be useful to limit the rights passed to a process. Examples of how this can be achieved are given below:

- add the obligation that the program must drop the capability after using the access rights for a certain number of times (the sanction could include dropping the capability),

- terminate the access window at the end of process execution or after using the access right a certain number of times,
- terminate the access window with the first passing of the access right by the process.

3.2.4 n-Person Rules

n-person rules allow accesses to an object only if a number of subjects agree to or allow the access. Events are a very natural way to express this condition as the approval of an access by a subject can be modeled as an event, e.g., the execution of a dummy program.

An n-person rule can now be defined by a single access window whose 'from_event' consists of all the permutations (if the order of approval does not matter) of the approvals by the n subjects. The single access window can be terminated in any desired way, e.g., the access of the object. The 'base_time' is likely to be relative.

3.2.5 Immediate Revocation of Capabilities

To implement immediate revocation of capabilities we can make use of the ability of restricting access windows when passing access rights. Thus, different from the pure capability system discussed in section 3.1.3, the merge mode for the access rights is set to COMBINE, and an access window in the initial access control list is defined only for the CREATOR.

To revoke an access right, we simply amend or create the 'to_event' set of all single access windows of the access right by an event describing the access to a dummy object. This object is created with an initial access control list which only gives the access to its creator. So whenever the capability is to be revoked, it is necessary only for the revoker to access this object. Accessing this object automatically closes the access window and invalidates the access rights passed by the revoker to other subjects, either directly or indirectly.

4 Adapting the Model for a Distributed Environment

In this chapter, we discuss how the model behaves in a distributed environment. The first problems we deal with are network partitioning and communication delay. We then look at the ordering of events, as events can only be ordered partially in a distributed environment without introducing nondeterminism. Finally the fact that total clock synchronization is impossible is discussed.

4.1 Network Partitioning and Communication Delay

Two of the major problems for distributed systems are network partitioning and communication delay. A distributed system should still be able to function properly under these circumstances. We discuss how our model behaves under such circumstances and how these problems can be addressed.

A serious, but inherent problem of discretionary access control (the type of control our model addresses) in a distributed environment is that changes in access rights (adding or revoking rights) and imposing penalties cannot be done across network partitions (e.g. one cannot change an access control list that is in another partition). Furthermore, even without network partitioning these actions across the network suffer from communication delay. Since this is an inherent problem, we can only demand a best effort to minimize these delays.

Access control enforcement has to function correctly in a distributed environment, i.e., it must not permit unauthorized access even in the presence of network partitioning and communication delay . It should also make decisions quickly and so avoid unnecessary delays.

4.1.1 Passing of Access Rights

The passing of access rights is more complex in our model than in other models, as it includes both capability and access control list, and is visualized in Figure 4.1:

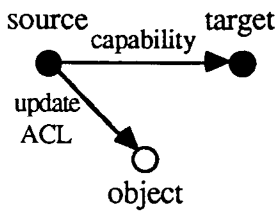

Figure 4.1: Passing of Access Rights

In the worst case the source subject, the target subject and the object are located on different machines. The following three basic cases of network partitioning can occur.

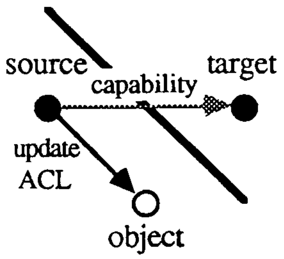

Figure 4.2: Case 1: Separation of Target Subject

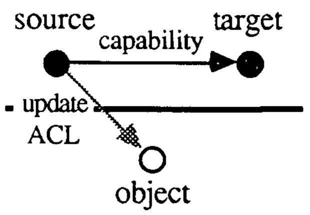

Figure 4.3: Case 2: Separation of Object

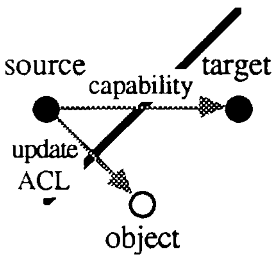

Figure 4.4: Case 3: Separation of Source Subject

In Figures 4.1 through 4.4, the black arrows indicate communication and the grey arrows indicate intended communication which can not take place due to network partitioning (marked by thick black lines).

Case 1: The access control list can be updated but the capability cannot be passed. Our proposal is to allow the access control list to be updated now, but the system should not automatically pass the capability when the partitions remerge. This is because it is possible for the target subject to drop the capability before the merge and then it will receive the capability again. The source subject can, however, pass the capability again when the partitions remerge. Handling passing of access rights in this way still allows passing access rights to groups (not programs) as no capabilities are involved in that case. Also if case 1 changes to case 3 without the partitions first merging, the target subject can still access the object properly if it has received or will receive the capability in some other way. Of course the simplest and cleanest strategy is not to update the access control list when the capability cannot be passed. However, this is less flexible and more restrictive than the proposed scheme.

Case 2: The capability could be passed but the access control list not updated. This could lead to security violation when the access window merge mode for the target is REPLACE (or the obligation merge mode is REPLACE or COMBINE) and the scenario changes from case 2 to case 3 without reaching the normal state in between to allow update of the access control list. In such case, a subject could receive the capability and use it together with an access window it received earlier which is less restrictive or has fewer obligations. Thus, the passing of the capability should not be permitted in this case.

Case 3: Any attempt of the source subject to pass an access right to the target subject will fail as the source subject is separated from the target subject and the object.

Passing access rights to groups requires an up-to-date group tree (specifically, the subtree with the group to which access right is to be passed as root) to be available, including the log of group changes. This is to allow the entries of all group members to be updated properly. If an up-to-date group tree is not available, the passing of the access right has to fail. Passing to the group ALL is an exception as every subject automatically belongs to it, and thus no group tree is needed.

4.1.2 Access Control

The problems caused by network partitioning and communication delays in the case of access checking are minor compared to those of passing access rights. This is because single access windows and obligation validity windows need only be checked when access to the object is requested. The problem is what to do with events which may influence the validity of a window (single access window or obligation validity window) and which occur during the check. To solve this problem the enforcement mechanism may proceed as follows when checking access rights:

1. Check 'from_time' and 'to_time' of all single access windows, and mark all those windows whose 'from_time' but not 'to_time' has passed for further processing. If none is marked, the access fails.

   Now check all 'to_events' of the marked single access windows. If 'to_event' has occurred then unmark the corresponding single access windows. If no marked single access window is left, then the access fails.
2. Check obligation validity windows. Mark those which are not closed, i.e., 'to_time' has not passed and 'to_event' not occurred. If no obligation validity windows is marked, then go to step 4.
3. Check 'from_time' and 'from_event' of marked obligation validity windows. If the window has started (or might start in the very near future, i.e., within this access check[6]) then trigger the obligation. However the blocking of a subject due to an assumed violation of this obligation (see section 4.1.3.1) does not affect this access.
4. Check 'to_time' and 'to_event' of marked single access windows. If neither of them has come to past for any marked single access window then unmark all single access windows and obligations, allow access and confirm assumption for triggering obligations.

   If the assumptions that the access occurred within the obligation validity window cannot be confirmed or access is not granted, then obligations and their consequences have to be removed (see section 4.1.3.2)

The amount of time needed to check the validity of a window can be reduced by cleaning up the access control lists (see appendix D) or by caching information obtained in previous checks (see section 4.1.4).

6 This is possible to estimate if 'from_time' is specified.

If a request for information to a remote site fails (normally indicating network partitioning) the worst case assumptions should to be made. In other words, we will assume the access window is now closed thus denying further access, or the conditions to trigger an obligation is satisfied. We shall discuss later what to do in the case when an obligation is triggered based on a wrong assumption.

When matching those events with description events, group memberships have to be considered also. If the whole group tree is not available (due to network partitioning) the worst case will be assumed (except for the group ALL which consists of all subjects). Thus, for 'from_events' in single access windows, subjects are not considered to belong to a group specified in a description event if it cannot be positively determined. Similarly, for the 'to_events' of single access windows, subjects are considered to belong to a group as described in a description event unless the contrary can be confirmed. For obligation validity windows the opposite assumptions are used.

4.1.3 Obligation Enforcement

The main problem of using events in obligations is that for enforcement purposes, theoretically we need to know instantaneously when an event occurs. Contrary to access checking, for obligation enforcement, checks if an event has occurred have to be made continuously and cannot be done only on request, i.e., we need to provide a mechanism in the sites where relevant events occur to inform the sites which need this information. This can be done by subscribing to this information or by any protocol which allows fast and reliable delivery of the event information, such as Await described in [Reed 79], instead of Read which can be used for access checking. If we receive information that an event has occurred, then we know for sure that it has happened. However the absence of information about an event cannot be used to conclude that the event did not occur, as the absence of information can be caused by communication delay or network partitioning. What can be done is that as soon as one part of the obligation has occurred, we can explicitly ask if the other parts have occurred; e.g. if the 'deadline event' has occurred it can be directly checked if the obligation was violated.

4.1.3.1 Deciding on When to Execute Which Penalty Action

The dilemma of what to do with penalty actions when it is believed an obligation could have been violated can be solved using the properties of the penalty actions (urgency). Urgent actions should not be delayed unnecessarily as otherwise the restriction of some access rights could be delayed.

Passing an access right can be delayed until the information of whether the obligation has been violated arrives. This is because the intention of 'passing an access right' is to have subjects share their access privileges with others, and not to restrict other subjects' access rights (the latter can be achieved by defining single access windows appropriately). Since a delay does not violate our intentions we can consider passing an access right as non-urgent.

Executing a program is the most difficult case to analyze as the purpose of the execution is not known. However, only two operations by a process could be critical - deleting an object and passing an access right. We have argued above that passing an access right can be delayed. Deleting an object by a process is only of concern if the object was not created by the process itself. Since processes are more logically created to access and manipulate objects rather than to just carry out the penalty action of deleting an object which can be done directly using the normal delete object primitive, we will assume (and demand) that processes are not used this way. Therefore we will consider executing a program as non-urgent and it can therefore be delayed.

Dropping a capability is considered urgent and can be done by temporarily invalidating it. This may cause some denial of service. Once the assumption that the obligation has been violated is validated, the dropping can be made permanent. Otherwise the capability can be made valid again. In making the dropping permanent, we should first check to make sure that the subject did not receive the capability again after the obligation was violated.

Logging out can be emulated by suspending any action by this user until it is known whether the assumption is correct. When this is known, the appropriate action can be taken, i.e., either logging the user out or lifting the suspension.

Aborting a process can be dealt with in a similar way to logging out. The execution of the process is simply suspended and will be continued or aborted as soon as it is known whether the obligation has been violated.

The actual deletion of an object can also be postponed until the fate of the obligation is known for sure. In the mean time the object must be made inaccessible to all subjects.

We distinguish between the urgent penalty actions (logging out, aborting a transaction, dropping a capability, and deleting an object) and non-urgent ones (executing a transaction and passing an access right). All actions taken based on these assumptions are treated like normal actions and form part of the history. Furthermore, the executing subjects must have the corresponding access rights and be in possession of the capability, otherwise the penalty action is neglected. In addition, processes cannot terminate and users not removed from the system if there are penalty actions they have to perform.

4.1.3.2 Handling Obligations

To make our analysis of obligations in a distributed environment easier to understand we shall do it in two steps. First we look at obligations without considering 'startevents' and 'deadline events'. These are considered in the second step.

Note that an obligation may contain several obligation elements of which only one has to be fulfilled for an obligation to be fulfilled. However for simplicity, the analysis will assume a single obligation element (to-do or not-to-do obligation elements). In either case the conservative (or pessimistic) view is used.

- to-do: the assumption that the obligation element has not been fulfilled is taken, i.e., unless receiving contrary information it is assumed that the events did not occur when the deadline is past.
- not-to-do: the violation of not-to-do obligation elements has to be checked continuously because as soon as the events matching the formal events occur the obligation is violated. Again the assumption that the obligation element has not been fulfilled is taken. If by the time the deadline arrives, we still do not know for sure if the obligation was violated, explicit checks can be made if those events have occurred.

When all elements of an obligation (and therefore the obligation) are violated the new sanction obligations are immediately imposed, as they do no immediate harm until they are violated. If it is later realized that this assumption was wrong, the following remedy can be used:

If the obligation (or any sanction obligations triggered due to assumed violations of those obligations) is still triggered, i.e., neither violated nor fulfilled, it is removed.

When enforcing sanction obligations, it is necessary to consider events that have occurred after the violation of the original obligation which caused the sanction obligation, i.e., the 'trigger time' of the sanction obligation is taken to be the deadline of the obligation if the latter contains to-do-obligation elements, otherwise it is taken to be the time the last not-to-do obligation element is violated.

When to impose penalty actions is a serious problem. If they are not imposed immediately we have the same problem as that in the obligation validity window, i.e., the penalty actions may be imposed too late. Therefore in case of doubt, it is assumed the obligations are violated and penalty actions are imposed. There is of course a possibility that the assumption was wrong and therefore the penalty unjustified. As shown earlier, this problem can be handled based on the properties of the different penalty actions, i.e., impose urgent penalty actions as a precautionary measure and wait on non-urgent penalty actions until we know for sure.

We now look at the influence of 'startevent' and 'deadline event' defining the obligation window and discuss how it can be managed.

As occurrences of 'startevents' cannot be detected instantaneously, the decision of whether an obligation is fulfilled or violated within the obligation window cannot be immediately ascertained when using 'startevents'.

If it is not known for sure whether the 'startevents' have occurred, it has to be assumed that they have occurred for not-to-do obligation elements, and not for to-do obligation elements. Based on these assumptions the urgent penalty actions are imposed temporarily and the obligations that are part of the sanctions triggered. If this assumption proves to be wrong, then sanctions that were temporarily imposed are revoked and the evaluation of the obligation continues. The revocation of sanctions invoked as a precaution is not really necessary since in some cases it is still not certain if a violation has occurred. However, revocation would reduce the number of blocked subjects and objects.

The late detection of 'deadline events' can lead to the problem that violations or fulfillments (for not-to-do obligation elements) of obligations are realized too late. As for the 'startevent', we face the possibility of imposing a penalty action too late.

As it may take some time to know if 'deadline events' have already occurred, again we take the conservative approach of assuming that they have occurred for to-do obligation elements and have not occurred for not-to-do obligation elements, probably at the same time the obligation is triggered.

4.1.3.3 Groups

When information about an event is received it may be necessary to check the group tree to determine if the event matches a description event using groups. This can be done by requesting the information directly from the group tree (and the associated log). If that information is not available for any reason, then the action to take depends on whether a to-do or not-to-do obligation element is involved. For to-do obligation elements, subjects are not assumed to belong to a group, while for not-to-do obligation elements, they are assumed to belong to a group until we know to the contrary, i.e., a violation of the obligation element is assumed in the event of incomplete information.

For obligation windows, conditional assumptions on whether an event influencing the 'startevent' or the 'deadline event' have occurred are made depending on the type of obligation element. As always the conservative approach is adopted. For to-do obligation elements, in case of uncertainty the obligation window is taken to be as small as possible, while for not-to-do obligation elements, the obligation window is taken to be as large as possible. So when looking at 'startevent', subjects are assumed to belong to a group used in a description event for not-to-do obligation elements, and not belong to a group for to-do obligation elements until determined otherwise. For the 'deadline event', the opposite assumptions are made.

No assumption has to be made for the group ALL as every subject belongs to it.

4.1.3.4 Delayed Imposing of Penalty Actions

Once it is decided that a certain penalty action has to be executed it may still take some time before it is actually executed. This may be due to communication delay, network partitioning (if the action has to be executed on a different site), general system overhead or a combination of the above factors. These delays are inherent and cannot be eliminated. However, the delays may be reduced by executing these actions and any involved communication at a higher priority (if possible) and by optimizing the location of information (see section 4.1.4).

4.1.3.5 Deadlock

Deadlocks are possible when subjects (or objects) are blocked (due to temporary urgent penalty actions). They are less of a problem for 'startevents' as they are for 'deadline events'.

For 'startevents', blocking may result in the temporary unavailability of one or more subjects or objects needed for their fulfillment. As long as the deadline is guaranteed to occur, all to-do obligation elements are violated while all not-to-do obligation elements are fulfilled when the deadline arrives because there is no obligation window (since the deadline has occurred without the 'startevent' happening).

The problem gets more serious when 'deadline events' are involved. We first assume that only 'deadline events', but neither a 'deadline period' nor a 'deadline time', are defined.

The problem occurs when blocked subjects and objects are needed to decide on the 'deadline event' and the fulfillment (or violation) of the obligations. As the deadline cannot occur, the only way to make sure that the subject gets unblocked is a definitive fulfillment or violation of the obligation. Such a decision is however impossible if the subjects or objects needed for the decision are blocked as well. Circular dependency of such blocking results in deadlocks. If a deadlock is caused by a circular dependency within one obligation only we call it an internal deadlock, otherwise it is known as an external deadlock.

4.1.3.6 Deadlock Prevention

There are different ways to prevent deadlocks from occurring:

- one way is the use of 'deadline period' or 'deadline time', instead of or in addition to 'deadline events'. These two parameters limit the time a subject can be blocked except for the inherent network delays mentioned earlier.

  For this method to work in practice, it is necessary to prevent users from defining 'deadline periods' or 'deadline times' that are too long, thus allowing virtual deadlocks especially when using additional obligations or when a user is included in his own user-authority-set. In these cases the user can change the access control and maliciously create a deadlock. Thus, some additional rules on obligations (and sanction obligations) are needed to prevent misuse. These rules may include:

- An upper limit is imposed on 'deadline periods' or the time between the definition of the obligation and the 'deadline time' if there are urgent penalty actions. These limits also apply to the 'deadline periods' and 'deadline times' of sanction obligations.
- A subject may not be part of the 'deadline event' description and at the same time be a subject that would be penalized to avoid some internal deadlocks. (There could still be internal deadlocks when a group the penalized subject belongs to is part of the 'deadline event' description).
- Additional or initial obligations would only be allowed if they cannot create a deadlock with other (sanction) obligations in the same access control list entry. Normal passing of access rights can still create deadlock situations if the merge mode for obligations is COMBINE.

- a second solution is based on the argument that penalty actions cannot be imposed without delay anyway (see section 4.1.3.4); so why go through all the trouble of trying to impose penalties as soon as possible? In addition most obligations are only influenced by the local environment so that delays in detecting violations would be relatively small. When we consider all penalty actions to be non-urgent then we do not have to block any subject (object) and thus no deadlock can occur. In this scheme all penalty actions are imposed only when it is sure that the obligation is violated (whether sanction obligations could still be triggered when a violation is assumed would have to be dealt with separately; not imposing them would reduce the overhead of enforcement, but once the original obligation is known to be violated there will be a price paid as they are triggered too late and the enforcement has to start retroactively). As the access is still checked by access windows, unauthorized access is prevented that way. However changes in access rights due to violated obligations would be delayed.

This solution reduces the importance of the role of obligations but would avoid major problems (in respect to security policies and implementation).

4.1.3.7 Continuous Enforcement

While dealing with all the problems mentioned above, the enforcement of the obligations must be continued. Recall that only one obligation element has to be fulfilled for an obligation to be fulfilled and both the 'startevent' and 'deadline event' may consist of a set of description events of which again only one has to be fulfilled to open and close the obligation window respectively.

4.1.3.8 Overview

The enforcement of an obligation can be viewed as a state machine. The state transitions (Figure 4.5) occur as follows:

$$\begin{tikzcd}
 & \text{triggered} \arrow[dl] \arrow[d, leftrightarrow] & \\
\text{fulfilled} & \text{uncertain} \arrow[l] \arrow[r] & \text{violated}
\end{tikzcd}$$

Figure 4 5: States of Obligation Enforcement

- 'triggered' → 'fulfilled': The state of an obligation moves from 'triggered' to 'fulfilled' if the obligation consists of to-do obligation elements only, and there is no 'deadline event' specified, and the information that one of the obligation elements is fulfilled arrives before the deadline expires. Additionally there must be no 'startevent' specified or it has already occurred.
- 'triggered' → 'uncertain': If the conditions mentioned above are not fulfilled:

  case 1: If the obligation contains not-to-do obligation elements or 'deadline event' is specified, then the transition occurs immediately after the obligation is triggered. Otherwise the transition occurs when the deadline expires without confirmation that anyone of the obligation elements is fulfilled.

case 2: If the obligation contains not-to-do obligation elements or 'deadline event' is specified, and 'startevent' is specified but it is not known for sure that is has occurred, the the transition takes place as soon as the obligation is triggered.

- 'uncertain' $\rightarrow$ 'triggered': If the obligation switched from 'triggered' to 'uncertain' due to case 2 mentioned above and it is learned that the 'startevent' has not yet occurred so that no decision can be made, the obligation can switch back to the 'triggered' state. This transition is not essential but avoids unnecessary blocking.
- 'uncertain' $\rightarrow$ 'fulfilled': If an obligation is in the state 'uncertain' and it is learned that at least one obligation element is fulfilled.
- 'uncertain' $\rightarrow$ 'violated': If an obligation is in the state 'uncertain' and it is learned that all obligation elements are violated.

The actions that should be taken in the different enforcement states are described in Table 4.1 for a scenario without regarding deadlock prevention.

fulfilled	uncertain	triggered	violated
- remove obligations that have been triggered (directly or indirectly) due to this obligation - remove all temporarily imposed urgent penalty actions caused by this obligation (both directly and indirectly triggered ones) - remove the fulfilled obligation	- trigger any sanction obligations - temporarily impose urgent penalty actions - continue enforcement and try to come to a conclusive decision	- remove obligations that have been triggered (directly or indirectly) due to this obligation - remove all temporarily imposed urgent penalty actions caused by this obligation (both directly and indirectly triggered ones)	- make urgent penalty actions permanent - impose any non-urgent penalty actions - remove the violated obligation

Table 4.1: Measures to Be Taken in the Different Enforcement States

4.1.4 Optimization of Availability

Many problems with network partitioning and communication delay can be avoided using a smart design and implementation of the system. The chief means to achieve this are replication and location of information needed to make the security decisions.

The information that can be replicated are:

- history

  Events and their orderings can without much overhead be stored at the following locations (with some additional communication cost) to facilitate faster access:

  - where the access occurs,
  - where the acting subjects resides,
  - when passing an access right the information can be replicated and stored at the location of the source subject, the location of the target subject and the location of the object.

Replication of parts of the history can reduce uncertainties, as events which might be used later at another site are replicated over there as they occur and not when they are needed. A problem that has to be addressed is the consistency of the replicated information. All the sites where an event is replicated should receive the information at the same time. Otherwise, it is necessary to check the other sites if the information required is not found in a particular location. This is especially important during network partitionings.

Single sites can also improve availability by storing (caching) replies from previous requests.

- group tree

  As the group tree may also be distributed, the changes have to be made in an orderly way (e.g. using transactions and pessimistic syntactic strategies) to maintain consistency of the replicated parts [Davidson 85]. A pessimistic approach is needed as protection and integrity may not even temporarily be endangered by making decisions based on inconsistent information. This means that some changes in the group structure cannot be made immediately, but this is again a problem with discretionary access control.

  The log that is associated with the group membership tree for group changes can also be replicated. As for the history, the replicated versions do not have to be up-to-date as long as the enforcement mechanism is aware of this. In this case, the other sites will have to be checked.

Choosing the location of information wisely can optimize the performance of the enforcement mechanism in the presence of network partitioning. The placement of the following information can be optimized:

- history

  as shown above the history can be easily replicated at various locations. This enables a faster and more efficient search for events as one can search for this information based on different keys, i.e., acting subject, involved object and target subject.

- group tree

  the locations of different parts of the group tree will probably follow some organizational structure, e.g. the subtree for an organization is stored within the subnet for this specific organizational unit. Furthermore, the root of the tree or the next few nodes up the subtree could be replicated so that group membership decisions can be made more quickly.
- triggered obligations

  while no replication is possible for triggered obligations (since we want to enforce it at one place only) their placement can still be optimized. The triggered obligations can be located at the following places:
  - where the access causing the triggering of the obligation occurs. If the obligation triggers sanction obligations, they should be placed at the same location as well, especially when the sanction obligations are triggered based on assumptions only,
  - where the next event of the 'startevent', or that of the 'deadline event' is expected to occur,
  - where the next event to fulfill (or violate) an obligation element is expected to occur.

  It is also possible to migrate a triggered obligation so that it is placed at a location most suited for its enforcement. However, the communication overhead resulting from the migration must also be considered.

  The best placement for triggered obligations cannot be determined exactly as the fulfillment of different description events (e.g. 'startevent', 'deadline event' and obligation elements) which may occur on different sites must be reported to one location.

4.1.5 Clock Synchronization

Due to network partitions clocks cannot be synchronized globally. Therefore clocks have to be accurate enough to keep within a certain time limit despite being separated for some time (this can also be supported by synchronizing clocks within a partition when applicable) (see also section 4.3).

4.2 Event Ordering

In distributed systems events can only be ordered partially, and not totally, without introducing some arbitrary notion (e.g. machine number) [Lamport 78], so we will settle for a partial ordering of events. Partial ordering of events considers the possible influence of one event on another one. Note that concurrent events cannot causally affect each other.

4.2.1 How to Determine Partial Ordering

Lamport shows how to partially order events among processes [Lamport 78]. We adapt these findings to events of interest to our model.

Timestamps are not used in the ordering of the events as we are only interested in the mutual influence of events and not in a more complete ordering. The timestamps are only used to determine if an event occurred before or after a certain point in time (e.g. 'base_time').

A problem is that the events in our security model may consist of several events on different sites and spread over a period of time. We show now how a partial ordering can be achieved under this situation and the possible influence of one event on another one.

The following assumptions are made:

- one processor per site
- communication maintains message ordering between sites (FIFO)
- objects are only accessible by one subject at a time. Accesses to the same object that overlap each other in time are assumed to occur concurrently [Reed 79] and are not considered in this analysis

4.2.1.1 Access (read, write, delete, execute)

Accessing an object is made up of three actions. First there is the request for accessing an object by a subject (req). Then comes the access on a possibly different site (acc), and finally there is the reply to the requesting subject (rep) which has to occur on the same site as the request (not considering migration of the subject). The event that is important the second step, the actual access, which are emphasized in the diagrams below. s1, s2, s3 and o1, o2, o3 indicate subjects and objects respectively.

Graphically this looks as follows:

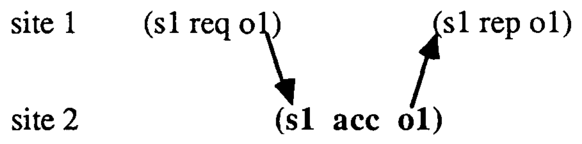

Figure 4.6: Normal Access

The arrows in the figure indicate communication between two sites. The horizontal axis is the time axis.

All the examples below are given for the three sites case with the one-site and two-site cases being simpler versions of it.

Case 1

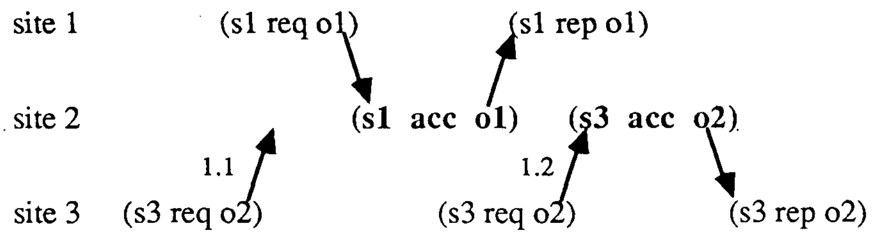

Figure 4 7: Case 1: Accesses on Same Site

In case 1, o1 and o2 can be identical, but do not have to.

case 1.1: (s1 acc o1) → (s3 acc o2) ('→' means precedes)

the access of o2 by s3 in this case is delayed for some reason (e.g. scheduling)

case 1.2: (s1 acc o1) → (s3 acc o2)

Case 2

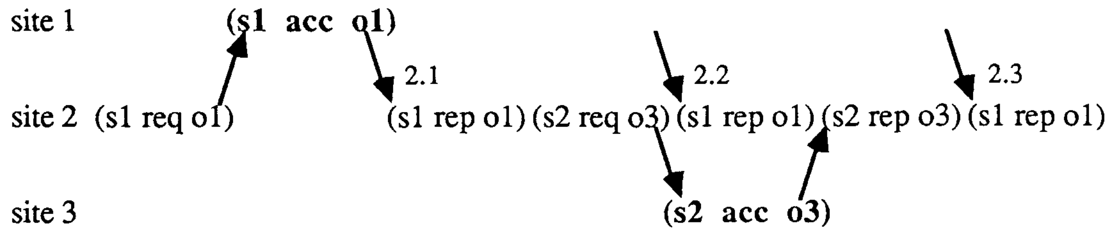

Figure 4.8: Case 2: Requests on Same Site

in case 2, s1 and s2 can be identical, but do not have to.

case 2.1: (s1 acc o1) → (s2 acc o3)

case 2.2: no direct ordering between (s1 acc o1) and (s2 acc o3)

case 2.3: no direct ordering between (s1 acc o1) and (s2 acc o3)

Case 3

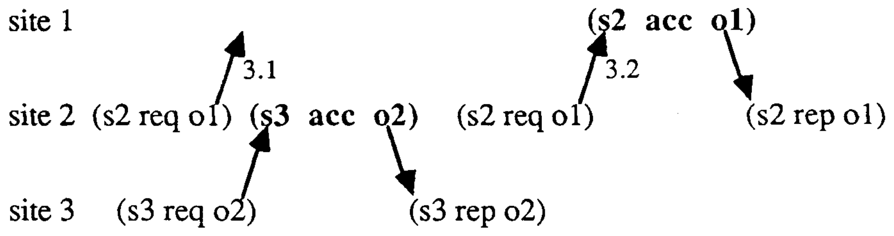

Figure 4.9: Case 3: First Access on Same Site as Second Request

case 3.1: no ordering between (s3 acc o2) and (s2 acc o1)

case 3.2: (s3 acc o2) → (s2 acc o1)

It is easy too see that an access a occurs before an access b, if and only if at least one of the following conditions is fulfilled:

- access a and access b occur at the same site and a occurs before b (cases 1.1 and 1.2)
- access a and the request of accessing b occur at the same site and a occurs before the request for accessing b (case 3.2)
- the requests of accessing a and b occur at the same site and the reply to the request for a arrives before the request to b (case 2.1)

4.2.1.2 Single Site Events

login, logout, start, stop, abort and discard are single site actions and therefore do not add any complexity. We can consider them as single events since no intersite communications occur.

4.2.1.3 Passing of Access Rights

The passing of access rights is split up into several security events. This is to make sure that all relevant actions are logged and that it will work even in the presence of network partitions or passing to a group other than a program, where the passing of capability is not necessary. In Figure 4.10, 's1 pass o1 s2' means that subject s1 initiates the passing of access right to object o1 to subject s2, 'acl-update' is the event of updating the appropriate access control list while 's2 receives o2' is the event of subject s2 receiving the capability to object o2.

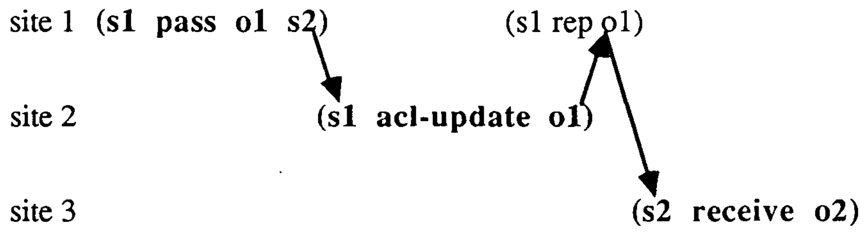

Figure 4.10: Passing of Access Rights

We have the following orderings within passing an access right:

(s1 pass o1 s2) $\rightarrow$ (s1 acl-update o1)

(s1 pass o1 s2) $\rightarrow$ (s2 receive o2)

(s1 acl-update o1) $\rightarrow$ (s2 receive o2)

The impact of parallel passing of access rights or passing of access rights and accesses on event ordering can be analyzed in the same way showed above for accesses.

4.2.1.4 Relation between execute Access and start

An execute access (on a program) always precedes the start of the created process (see Figure 4.11). This requires that the reply to the access is sent before the creation of the process. In Figure

4.11, 's1 execute o1' means that subject s1 executes program o1 and 'p1 starts' is the event of the associated process p1 starting.

$$\begin{tikzcd}
\text{site 1} & \text{(s1 req o1)} \arrow[dr] & & \text{(s1 rep o1)} & \textbf{(p1 starts)} \\
\text{site 2} & & \textbf{(s1 execute o1)} \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Figure 4.11: Relation between Execute Access and Start

4.2.2 General Ordering Rules

We now can state the general rules for partially ordering the security-relevant events:

- if two security-relevant events occur on the same site they are ordered on the time of occurrence
- if one security-relevant event (a) causes another one (b) then they are ordered (a → b)
- if request to and reply from a security-relevant event (a) occur before the request to another security-relevant event (b) at the same site then they are ordered (a → b)
- if a security-relevant event (a) occurs before the request for another security-relevant event (b) at the same site then those events are ordered (a → b)

The partial ordering is also irreflexive and transitive (see [Lamport 78]).

With these general rules it is possible to figure out the ordering between security-relevant events.

4.3 Clock Synchronization

In distributed systems, each site maintains its own clock which is responsible for that site. The clock is used to decide on access windows and other parameters specified in real time. This requires that the clocks on different sites are synchronized.

4.3.1 Physical Clock Synchronization

Though global clock synchronization may be difficult to achieve due to communication delays and network partitioning, we only have to ensure that all clocks only differ by a small amount (to within a second) to ensure that single access windows, 'base_times' and time-differences are viewed (almost) identically from every site.

Clocks do not only have to be synchronized among each other (internal time synchronization) but also have to be close to some external time signal (e.g. UTC) (external time synchronization). Clocks that are synchronized externally are also synchronized internally.

The first condition for clocks to support our model in a distributed environment is:

Physical Clock Condition: all clocks have to be within a small range (less than a second) of an external time signal and thus of each other.

The reason why some small discrepancy among the clocks on different sites can be tolerated is that the only purpose of the physical clock condition is to synchronize unrelated events such that when we look for events that occurred after a 'base_time' we have about the same absolute starting point. This does not have to be totally accurate as events that are of interest are casually related and therefore can be ordered without relying solely on the physical clocks.

4.3.2 Logical Clock Synchronization

Even if the clocks are physically synchronized according to the condition above we can have the problem that partially ordered events do not have compatible timing. In Figure 4.12, for example event b occurs before event c but has a later timestamp as the clocks on site 1 and site 2 are not perfectly synchronized. This is to be avoided, because otherwise an event could be considered relevant for fulfilling a description event despite having occurred after one that is not considered (in Figure 4.12, b is considered but not c if one is considering all events that occur after 11:00 based on the timestamp event though b actually occurred before c). Thus another condition which we shall call logical clock condition is required.

Logical Clock Condition: if an event precedes another one then it must have a smaller timestamp than the event it precedes.

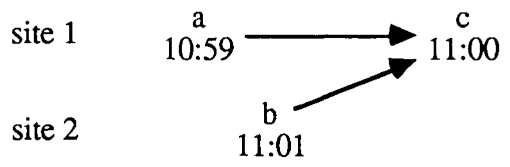

Figure 4.12: Violation of Logical Clock Condition

One way to satisfy the logical clock condition (with physical clocks) would be to synchronize the clocks such that the minimal communication delay (including processing overhead) is larger than the maximal differences of the clocks. This however is very difficult to achieve, as in deterministic algorithms the accuracy of synchronization depends mainly on the uncertainty about the time passing between generating and receiving a synchronization message and on the time between synchronizations [Gusella 89].

Although there exist several deterministic algorithms to synchronize clocks (e.g. [Gusella 89], [Lamport 84]), the inaccuracy caused by the factors mentioned above makes synchronization of the desired accuracy virtually impossible. There also exists a probabilistic approach to synchronize clocks which provides a higher precision than the deterministic ones (good enough to fulfill the logical clock condition) but with some probability of failing to synchronize the clocks [Cristian 89] (which would not necessarily mean the loss of synchronization). Further improvements such as estimating the drift of a hardware clock and so build self-adjusting clocks can also lead to better results.

4.3.3 How to Fulfill Both the Physical and Logical Clock Conditions

If a clock synchronization that is always better than the minimal communication delay cannot be guaranteed, other ways to fulfill the logical clock condition have to be looked at including turning the clock of the 'receiving' site forward [Lamport 78] (without violating the physical clock condition) and correcting this by later slowing down the clock, or delaying the synchronizing event until the logical clock condition is fulfilled [Jefferson 85]. The use of these measures can be reduced by having the clocks synchronized as accurately as possible.

We saw above that we need clocks that satisfy both the physical and logical clock conditions.

One way would be to have both a logical and a physical clock. The physical clocks can be synchronized among themselves. The logical clocks on the one hand ensure the logical clock condition is satisfied, and on the other hand must stay within a limit of the local physical clock despite the adjustment due to the conditions imposed on logical clocks. This can be achieved by turning the clocks forward if the physical clock condition would not be violated by this, or by blocking operations otherwise.

In this way we can ensure all the clocks do not deviate from the external time signal more than the synchronization error of the physical clocks plus the maximal allowed temporary deviation of the logical clock.

4.4 Summary

In this chapter we showed how our model can be adapted to work properly in a distributed environment. In the first part - the enforcement mechanism - both access control and obligation enforcement are examined for their behavior in a distributed environment . The ways to deal with network partitioning and communication delay are pointed out and some restrictions are formulated to make the model work smoothly in a distributed environment. Then it is shown how the security-relevant events can be partially ordered without giving up any degree of security. And finally the physical and logical clock conditions, which have to be satisfied at all times for the model to work properly, are stated and some ways to achieve this are suggested.

5 Conclusions

In this thesis we have proposed a new access control model for general purpose distributed systems. The model is based on time and events and allows the specification of security policies that are highly flexible and versatile.

The major building blocks of the model are access windows (pre-conditions) and obligations (post-conditions) which are associated with every ‹subject, right›-pair. These building blocks are then combined with a flexible underlying mechanism (a combination of capabilities and access control lists) such that none of the flexibility provided by the access windows and obligations was lost. A simple group mechanism, which reduces the amount of information that has to be stored and allows the specification of more general security policies, completes the model.

The versatility of the model has been demonstrated by several examples showing how some existing models may be emulated and how some special problems can be solved in terms of the new model. Finally the behavior of the model in a distributed environment was discussed to examine the problems that may arise in such an environment and how they can be overcome.

While this thesis may constitute a step in the direction of using conditions which are based on time and events in the area of computer security, more research remains to be done. Further research may include the following topics:

- implementation of a prototype

  Until now this model exists only on paper, but to check the feasibility of this model a prototype has to be implemented. The major problems in doing this are the relative complexity of the model, which makes the correct implementation difficult. Also research into efficiency and the creation of an easy-to-use user interface such that the definition of the system constraints is not overly complicated are needed.

- exploration of other possible security policies that can be expressed by our model

  We have not exhaustively explored the wide variety of security policies that can be expressed by our model. Once a prototype is implemented the whole spectrum of possible security policies can be used and new types of security policies can be tested.

- exploration of alternatives to the group mechanism and/or the underlying mechanism

  While developing our model we aimed at supporting the flexibility of the system constraints as much as possible. This led to our choice of a combination of access control lists and capabilities as an underlying mechanism and the provision of a simple group mechanism. Other approaches to support the system constraints with different emphases on different aspects (e.g. ease of implementation and use vs. maximal flexibility and complexity) should be studied.

References

[Bell 76] Bell, D., LaPadula, L.: *Secure Computer System: Unified Exposition and Multics Interpretation*, ESD-TR-75-306, MITRE Corporation, Bedford, Massachusetts, March 1976.

[Brewer 89] Brewer, D., Nash, M.: *The Chinese Wall Security Policy*, Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, May 1 - 3 1989, pp. 206 -214.

[Clark 87] Clark, D., Wilson, D.: *A Comparison of Commercial and Military Computer Security Policies*, Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, April 27 - 29 1987, pp. 184 -194.

[Cristian 89] Cristian, F.: *A Probabilistic Approach to Distributed Clock Synchronization*, Proceedings of the 9th International Conference on Distributed Computing Systems, 1989, p. 288 - 296.

[Davidson 85] Davidson, S., Garcia-Molina H., Skeen, D.: *Consistency in Partitioned Networks*, ACM Computing Survey, Vol. 17, No. 3, September 1985, pp. 341 - 370.

[Denning 79] Denning, D., Denning, P.: *Data Security*, ACM Computing Surveys, Vol. 11, No. 3, September 1979, pp. 227 - 249.

[Dennis 66] Dennis, J., Van Horn, E.: *Programming Semantics for Multiprogrammed Computations*, Communications of the ACM, Vol. 9, No.3, March 1966, pp. 143 - 155.

[DOD 83] Department of Defense Computer Security Center: *Department of Defense Trusted Computer System Evaluation Criteria*, Fort Meade, Maryland, August 1983.

[Ekanadham 79] Ekanadham, K., Bernstein, A.: *Conditional Capabilities*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979, pp. 458 - 464.

[Gligor 79] Gligor, V: *Review and Revocation of Access Privileges Distributed Through Capabilities*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, November 1979, pp. 575 - 586.

[Goguen 84] Goguen, J., Meseguer, J.: *Unwinding and Inference Control*, Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, April 29 - May 2 1984, pp. 75 - 85.

[Gusella 89] Gusella, R., Zatti, S.: *The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD*, IEEE Transactions on Software Engineering, Vol. 15, No. 7, July 1989, pp.847 - 853.

[Hartson 84]	Hartson, R.: *Implementation of Predicate-Based Protection in MULTISAFE*, Software - Practice and Experience, Vol. 14, No. 3, March 1984, pp. 207 - 234.
[Jacob 89]	Jacob, J.: *On The Derivation of Secure Components*, Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 1 - May 3 1989, pp. 242 - 247.
[Jefferson 85]	Jefferson, D.: *Virtual Time*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, pp. 404 - 425.
[Johnson 88]	Johnson, D., Thayer, J.: *Stating Security Requirements with Tolerable Sets*, ACM Transactions on Computer Systems, Vol. 6, No. 3, August 1988, pp. 284 - 295.
[Karger 84]	Karger, P., Herbert, A.: *An Augmented Capability Architecture to Support Lattice Security and Traceability of Access*, Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, April 29 - May 2 1984, pp. 2 - 12
[Lamport 78]	Lamport, L.: *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 559 - 565.
[Lamport 84]	Lamport, L., Melliar-Smith, P.: *Byzantine Clock Synchronization*, in Proceedings of 3rd ACM Annual Symposium on Principles of Distributed Computing, Vancouver, B.C., Aug. 1984, pp. 68 - 74.
[Lampson 69]	Lampson, B.: *Dynamic Protection Structures*, Proceedings of the AFIPS 1969 Fall Joint Computer Conference, AFIPS Press, Montvale, New Jersey, Vol. 35, pp. 27 - 38.
[Lampson 74]	Lampson, B.: *Protection*, Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971, pp. 437 - 443, reprinted in Operating Systems Review, Vol. 8., No. 1, January 1974, pp. 18 -24.
[Landwehr 84]	Landwehr, C., Heitmeyer, C., McLean, J.: *A Security Model for Military Message Systems*, ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 198 - 222.
[Lipner 82]	Lipner, S.: *Non-Discretionary Controls for Commercial Applications*, Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, April 26 - 28 1982, pp. 2 - 10.
[Minsky 84]	Minsky, N.: *Selective and Locally Controlled Transport of Privileges*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, pp. 573 - 602.

[Minsky 85]	Minsky, N., Lockman, A.: *Extending Authorization by Adding Obligations to Permissions*, Technical Report, Department of Computer Science, Laboratory for Computer Science Research, LCSR-TR-67, Rutgers University, New Brunswick, New Jersey, January 1985.
[Nash 90]	Nash, M., Poland, K.: *Some Conundrums Concerning Separation of Duty*, Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, May 7 - 9 1990, pp. 201 - 207.
[Organick 72]	Organick, E.: *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Massachusetts, 1972.
[Reed 79]	Reed, D., Kanodia, R.: *Synchronization with Eventcounts and Sequencers*, Communications of the ACM, Vol. 22, No. 2, February 1979, pp. 115 - 123.
[Wilkes 68]	Wilkes, M.: *Time-Sharing Computer Systems*, American Elsevier Publishing Company, Inc., New York, 1968.

A Formal Model Description

In this appendix the main parts of the model introduced in chapter 2 are more formally specified assuming a non-distributed environment.

The components of the model are defined in the first part, and the major procedures in the second one. Data types and member of enumeration types will be in SMALL CAPITALS and comments are printed in *italics* and enclosed by '(*' and '*)'.

A.1 Model Components

We first define the different data types and then define the system using these data types. The name of the elements of the data types are later used for the definition of the major procedures.

A.1.1 Data Types

The data types needed to state the model formally are presented. In these definitions the data types BOOLEAN, TIME and TIMEPERIOD are basic data types. '?' is a special value indicating undefined.

R: {READ, WRITE, DELETE, EXECUTE} *(* set of rights *)*

A: {LOGIN, LOGOUT, EXECUTE, START, END, ABORT, READ, WRITE, DELETE, PASS, UPDATE-ACL, RECEIVE, DISCARD} *(* set of actions *)*

MERGE MODES: {RETAIN, REPLACE, COMBINE}

C: object: O[7] *(* capability *)*
right: R

REAL EVENT:
- s: S_U[8] $\cup S_{PR}$
- action: A
- parameter: ordered set of $(S \cup O \cup C)$
- ts: TIME *(* timestamp *)*

DESCRIPTION EVENT:
- s: $S \cup$ {SELF, OTHER}
- action: A
- param: ordered set of $(S \cup$ {SELF, OTHER} $\cup O \cup$ {ANY} $\cup C)$

FORMAL EVENT: set of (ordered set of DESCRIPTION EVENT)

conditions:
- the ordered sets may not be empty
- an empty set, ∅, indicates that the formal event is undefined

[7] O is defined in the section A.1.2.

[8] S, S_U, S_{PG}, S_{PR}, and S_G are defined in the section A.1.2.

SINGLE ACCESS WINDOW:	from_time:	TIME
	from_event:	FORMAL EVENT
	to_event:	FORMAL EVENT
	to_time:	TIME
	copiable:	BOOLEAN
	overwriteable:	BOOLEAN
	base_time:	TIME
	rel_base_time:	TIMEPERIOD

conditions:
- $(\text{from\_event} \neq \emptyset) \vee (\text{from\_time} \neq ?)$
- $(\text{to\_event} \neq \emptyset) \vee (\text{to\_time} \neq ?)$
- $((\text{from\_event} \neq \emptyset) \wedge (\text{to\_event} \neq \emptyset)) \vee (\text{base\_time} \neq ?)$
- $((\text{base\_time} = ?) \wedge (\text{rel\_base\_time} \neq ?)) \vee$
  $((\text{base\_time} \neq ?) \wedge (\text{rel\_base\_time} = ?))$

ACCESS WINDOW:	access_window:	set of SINGLE ACCESS WINDOW
	merge_mode:	MERGE MODES

OBLIGATION ELEMENT:	requirement_type:	{TO_DO, NOT_TO_DO}
	basic_requirement:	ordered set of DESCRIPTION EVENT

OBLIGATION:	obligation_element:	set of OBLIGATION ELEMENT
	startevent:	FORMAL EVENT
	deadline_event:	FORMAL EVENT
	deadline_time:	TIME
	deadline_period:	TIMEPERIOD
	sanction:	SANCTION
	obligation_validity_window:	SINGLE ACCESS WINDOW

condition: - $(\text{deadline\_event} \neq \emptyset) \vee (\text{deadline\_time} \neq ?) \vee (\text{deadline\_period} \neq ?)$

SANCTION: new_obligation: set of OBLIGATION

penalty: set of (subj: $S_{PR} \cup S_U \cup \{\text{SELF, CREATOR, PROCESS}\}$

action: A - {LOGIN, START, END, READ, WRITE, UPDATE-ACL, RECEIVE}

param: ordered set of $(S \cup O \cup C)$)

(remark: - obligation validity windows meaningless in new_obligation *)*

TRIGGERED OBL ENTRY:	activation_time:	TIME
	activator:	$S_U \cup S_{PR}$
	startevent:	FORMAL EVENT

deadline_event: FORMAL EVENT
deadline_time: TIME
requirement: set of OBLIGATION ELEMENT
sanction: SANCTION

conditions: - PROCESS and CREATOR not allowed in DESCRIPTION EVENTs

ACCESS CONTROL LIST: set of (subj: S
right: R
aw: ACCESS WINDOW
ob: obligation: set of OBLIGATION
merge_mode: MERGE_MODE)

conditions: - CREATOR not allowed in DESCRIPTION EVENTs
- PROCESS in DESCRIPTION EVENTs only allowed if subj of list element $\in S_{PG}$

INITIAL ACCESS CONTROL LIST: set of (subj: $S \cup \{CREATOR\}$
right: R
aw: ACCESS WINDOW
ob: obligation: set of OBLIGATION
merge_mode: MERGE_MODE)

conditions: - at most one entry per subject and right
- PROCESS in DESCRIPTION EVENTs only allowed if subj of list element $\in S_{PG}$
- in initial access control lists penalty actions of any sanction may only include the following subjects:
  - SELF
  - CREATOR in entry for CREATOR
  - subjects if entry for the same subject
  - PROCESS if entry for $s \in S_{PG}$

A.1.2 Security State

The security state of a system is described by the quintuple ‹S, O, history, triggered, actualtime› and the sets/functions c, group, acl, iacl, user-auth-set, and auth.

Quintuple

S: set of subjects, with S_U: set of users,
S_{PG}: set of programs,
S_{PR}: set of processes,
S_G: set of groups, including ALL containing all other subjects

(due to the group function (see below) subjects remain members of S, but do not remain member of any subset if they are removed *)*

O: set of objects, with $O \supseteq S_{PG}$

history: set of REAL EVENT

triggered: set of TRIGGERED OBL ENTRY

actualtime: TIME *(* running time *)*

Sets/Functions

c: $S_U \cup S_{PG} \cup S_{PR} \rightarrow$ set of C *(* capabilities a subject owns *)*

group: $S \times \text{TIME} \rightarrow S_G \cup S_{PG} \cup \{\text{NIL}, \emptyset\}$

(defines group tree at a given time; NIL is reserved to designate the group of removed subjects; removed subjects remain member of S for formal reasons (group) but do not belong to any of the subsets (S_U, S_{PG}, S_{PR}, or S_G). $\emptyset$ is reserved for the non existing group membership of ALL *)*

$\forall\, t \in \text{TIME}: \text{group}(\text{ALL}, t) = \emptyset$ *(* ALL is the root of the group tree *)*

$\forall\, s \in S, \forall\, t \in \text{TIME}, n \rightarrow \infty: \text{group}^n(s, t) = \emptyset$ *(* no cycles allowed *)*

$\forall\, s \in S_U, \forall\, t \in \text{TIME}: \text{group}(s, t) \in S_G$ *(* users must be leaves of the tree *)*

$\forall\, s \in S_G, \forall\, t \in \text{TIME}: \text{group}(s, t) \in S_G$ *(* only groups atop groups *)*

$\forall\, s \in S_{PG}, \forall\, t \in \text{TIME}: \text{group}(s, t) \in S_G$ *(* only groups atop programs *)*

$\forall\, s \in S_{PR}, \forall\, t \in \text{TIME}: \text{group}(s, t) \in S_G \cup S_{PG}$ *(* only groups and programs atop processes *)*

acl: O $\rightarrow$ ACCESS CONTROL LIST *(* access control list belonging to an object *)*

iacl: $S_U \rightarrow$ set of INITIAL ACCESS CONTROL LIST *(* initial access control list, that is used when a subject creates a new object *)*

user-auth-set: $S_U \rightarrow$ set of S_U, $(\forall\, x \in S_U, \text{auth}(x) \neq \emptyset)$ *(* user-authority-set of a subject . A non-empty set of users is returned *)*

auth: $O \rightarrow$ set of S_U, $(\forall\, x \in O, \text{auth}(x) \neq \emptyset)$ *(* authority of an object. A non-empty set of users is returned *)*

A.2 User and System Actions

In this section we show how users or system operations can change the security state of the system.

A.2.1 Basic Functions

```
(* is-description-event checks if real event re corresponds to the description event de for a user s *)
is-description-event(s: $S_U \cup S_{PR}$, re: REAL EVENT, de: DESCRIPTION EVENT, t: TIME): BOOLEAN
  IF $s \in S_{PG}$ THEN
    replace all occurrences of PROCESS in de by s
  return  (re.action = de.action) $\wedge$
          ((de.s = re.s) $\vee$ (de.s $\in$ group*[9](re.s, t) $\vee$ ((de.s = SELF) $\wedge$ (s = re.s)) $\vee$
          ((de = OTHER) $\wedge$ (s $\neq$ re.s)) $\wedge$
          ($\forall$ i (i := 1 ... |de.param|[10] ):  (de.param$_i$[11] = re.parameter$_i$) $\vee$
                                                      ((de.param$_i$ = ANY) $\wedge$ (re.param$_i$ $\in$ O))$\vee$
                                                      (de.param$_i$ $\in$ group*(re.s, t)) $\wedge$
                                                      ((de.param$_i$ = SELF) $\wedge$ (re.param$_i$ = s)) $\vee$
                                                      ((de.param$_i$ = OTHER) $\wedge$ (re.param$_i$ $\neq$ s)))

(* op-saw checks if a single access window saw is open for a subject s *)
op-saw(s: S, saw: SINGLE ACCESS WINDOW ): BOOLEAN
  IF base_time $\neq$ ? THEN                                   (* 'base_time 'is absolute *)
    base := saw.base_time
  ELSE                                                         (* 'base_time' is relative *)
    base := actualtime + rel_base_time
  return  ((actualtime $\geq$ saw.from_time) $\vee$ (saw.from_time = ?)) $\wedge$
          ((actualtime $\leq$ saw.to_time) $\vee$ (saw.to_time = ?)) $\wedge$
          ((saw.from_event = $\emptyset$) $\vee$
          ($\exists$ {$x_1$, ..., $x_n$}:  $\forall$ i (i := 1 ... n): ($x_i \in$ history $\wedge$ ($x_i$.ts > base)) $\wedge$
                               $\forall$ i (i := 1 ... n-1): ($x_i$.ts < $x_{i+1}$.ts) $\wedge$
```

9 * defines the set of all nodes in the tree above the parameter of the function.

10 |x| returns the number of elements in set x.

11 The index i describes the ith element in the set.

```
                    $\exists$ y $\in$ saw.from_event:
                      ($|y| = n$) $\wedge$ $\forall$ i (i := 1 ... n): is-description-event(s, $x_i$, $y_i$, $x_i$.ts))) $\wedge$
     ((saw.to_event = $\varnothing$) $\vee$
     ($\exists$ {$x_1$, ..., $x_n$}: $\forall$ i (i := 1 ... n): ($x_i \in$ history $\wedge$ ($x_i$.ts > base)) $\wedge$
                    $\forall$ i (i := 1 ... n-1): ($x_i$.ts < $x_{i+1}$.ts) $\wedge$
                    $\exists$ y $\in$ saw.to_event:
                      ($|y| = n$) $\wedge$ $\forall$ i (i := 1... n): is-description-event(s, $x_i$, $y_i$, $x_i$.ts)))
```

A.2.2 Object Creation

When a subject s ∈ S creates an object o with a requested initial access control list (riacl), the following changes to the security state occur.

```
(* add new object to the appropriate sets *)
$O := O \cup \{o\}$
IF o is a program THEN
  $S := S \cup \{o\}$
  $S_{PG} := S_{PG} \cup \{o\}$
  $\forall$ t > actualtime:
    group(o, t) := ALL                        (* assign group membership from now on *)
(* if a user creates an object then his user-authority-set becomes the authority of the object,
otherwise the user-authority-set of the user at the head of the invocation chain of the process is
used; the user-authority-set also provides a set of initial access control lists from which one is taken
*)
IF $s \in S_U$ THEN
  auth(o) := user-auth-set(s)
  IF riacl $\in$ iacl(s) THEN
    acl(o) := riacl
  ELSE
    abort object creation                     (* includes removal of object from all sets *)
ELSE (* $s \in S_{PR}$ *)
  auth(o) := user-auth-set(invoker(s))        (* invoker returns user at head of invocation chain *)
  IF riacl $\in$ iacl(invoker(s)) THEN
    acl(o) := riacl
  ELSE
    abort object creation                     (* includes removal of object from all sets *)
```

```
(* give capabilities of object to its creator *)
$\forall$ r $\in$ {READ, WRITE, DELETE}
  c(s) := c(s) $\cup$ {‹o, r›}
IF o is a program THEN
  c(s) := c(s) $\cup$ {‹o, EXECUTE›}
(* remove CREATOR entry *)
FOR $\forall$ r $\in$ R
  IF entry for s and CREATOR and r in riacl THEN
    sptr := x $\in$ acl(o): (x.r = r) $\wedge$ (x.subj = s)        (* pointer to creating subjects entry *)
    pass(s, CREATOR, o, r, $\varnothing$, sptr.aw)      (* see section A.2.3 for passing access right *)
    acl(o) := acl(o) - entry described by sptr               (* remove subject's entry *)
replace all occurrences of CREATOR in acl(o) by s
```

A.2.3 Passing of Access Right

The passing of an access right r $\in$ R on object o $\in$ O from source $\in$ ($S_U \cup S_{PR}$) to target $\in$ S with additional obligations ao (see section 2.8.4) and the restricted access window raw (replace PROCESS by source; also see appendix B for details) is defined as follows:

```
pass( source: $S_U \cup S_{PR}$; target: S, o: O; r: R; ao: set of OBLIGATION;
      raw: set of SINGLE ACCESS WINDOW)
  (* check if source is in possession of the capability belonging to the access right *)
  IF ‹o, r› $\in$ c(source) THEN
    (* add capability to capability set of target if it is a program, process or user and is not already
    in possession of it *)
    IF (target $\in S_U \cup S_{PR} \cup S_{PG}$) $\wedge$ (‹o, r› $\notin$ c(target)) THEN
      c(target) := c(target) $\cup$ {‹o, r›}
      update-list := {target}                          (* list of entries to be updated *)
    IF target $\in$ ($S_G \cup S_{PG}$) THEN          (* also update access control lists of group members *)
      $\forall$ gm: target $\in$ group*(gm)
        IF $\exists$ y $\in$ acl(o): (y.r = r) $\wedge$ (y.subj = gm) THEN     (* only update if entry exists *)
          update-list := update-list $\cup$ {gm}
```

```
∀ to-update ∈ update-list
  (* updateptr is used as a pointer for the entry in the access control list of o which is valid for
  the subjects whose access right r is updated; it has the value NIL if there is no entry *)
  updateptr := NIL
  subject := to-update
  WHILE (updateptr = NIL) ∧ (subject ≠ ∅) DO
    updateptr := x ∈ acl(o): (x.r = r) ∧ (x.subj = subject)
    subject := group(subject, actualtime)
  (* check if both access window and obligations of entry to be updated are retained and if
  there is an entry for the target subject at all; merging can be skipped in this cases *)
  IF  (updateptr ≠ NIL) ∧ (updateptr.aw.merge_mode ≠ RETAIN) ∧
      (updateptr.ob.merge_mode ≠ RETAIN) THEN
    (* set system constraints for target to empty in access control list if no entry exists *)
    IF subject ≠ to-update THEN                                   (* only target can enter here *)
      (new_entry is pointer to new entry)
      new_entry.aw access_window := updateptr.aw.access
      new_entry.aw.merge_mode := updateptr.aw.merge_mode
      new_entry.ob.obligation := updateptr.ob.obligation
      new_entry.ob.merge_mode := updateptr.ob.merge_mode
      new := TRUE                                                 (* new entry prepared *)
    ELSE
      new_entry := updateptr
      new := FALSE
    (* existing access windows have to be removed from entry to be updated, if merge mode is
    REPLACE and they are overwriteable *)
    IF new_entry.aw.merge_mode = REPLACE THEN
      ∀ x ∈ new_entry.aw.access_window
        IF x.overwriteable THEN
          new_entry.aw.access_window := new_entry.aw.access_window - {x}
```

```
(* in combine and replace mode, all copiable access windows have to be copied from
source to the entry to be updated *)
IF new_entry.aw.merge_mode $\in$ {REPLACE, COMBINE} THEN
  $\forall$ x $\in$ raw
    IF x.copiable THEN
      new_entry.aw.access_window := new_entry.aw.access_window $\cup$ {x}
(* existing obligations have to be removed from entry to updated, if merge mode is
REPLACE and they are overwriteable *)
IF new_entry.ob.merge_mode = REPLACE THEN
  $\forall$ x $\in$ new_entry.ob.obligation
    IF x.obligation_validity_window.overwriteable THEN
      new_entry.ob.obligation := new_entry.ob.obligation - {x}
(* in COMBINE and REPLACE mode, all copiable obligations have to be copied from source
to entry to be updated *)
IF new_entry.ob.merge_mode $\in$ {REPLACE, COMBINE} THEN
  (* sptr is used as a pointer for source's entry for right r in the access control list of o; it
  has the value $\emptyset$ if there is no entry *)
  sptr := NIL
  subject := source
  WHILE (sptr = NIL) $\wedge$ (subject $\neq \emptyset$) DO
    sptr := x $\in$ acl(o): (x.r = r) $\wedge$ (x.subj = source)
    subject := group(subject, actualtime)
  $\forall$ x $\in$ sptr.ob.obligation
    IF x.obligation_validity_window.copiable THEN
      replace process by source where appropriate
      new_entry.ob.obligation := new_entry.ob.obligation $\cup$ {x}
  $\forall$ x $\in$ ao
    new_entry.ob.obligation = new_entry.ob.obligation $\cup$ {x}
(* add new entry to access control list if different from entry previously valid *)
IF new $\wedge$ ((new_entry.aw $\neq$ updateptr.aw) $\vee$ (new_entry.ob $\neq$ updateptr.ob)) THEN
  acl(o) := acl(o) $\cup$ {<s, r, new_entry.aw, new_entry.ob>}
```

A.2.4 Accessing an Object

We first determine the entry in the access control list which is valid for the requesting subject (s) and the type of request (r).

```
sptr := NIL   (* sptr will point to the access control list entry valid for subject s and right r *)
subject := s
WHILE (sptr = NIL) $\wedge$ (subject $\neq \emptyset$) DO
  sptr := x $\in$ acl(o): (x.r = r) $\wedge$ (x.subj = subject)
  subject := group(subject, actualtime)
```

Then we have to replace every occurrence of PROCESS the the subject if the requesting subject is a process.

A subject $s \in S$ can access an object $o \in O$ with right $r \in R$, if and only if

$(\langle o, r\rangle \in c(s)) \wedge (\exists\, z \in$ sptr.aw.access_window: op-saw(s, z)

This has the following effects on triggered:

```
$\forall$ x $\in$ sptr.ob.obligation
  IF op-saw(s, x.obligation_validity_window) THEN
    triggered := triggered $\cup$ {‹actualtime, s, x.startevent, x.deadline_event, min(x.deadline_time,
    actualtime + x.deadline_period), x.obligation_element, x.sanction›}
```

There is also the special case that an authority of an object can always delete the object, i.e., $s \in S_U$ can access $o \in O$ with right $r \in R$ if:

$(s \in \text{auth}(o)) \wedge (r = \text{DELETE})$

A.2.5 Execution of a Program

The execution of a program $p \in S_{PG}$ by a subject $s \in \{S_U \cup S_{PR}\}$ proceeds as follows:

```
1. invocation of program, i.e., s must have execution access to p
   history := history $\cup$ {‹s, EXECUTE, {p}, actualtime›}
2. creation of process pr
   S := S $\cup$ {pr}
   $S_{PR}$ := $S_{PR}$ $\cup$ {pr}
   c(pr) := c(p)
   $\forall$ t > actualtime:
     group(pr, t) := p                    (* assign group membership from now on *)
   history := history $\cup$ {‹pr, START, $\emptyset$, actualtime›}
```

```
3. parameter passing (not mandatory)
   history := history $\cup$ {‹s, PASS, {p, c}, actualtime›}
   history := history $\cup$ {‹s, ACL-UPDATE, {p, c.o}, actualtime›}
   history := history $\cup$ {‹p, RECEIVE, {s, c}, actualtime›}
4. parameter return (not mandatory)
   history := history $\cup$ {‹p, PASS, {s, c}, actualtime›}
   history := history $\cup$ {‹p, ACL-UPDATE, {s, c.o}, actualtime›}
   history := history $\cup$ {‹s, RECEIVE, {p, c}, actualtime›}
5. process termination
   history := history $\cup$ {‹pr, END, $\varnothing$, actualtime›}, or
   history := history $\cup$ {‹pr, ABORT, $\varnothing$, actualtime›}
   (* program is only removed from program set but not from subjects because of group
   function *)
   $S_{PR} := S_{PR} - \{pr\}$
   $\forall$ t > actualtime:
     group(pr, t) := NIL      (* subject does not belong to any group anymore*)
```

A.2.5 Obligation Enforcement

```
$\forall$ x $\in$ triggered
  (* check if start of obligation window already occurred; set started accordingly *)
  IF x.startevent = $\varnothing$ THEN   (* no 'startevent'; obligation window started immediately *)
    starttime = x.activation_time
    started := TRUE
  ELSE
    (* se contains all series of events that fulfill the 'startevent' *)
    se:= {{$z_1, \dots, z_n$}:  $\forall$ i (i := 1 ... n): ($z_i \in$ history $\wedge$ ($z_i$.ts > x.activation_time)) $\wedge$
                          $\forall$ i (i := 1 ... n-1): ($z_i$.ts < $z_{i+1}$.ts) $\wedge$
                          $\exists$ y $\in$ x.startevent:
                            (|y| = n) $\wedge$
                            $\forall$ i (i := 1 ... n): is-description-event(x.activator, $z_i$, $y_i$, $z_i$.ts)}
    IF se = $\varnothing$ THEN    (* 'startevent' did not yet occur *)
      started := FALSE
    ELSE
      started := TRUE
    starttime := $\infty$
    $\forall$ e $\in$ se            (* find earliest fulfillment of 'startevent' *)
      starttime := min(last(e).ts, starttime)
(* check if deadline of obligation window already occurred; set finished accordingly *)
IF x.deadline_time $\neq$ ? THEN
  dlt := x.deadline_time
IF x.deadline_event = $\emptyset$ THEN
  finished := actualtime > x.deadline_time
  dlt := x.deadline_time
ELSE
  dl := {{$z_1, \ldots, z_n$}:  $\forall$ i (i := 1 ... n): ($z_i \in$ history $\wedge$ ($z_i$.ts > x.activation_time)) $\wedge$
                                 $\forall$ i (i := 1 ... n-1): ($z_i$.ts < $z_{i+1}$.ts) $\wedge$
                                 $\exists$ y $\in$ x.deadline_event:
                                   (|y| = n) $\wedge$
                                   $\forall$ i (i := 1 ... n): is-description-event(x.activator, $z_i$, $y_i$, $z_i$.ts)}
  IF dl = $\emptyset$ THEN
    finished := (x.deadline_time $\neq$ ?) $\wedge$ (actualtime > x.deadline_time)
  ELSE
    finished := TRUE
    $\forall$ e $\in$ dl
      dlt := min(last(e).ts, dlt)              (* compare with last event of ordered event set *)
(* check for violation or fulfillment of obligations *)
IF $\neg$ started $\wedge$ finished THEN
  IF $\exists$ y $\in$ x.requirement: y.requirement_type = NOT_TO_DO THEN
    TRIGGERED := TRIGGERED - {x}
  ELSE
    violated := TRUE
(* check if obligation is violated or fulfilled *)
IF started THEN
  fulfilled :=
    $\exists$ y $\in$ x.requirement:          (* check if a to-do or a not-to-do obligation element is fulfilled *)
      ((y.requirement_type = TO_DO) $\wedge$
      ($\exists$ {$z_1, \ldots, z_n$}:  $\forall$ i (i := 1 ... n): ($z_i \in$ history $\wedge$ ($z_i$.ts > starttime) $\wedge$ ($z_i$.ts $\leq$ dlt)) $\wedge$
                          $\forall$ i (i := 1 ... n-1): ($z_i$.ts < $z_{i+1}$.ts) $\wedge$
```

```
                        $(|y.basic\_requirement| = n) \wedge$
                        $\forall i \; (i := 1 \ldots n)$:
                          is-description-event(x.activator, $z_i$, y.basic_requirement$_i$, $z_i$.ts)) $\vee$
    ((y.requirement_type = NOT_TO_DO) $\wedge$
    $(\neg\exists \{z_1, \ldots, z_n\}$: $\forall i \; (i := 1 \ldots n)$: $(z_i \in$ history $\wedge$ $(z_i.ts >$ starttime$)$ $\wedge$ $(z_i.ts \leq$ dlt$))$ $\wedge$
                        $\forall i \; (i := 1 \ldots n-1)$: $(z_i.ts < z_{i+1}.ts)$ $\wedge$
                        $(|y.basic\_requirement| = n) \wedge$
                        $\forall i \; (i := 1 \ldots n)$:
                          is-description-event(x.activator, $z_i$, y.basic_requirement$_i$, $z_i$.ts))
  IF fulfilled THEN
    triggered := triggered - {x}        (* remove entry in list of triggered obligations *)
  ELSE
    IF finished THEN
      violated := TRUE
    ELSE
    (* an obligation is violated during obligation window if all obligation elements are violated *)
      violated :=
        $\forall y \in$ x.requirement:
          (* not-to-do obligation element violated *)
          ((y.requirement_type = NOT_TO_DO) $\wedge$
          $(\neg\exists \{z_1, \ldots, z_n\}$: $\forall i \; (i := 1 \ldots n)$: $(z_i \in$ history $\wedge$ $(z_i.ts >$ starttime$)$ $\wedge$ $(z_i.ts \leq$ dlt$))$ $\wedge$
                              $\forall i \; (i := 1 \ldots n-1)$: $(z_i.ts < z_{i+1}.ts)$ $\wedge$
                              $(|y.basic\_requirement| = n) \wedge$
                              $\forall i \; (i := 1 \ldots n)$:
                                is-description-event(x.activator, $z_i$, y.basic_requirement$_i$, $z_i$.ts)) $\vee$
IF violated THEN
  $\forall y \in$ x.sanction.penalty                                        (* execute penalties *)
    IF y.subj = SELF THEN
      y.subj := x.activator
    $\forall i \; (i := 1 \ldots |y.param|)$
      IF y.param$_i$ = SELF THEN
        y.param$_i$ := x.activator
    execute(y)          (* the system has to guarantee that this penalty is executed immediately *)
```

```
$\forall$ y $\in$ x.sanction.new_obligation                    (* impose new obligations *)
  triggered := triggered $\cup$ {‹actualtime, y.activator, y.startevent, y.deadline_event,
                         min(y.deadline_time, actualtime + y.deadline_period),
                         y.obligation_element, y.sanction›}
triggered := triggered - {x}
```

B Ways to Restrict Access Windows

There are three general methods to restrict an access window:

i) remove one or more single access windows from the access window

ii) duplicate a single access window, restrict it and the copy separately[12]

iii) restrict one or more of the single access windows
   - make single access window non-copiable
   - reduce size of timeframe
     - increase or define 'from_time'
     - decrease or define 'to_time'
   - reduce size of event frame
     - define 'from_event'
     - define 'to_event'
     - decrease 'base_time' in case 'from_event' is undefined, i.e., set back the point in time from which the search in the history starts
     - increase 'base_time' in case 'to_event' is undefined, i.e., set forward point in time from which the search in the history starts
     - add one or more additional sets of description events to 'to_event'
     - remove one or more (but not all) sets of description events from 'from_event'
     - add one or more description events to any set of description events in 'from_event'
     - remove one or more (but not all) description events from any set of description events in 'to_event'
     - replace any occurrence of ANY in a description event for 'from_event' by an object or capability according to the requested parameter
     - replace any occurrence of a group in a description event for 'from_event' by a member of the subtree to which the group is root
     - replace any occurrence of an object or capability in a description event for 'to_event' by ANY, depending on the requested parameter (this can also be achieved by adding a new ordered set of description events)

[12] There is no additional access gained by duplicating an access window. However duplication allows a more flexible restriction of single access windows (see point iii); e.g. 9am to 5pm can be split into two more restrictive intervals - 9am to 11am and 2pm to 5pm.

- replace any occurrence of a group or a subject in a description event for 'to_event' by a group further up the group tree (this can also be achieved by adding a new ordered set of description events)

C Restriction of System Constraints Containing SELF or OTHER

The use of SELF and OTHER allows system constraints to be defined with respect to the acting subject. This means that system constraints may be different depending on the subject that is accessing the resource.

We now show how one may prevent passing increased access rights to other subjects.

Single Access Window

The goal here is to make the single access windows at most as wide as they were for the source subject, i.e., we want to make 'from_event' harder to occur and 'to_event' easier to occur. The ways we can achieve this goal have been listed in appendix B.

- 'from_event'
  - SELF: replace every occurrence of SELF with the identifier of the source subject and SELF, i.e., given the description event ‹SELF, ..., ...› in an ordered set of description events, duplicate this ordered set and replace ‹SELF, ..., ...› by ‹source, ..., ...› ‹SELF, ..., ...›, and ‹SELF, ..., ...› ‹source, ..., ...› respectively. This guarantees that neither the source subject nor the target subject (represented by SELF) can provide the necessary event themselves but both of them must contribute, thus making the 'from_event' harder to occur than if only one of them had to contribute.
  - OTHER: we want to make sure that neither the source subject nor the target subject may open the window, as the intersection of OTHER from the point of the source and the target respectively contains all subjects except themselves. To make the 'to_event' harder to occur neither the source subject nor the target subject may fulfill the description event with OTHER. Thus we need to keep the entry ‹OTHER, ..., ...› to prevent the target subject from opening the window, but also have to add to the 'to_event' the entry ‹source, ..., ...› to prevent the source subject from opening the window (it can open the window as it belongs to OTHER, but also closes it at the same time). This solution may be overly restrictive, as once ‹source, ..., ...› occurred the window is unnecessarily closed (denial-of-service). We can however reduce this problem by duplicating the single access window and replacing ‹OTHER, ..., ...› by ‹OTHER, ..., ...› ‹source, ..., ...› in the 'from_event' and ‹source, ..., ...› ‹source, ..., ...› in the 'to_event'. This procedure can be repeated as many times as necessary to reduce the problem of denial-of-service.

- 'to_event'
  - SELF: we want to replace every occurrence of SELF with the source subject or SELF, i.e., given the description event ‹SELF, ..., ...› in an ordered set of description events, duplicate this ordered set and replace ‹SELF, ..., ...› by ‹source, ..., ...› in the duplicate. This way either the source or the target subject can close the window.
  - OTHER: replace any occurrence of OTHER by ALL (the union of OTHER seen from the source subject and seen from the target subject respectively consists of all subjects). Thus the single access window can be closed by any subject defined by OTHER from both the source and the target subject's point of view.

Obligations

Obligations cannot be changed or removed but we can add additional obligations.

- obligation validity window: the obligation validity window of the source subject should completely overlap that of the target subject. The following operations will guarantee this:
  - 'from_event': add an obligation which uses this 'from_event' as 'to_event' and define the 'from_event' according to the 'to_event' for single access windows, i.e., the new obligation with the same obligation elements and sanctions is defined and their composed obligation window starts at least as early as the existing one regardless if the latter one is seen from the source or the target subject.
  - 'to_event': add an obligation which uses this 'to_event' as 'from_event' and define the 'to_event' according to the 'from_event' for single access windows, i.e., the new obligation with the same obligation elements and sanctions is defined and their composed obligation window at least as last as the existing one regardless if the latter one is seen from the source or the target subject.
- obligation window: split obligation elements into groups of to-do and not-to-do elements. Add obligation for each of the two groups and reduce obligation window for to-do group and increase obligation window for not-to-do group.
  - 'startevent': see single access window 'from_event' for to-do part and 'to_event' of single access window for not-to-do part.
  - 'deadline event': see single access window 'to_event' for to-do part and 'from_event' of single access window for not-to-do part.
- obligation element: add obligation equal to the original one but change obligation element according to its type:

- to-do: see 'from_event' of single access window.
- not-to-do: see 'to_event' from single access window.

- sanction: make sure sanction to source subject stay.
  - obligation: add obligation equal to original one but adapt sanction obligations such that also the source subject also gets the sanction obligations imposed on it.
  - penalty: add obligations equal to original one with the exception that SELF in the penalty will be replaced by the source (OTHER is not permitted in the penalty).

D Cleanup

Goals: - Remove entries or parts in entries that do not have any impact on access control but may slow down the system. Thus cleanup is mainly a tuning function.
- Improve the availability and reliability in distributed environments in the presence of network partitions and communication delays.

When and What Can Be Cleaned up

When an object is deleted (the access control list may be used for hints as to which subjects may possess capabilities for the object), and as a background system job,

1) remove capabilities to objects that no longer exist.

When a subject is removed (its capabilities may be used as hints to look for entries in access control lists), and as a background system job,

2) remove access control list entries of subjects that no longer exist.

As background system job,

3) remove access control lists entries which are duplicates of the next group further up in the group tree which has an entry.

As background system job or ss part of access control check,

4) single access windows: remove if
   - 'to_time' has passed
   - 'to_event' has been fulfilled for an ordered set of description event not containing SELF, OTHER, PROCESS or a relative 'base_time',
5) obligation: remove if
   - 'to_time' of obligation validity window has passed
   - 'to_event' of obligation validity window has been fulfilled for an ordered set of description event not containing SELF, OTHER, PROCESS or a relative 'base_time',
6) merge single access windows together (e.g. 9am - 1pm and 11am - 5pm ⇒ 9am - 5pm) (if the overwriteable and copiable attributes of both single access windows are the same),
7) eliminate single access windows that are also expressed by other single access windows (e.g. 11am - 2pm is contained in 9am - 5pm) (furthermore, one can assume that for windows already open, 'from_time' is equal to the actual time) (if the overwriteable and copiable attributes of both single access windows are the same),

8) merge obligations together if their validity window are continuous (see 6) and the obligation elements and sanctions are identical (if the overwriteable and copiable attributes of both obligation validity windows are the same),
9) eliminate obligations that are expressed by other obligations (if the overwriteable and copiable attributes of both obligation validity windows are the same),
10) eliminate entries in access control lists which are identical to entries of the group to which they belong,
11) if 'from_event' is fulfilled then set 'from_event' to undefined and set 'from_time' to the maximum of the actual time and the existing 'from_time' (unless SELF, OTHER, PROCESS or relative 'base_time' is used).

As part of the obligation enforcement,

12) remove obligation elements that have been violated from the list of triggered obligations.