A Semantic Base for Verifying the Security of a Computer System

Saad C. Himmich

Computer Science Report No. TR-92-31
October 5, 1992

A Semantic Base for Verifying the Security of a Computer System

Saad C. Himmich

Computer Science Report No. TR-92-31
October 5, 1992

ABSTRACT

Nearly all the projects to design secure computer systems for processing classified information have had a formal mathematical model for security as part of the top-level definition of the system. Such a model functions as a concise and precise definition of the desired behavior of the security-relevant portions of the system.

This thesis proposes one such model based on the take-grant protection model which describes the access control facilities for shared resources in computer systems. The current state of the system is given by a finite, directed labelled protection graph G(V, E). V is the set of vertices and E the set of edges, where labels include a finite set of rights. First, we show how the extensions made to the take-grant model permit revocation and confinement. Second, a number of results are derived in the form of theorems to be applied to a protection graph. They show how access rights may be transmitted in that graph. It is possible to determine in linear-time in the size of the graph if a given node can obtain rights to another one. An algorithm to do so is presented. Finally, a definition of a "trusted component" in a system is proposed and it is shown how it is possible to characterize that "trust" in our model.

ACKNOWLEDGEMENTS

I wish to express my deep gratitude to my academic advisor, Bill Wulf, for his time, patience and guidance, without which this work would never have been completed. Special thanks go to Anita Jones for taking the time to review my thesis and to help me improve its quality. I would also like to thank the other member of my committee, Andrew Grimshaw, for his cooperation.

I also owe thanks to the members of Wulf's research group for providing ideas and concepts and for their assistance in the preparation of my presentation.

Many friends have made my life in Charlottesville enjoyable. I thank Anu for his moral support and Valerie for her love and comfort.

Finally, I would like to thank my parents and my sister Karima for their constant encouragements to pursue my goals.

ABSTRACT

Nearly all the projects to design secure computer systems for processing classified information have had a formal mathematical model for security as part of the top-level definition of the system. Such a model functions as a concise and precise definition of the desired behavior of the security-relevant portions of the system.

This thesis proposes one such model based on the take-grant protection model which describes the access control facilities for shared resources in computer systems. The current state of the system is given by a finite, directed labelled protection graph G(V, E). V is the set of vertices and E the set of edges, where labels include a finite set of rights. First, it is shown how the extensions made to the take-grant model permit revocation and confinement. Second, a number of results are derived in the form of theorems to be applied to a protection graph. They show how access rights are allowed to be transmitted in that graph. It is possible to determine in linear-time in the size of the graph if a given node can obtain rights to another one. An algorithm to do so is presented. Finally, a definition of a "trusted component" in a system is proposed and it is shown how it is possible to characterize that "trust" in our model.

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 Motivation for research

Because it is critical that the integrity of programs and data in a computer system be maintained, it is important to conduct a thorough study of the protection mechanism used in a system.

To underline the importance of a thorough study of protection systems, observe that flaws in a protection system are not necessarily detected by the common user and one cannot count on users to systematically report such flaws. In fact, the user is not necessarily aware that information was stolen from him and the "thief" will not report the flaw that allowed him to obtain that information. In addition, in some cases, the disclosure of sensitive information can have drastic, irreversible consequences.

This irreparable nature is not common in other contexts. For example, one can expect the user of a network to report flaws in the design of a network protocol. The protocol could then be modified because, for instance, the throughput of the network is very low under heavy load or because an error in the election algorithm allows two nodes in the network to become master nodes. Once the error is detected, modifications can be made to the mechanism without too much damage.

In the case of a protection system however, it is not possible to detect flaws in the same way. To illustrate, let us imagine that one's sole policy is to prevent a given top-secret piece of information from being seen. If there is a flaw in the protection system and this information becomes visible, no matter how effective the protection mechanism is rendered afterwards, the policy has already been violated. It is this irreversibility in the consequences of a flaw in a protection mechanism that makes it very important to be able to design a provably secure system.

Proving that a system is secure is not an easy task. In fact, for arbitrary systems it is undecidable as shown in [HRU76]. For some systems however, the problem of determining whether or not the system is "safe" can be decided. The first step towards proving a system secure is to define the *security policy* that must be enforced (by stating what information is to be protected). Next, the system must be shown to be in a state where that policy is not and will not be violated. Finally, the protection system must correctly implement the model used to specify the policy.

A security policy can be expressed as a set of specifications in much the same way as a program can. These specifications have to be based on a formal definition which makes formal proofs of security possible as argued in [LAN81]. A framework must be developed in which formal specifications are written. This reduces the set of security policies to those that can be expressed in the framework. A properly chosen framework can make the problem of determining whether the state of a system is safe, decidable.

Yet, the framework should be expressive enough to permit the description of security policies of interest. Consider, for example, a framework in which the only policy that can be expressed is the policy that allows all users of a computer system to access all resources in the system. It is trivially decidable whether a state of that system is safe (all states are safe). However, most policies are not captured by the framework, which makes it useless.

An analogous situation is found in the choice of grammars in compiler construction. It is undecidable whether a given grammar is ambiguous. However, if we restrict that grammar to be LR(1), the problem becomes decidable (all LR(1) grammars are unambiguous). It so happens that for most programing languages there exists an LR(1) grammar. On the other hand, we could have chosen to generate languages using right linear grammars which would have also made the problem of ambiguity decidable. The difference is that most programing languages cannot be generated by right linear grammars. Thus LR(1) is an appropriate balance of expressiveness and decidability.

This thesis defines a framework in which to express specifications of a security policy. The framework is based on the *take-grant* model as defined in [SNY81a] for which several results concerning the transfer of rights and of information have already been established. The take-grant model suffers one problem though. It fails to capture policies for which revocation or confinement have to be enforced. Consequently, we propose an extension to the take-grant model to address revocation and confinement and show how some of the results obtained for the take-grant model are affected.

1.2 Background

In this section, we describe the take-grant model [SNY81a]. We will present our extended model in chapter 2.

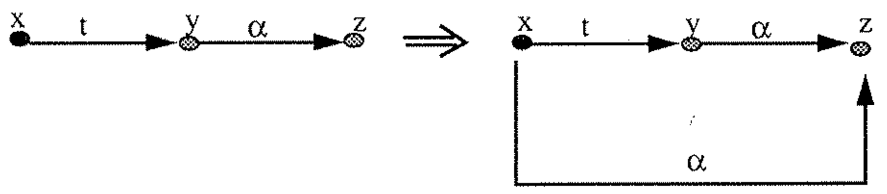
The take-grant model represents the protection state of a system as a directed graph called a *protection graph*. Nodes in the graph can be active (subjects) or passive (objects). Directed edges determine the rights that a node has over another one. They are also called *arcs*. Arcs are labelled with a subset of rights taken from a set of rights R. R includes {t, g} where t and g represent *take* and *grant* rights. The dynamics of the system are described by a set of *rules* that characterize how the graph is allowed to evolve. Because they result in a modification of the graph, these rules are also referred to as *graph rewriting rules*.

Subjects (graphically represented by ●) differ from objects (represented by ○) in that they can modify the graph in a way specified by the rules. Subjects are said to *initiate* rules. When a subject in a graph G initiates a rule, a new graph G' is constructed from G. We say that G' is *derived* from G which we write as G |- G'.

In the description of the graph rewriting rules below, we represent nodes that can either be subjects or objects by ◍ . The following rules are those of the traditional take-grant model and are defined in [SNY81a].

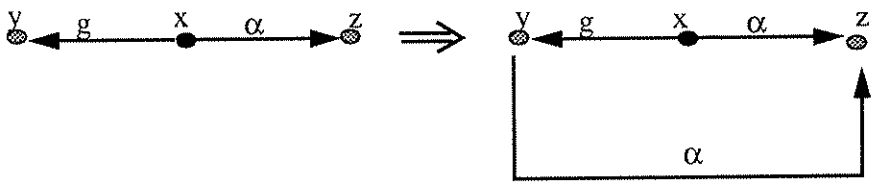
take rule:

Let x, y, z be three distinct vertices in a protection graph and let x be a subject. Let there be an arc from x to y labelled with t rights and an arc from y to z labelled with α rights. If x initiates the take rule, a new arc from x to z labelled with α rights is added to the graph.



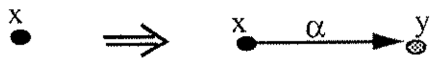
grant rule:

Let x, y, z be three distinct vertices in a protection graph and let x be a subject. Let there be an arc from x to y labelled with g rights and an arc from x to z labelled with α rights. If x initiates the grant rule, a new arc from y to z labelled with α rights is added to the graph.



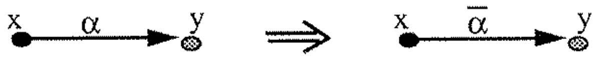
create rule:

Let x be a subject in a protection graph. If x initiates the create rule, a new vertex y is added to the graph and an arc from x to y labelled with α rights is added to the graph.



remove rule:

Let x, y be two distinct vertices in a protection graph and let x be a subject. Let there be an arc from x to y labelled with a set of rights containing α rights. If x initiates the remove rule, the α rights are removed from that set of rights.



The take-grant model has been extensively studied and a number of results established. In particular, a theorem that gives necessary and sufficient conditions for a node in a protection graph to obtain rights to another node has been proven. The *Can-Share* predicate was stated in [LIP77]: let p and q be two distinct vertices in a protection graph G_0, Can-Share(α, p, q, G_0) holds if there exists a graph derived from G_0 in which p has α rights to

q. It was argued that the Can-Share predicate has the interesting property that it can be tested in time that is linear in the size of the graph.

1.3 Related work

Various extensions have been proposed to the take-grant model. Historically, the model was first introduced in [JLS76]. [LIP77] and [SNY77] study the transfer of rights in a protection system that can be represented by the take-grant model. [SNY81a] defines the Can-Share predicate and [SNY81b] explores the notion of *theft* and *conspiracy* in the take-grant model.

[BIS77] introduces the notion of *information flow* and investigates further the transfer of rights. More recently, [BIS84] and [BIS88] extend this work by considering the notion of *theft of information.* [BIS84] also proposes an extension to the model where the concept of *group* is defined.

1.4 Thesis organization

The thesis is organized as follows:

In chapter 2, we introduce and present our extension to the take-grant model: namely, the introduction of special labels of the arcs in a protection graph. We also show how these extensions permit revocation and confinement which could not be done in the take-grant model.

In chapter 3, we show how our model can be applied to an existing system: the WM. We give an equivalent, in terms of the model, of some of the operations of the WM protection mechanism.

Chapter 4 gives a new proof for the decidability of whether a given transfer of rights is possible in a computer where the protection system is expressed in the model described in chapter 2.

In chapter 5, we state and prove a theorem that gives necessary and sufficient conditions for a transfer of rights to happen. This theorem is a generalization of a similar result obtained for the take-grant model.

Chapter 6 proposes a linear-time algorithm for verifying these conditions. In fact, the algorithm is very general and can be applied other graph theoretic formulated problems.

In chapter 7, we examine some of the assumptions made in chapter 5 which leads us to exploring the idea of trust.

Chapter 2 Presentation of the model

In this chapter we give a formal description of the model on which our work is based, and show how revocation and confinement can be addressed in this model. Informally, we say that the rights granted to an object are revoked if they can be taken back after they were granted. These rights are said to be confined if they can be held within the object to which they were granted.

The model we propose is an extension of the take-grant model, described in the introduction. Because the extension is pervasive in the sense that each component of the take-grant model are modified to carry more information, it is not possible to restrict the description to the parts that were added to the model. Some terms however are borrowed from the take-grant model and will not be redefined.

2.1 Graphical representation

The protection state of a system may be represented by a graph G = (V, E) called a *protection graph.* The graph is directed and consists of a set of vertices (or nodes), V, and a set of edges, E, called arcs. It actually is a graph theoretic reformulation of an *adjacency matrix* in the *access matrix model* [LAM71]. Because the graph need only include arcs corresponding to nonnull entries in the access matrix, it provides a compact way to present the same information given in a relatively sparse access matrix.

2.1.1 Vertices

Each node in the graph is an object or a subject in the system. Subjects, represented by filled circles, are those nodes that can initiate a rule in the model; they are active. Objects, represented by unfilled circles, are passive. We let S be the set of subjects and O the set of objects. The set of nodes is the union of S and O, $V = S \cup O$.

Since certain transformation rules state conditions that can apply to either objects or subjects, the nodes corresponding to such objects/subjects are drawn in gray. This simply is a convention for writing rules or describing graphs. For any given graph, all vertices are either subjects (filled circles) or objects (unfilled circles).

2.1.2 Arcs

Arcs in the graph relate two vertices by showing what rights the first (tail) has over the second (head).

Definition 2.1: Arcs are directed edges in the protection graph labelled with quadruplets from In x Out x N x 2^R, where:

In is the set of integer *in-values*.

Out is the set of integer *out-values*.

N is the set of elements that represent the *nature* of the arcs.

N = {static, local, formal}

R is the set of *rights*.

$R = R_G \cup R_T$, where R_G is the set of *generic rights* and R_T the set of *type-specific rights*.

$R_G = \{c, d\}$.

$R_T \supset R_C$, where $R_C = \{t, g, call, i_t, i_g, i_{call}\}$.

The two first elements, the in-values and the out-values, allow us to implement *indirection*. An arc with in-value m can "indirect" through another arc with out-value m. We say there is a *chain of indirection* from node x to node y if there is a directed path from x to y in which two adjacent edges have the same value for respectively their in-value and out-value. We will see this in more detail in section 2.3 as we explain how revocation can be performed using indirection.

The third element in the definition of an arc, its nature, is introduced to address confinement. This issue is discussed in section 2.4. For the present, it is sufficient to understand that arcs are associated with a nature that can take on three different values and that the rules described later exploit this fact. As the names (static, local and formal) suggest, however, the nature of an arc will reflect its status in a subroutine-like call chain. The nature of an arc is also referred to as a *color*. Graphically, an arc is represented by a directed edge drawn in three different ways:

- a solid line if the nature of the arc is static.
- a dashed line if the nature of the arc is local.
- a dotted line if the nature of the arc is formal.

Further, we define A_0 to be the set {static, local, formal}, A_1 to be {local, formal} and A_2 to be {formal}. These sets will be useful in the definition of the rules.

The last element in the definition of an arc is the set of rights. There are two distinct kind of rights. Type-specific rights apply to the node pointed to by the arc whereas generic rights apply to the arc itself. For example, take rights allow a subject to obtain the arcs emanatinf from a vertex. On the other hand, the generic rights c (copy) and d (delete) labeling an arc control whether that arc can be copied or deleted.

To summarize, an arc in the graph connects two vertices and is labelled with an in-value, an out-value, a nature and a set of rights. Graphically, it is represented as shown in figure 2.1. By convention, in- (out-) values are noted at the tail (head) of the arc.

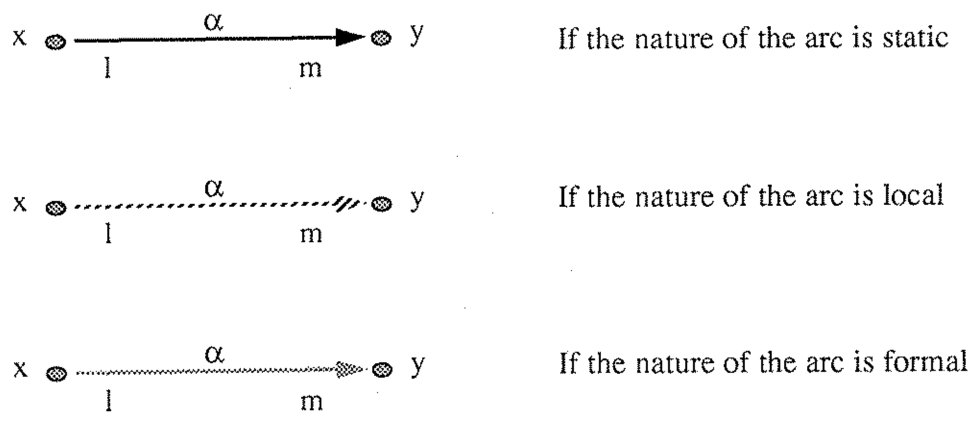

x
α
y
l
m
If the nature of the arc is static
x
α
y
l
m
If the nature of the arc is local
x
α
y
l
m
If the nature of the arc is formal

Figure 2.1

We now define a notation for the arcs that will formalize the rules that express the transition function for this model. *x(l, m, a, α)y* means that there is a directed arc from node x to node y, where l is the out-value of the arc and m its in-value, a is the nature of the arc and α are the rights. For convenience we will write x(*, *, a, {t, g})y to mean that there is an arc from x to y with, *at least*, take and grant rights and with any in- and out-values. We also want to state sometimes that some arc is not labelled with a given set of rights α. For that, we introduce the following notation. $\bar{\alpha} \in R_{xy}$ will be taken to mean $\alpha \notin R_{xy}$, R_{xy} represents the set of rights that node x has to node y.

In order to simplify the notation further, we may abbreviate the quadruplet as x[a,α]y, which means that the value of the first two parameters in the label (l, m, a, α) is irrelevant. Hence,

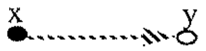
x[a,α]y $\equiv$ x(*, *, a, α)y

In the next section, we give a formal description of the graph rewriting rules of the model we propose. These rules show the possible transformations of a protection graph. In particular, when an arc is added to the graph, they specify what the nature of the arc can be.

2.2 The rules

As seen above, the take-grant model has been enriched by the introduction of indirection, colored arcs and generic rights. The rules defined in [BIS79] have to be extended accordingly.

In our model, not only do the rules depend on the rights that an object has to another one, but they also depend on the nature of the arcs labelled with those rights. Graphically, we would have to represent as many rules as there are different natures for the arcs involved. To avoid that, we show the *maximum* requirement applying to the nature of the arcs when we define the rules. The maximum is defined relative to the following ordering of the arcs (convention): *static* > *local* > *formal.*

To illustrate the above, figure 2.2 shows how the arc 

should be interpreted as either a local or a formal arc:

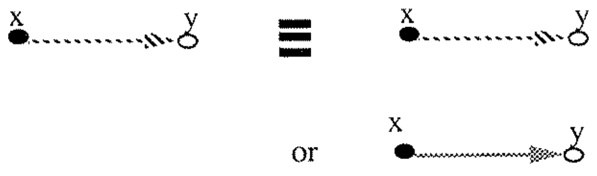


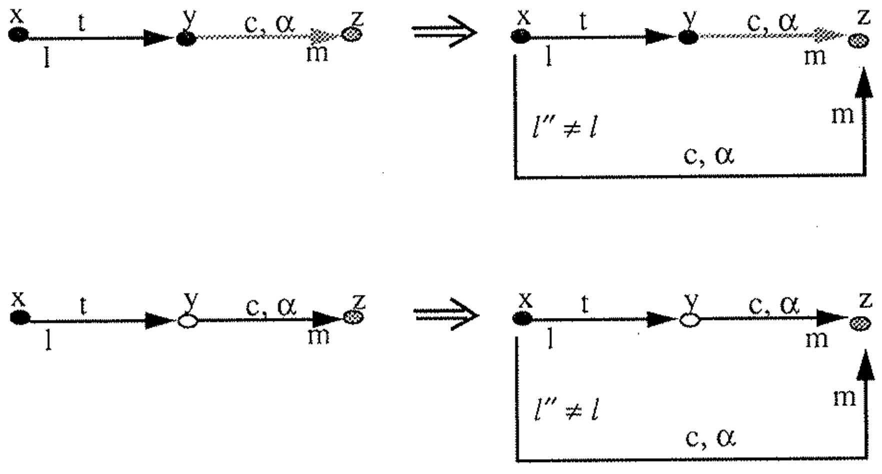
Figure 2.2

In the following sections, we describe the rules using the above conventions.

2.2.1 Take rule

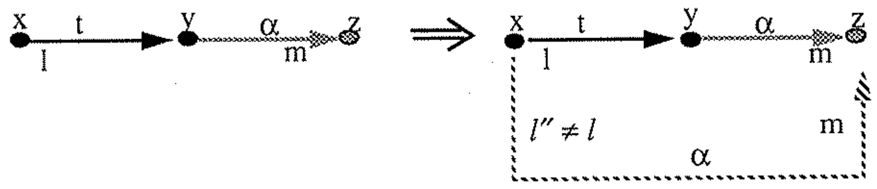
Let x, y, z be three distinct vertices in a protection graph G and let x be a subject. Let there be an edge from x to y labelled with take rights and an edge from y to z labelled with α rights. The take rule allows x to acquire the rights α to z. Upon initiation of the take rule, an arc from x to z labelled with α rights is added to the graph.

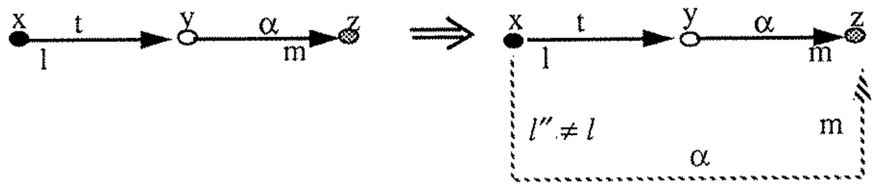
If y has c rights to z, then the arc from x to z may be of any nature {static, local, formal}. Otherwise, if y does not have c rights to z, the resulting arc must be either local or formal. Therefore, y has some control (as illustrated in the discussion on confinement in section 2.4 on page 17) over the way its arc is transferred through the use of the c right (copy right). Moreover, if y has an arc with nature different than formal, that would prevent any subject from taking the arc. This means that there are two ways for y to restrict the access that other vertices in the graph will have over z. More formally, the rules can be written as follows:



(1)

If x(l, *, *, t)y, y(*, m, a, {c} U α)z, ((y ε S ^ a ε A_2) v y ε O), α in R, and $l \neq l''$ then x(l'', m, *, {c} U α)z

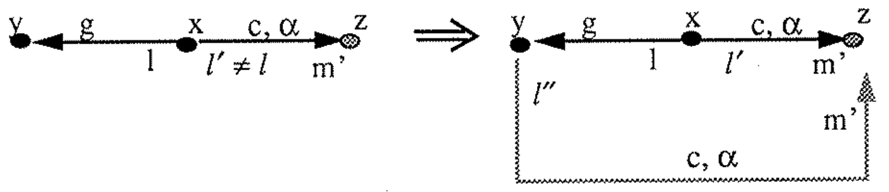


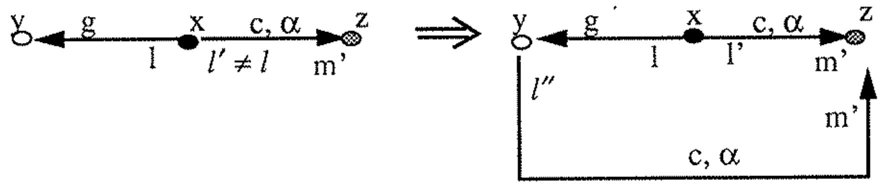


If x(l, *, *, t)y, y(*, m, a, α)z, x ε S, ((y ε S ^ a ε A_2) v y ε O),
a'' ε A_1 , α in R and $l \neq l''$
then x(l'', m, a'', α)z

2.2.2 Grant rule

Let x, y, z be three distinct vertices in a protection graph G and let x be a subject. Let there be an edge from x to y labelled with grant rights and an edge from x to z labelled with α and c rights. The grant rule allows x to give y the rights α that it has over z. The presence of the c right is necessary. The consequences of this restriction on the ability to grant rights is discussed in section 2.4.





If x(l, *, *, g)y, x(l', m', *, {c} U α)z, x ε S, ((y ε S ^ a'' ε A_2) v y ε O), α in R, and $l \neq l'$ and ∀ k st. y(k, *, *, *)y', for some y' ε V, $k \neq l''$
then y(l'', m', a'', {c} U α]z

2.2.3 Create rule

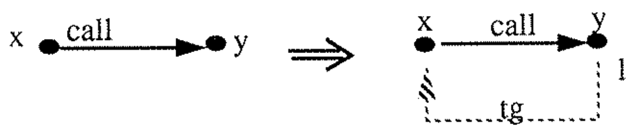
Let x be a subject in a protection graph G. If x initiates the create rule, a new vertex y is added to the graph together with an arc from x to y labelled with α rights, provided that the out-value of the arc created differs from the out-value of any other arc that x has.

$$x \in S \wedge z \notin G \wedge (\neg\exists l, m', a', r', y, x\,(l, m', a', r')\,y) \Rightarrow x\,(l, m, a, \alpha)\,z$$

Which graphically can be seen, under the conditions mentioned above, as:

2.2.4 Call rule

Let x, y be two distinct subjects in a protection graph G. Let there be an edge from x to y labelled with call rights. When x calls y, an arc from y to x with tg rights and local nature is created.

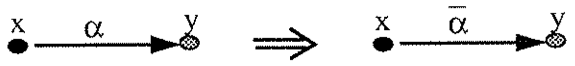


If x[*, call]y, a ε A_1 and $\neg\exists$ z ε G such that y(l, *, *, *)z
then y(l, *, a, {t,g})x

2.2.5 Restrict rule

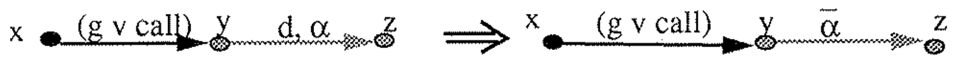
This rule has two forms:

A subject x can remove some of the rights that it has to a node.



If x[a, α]y, x ε S, a ε A_0, α in R
then x[a, $\overline{\alpha}$]y.

A subject x can delete rights in a node for which it has grant right or call right provided that the arc labelled with the right to be deleted is also labelled with the delete right.

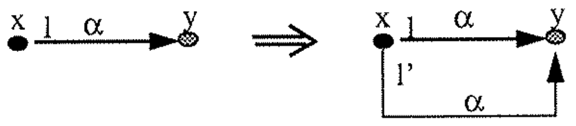


If (x[*, g]y $\vee$ x[*, call]y) $\wedge$ y[a', {d, α}]z $\wedge$ ((y ε S ^ a' ε A_2) v y ε O),
x ε S, α in R
then y[a', $\overline{\alpha}$]z.

In both cases, an arc is removed from the graph if all its non-generic rights have been deleted.

2.2.6 Duplicate rule

Let x, y be two distinct vertices in a protection graph G and let x be a subject. Let there be an edge from x to y labelled with α rights. x can duplicate an arc that it has as long as this arc stays in the same group. For example, we could have the following:



More generally,
If $x[a, \alpha]y$, $x \varepsilon S$, $a \varepsilon A_0$, $a' \varepsilon A_0$, $a > a'$ and α in R
then $x[a', \alpha]y$.

2.3 Indirection and revocation

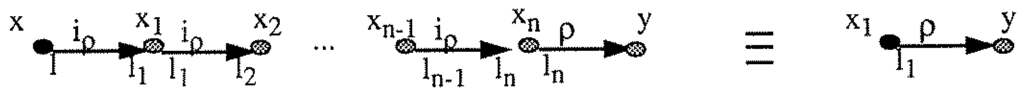
Indirection was introduced to the model in order to permit revocation. In the take-grant model [BIS79], once a node x in a protection graph grants rights to another node y, x cannot "*revoke*" the access obtained by y. In fact, only y can remove the arc newly created that gives y the rights granted by x. In the next subsections, we show how the introduction of new special rights (i_t, i_g, i_{call}) called *indirect rights* changes that. We first define indirection and explain how the rules in the model are affected. Then, we show how revocation is possible using indirection.

2.3.1 Graphical interpretation of indirection

In the original take-grant model, a node in a protection graph can only have rights to its neighbors. For a node x to take, grant or use any type-specific right on a node y, x and y must be connected by an arc in the graph.

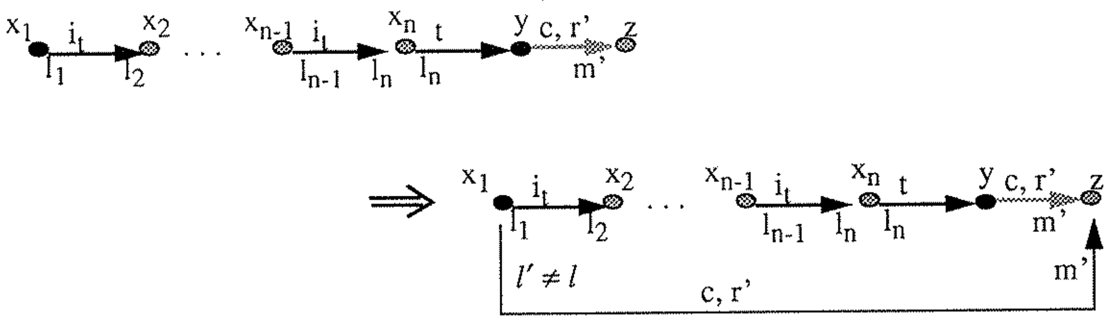
In the indirection scheme, we generalize the conditions on which an access can occur to be as follows: x can have ρ rights to y if there is a chain of indirection between x and y. This chain of indirection must be of the form: $x(l, l_1, *, i_\rho)x_1$, $x_1(l_1, l_2, *, i_\rho)x_2$, ..., $x_{n-1}(l_{n-1}, l_n, *, i_\rho)x_n$, $x_n(l_n, *, *, \rho)y$, where $x_1, x_2, ..., x_n$ are nodes in the graph.

Although no arc is actually added to the graph, it is equivalent to think of a chain of indirection as an arc between x and y as illustrated below:



$x(l, l_1, *, i_\rho)x_1, x_1(l_1, l_2, *, i_\rho)x_2, \ldots, x_{n-1}(l_{n-1}, l_n, *, i_\rho)x_n, x_n(l_n, *, *, \rho)y$
$\Leftrightarrow x(l, *, *, \rho)y$

As a consequence, a subject may access remote nodes in the graph. In particular, a subject can take, grant or call a remote node if there is a chain of indirection between the two nodes. What follows is the indirect version of the take rule (1) defined in section 2.2.1 on page 11. Similar interpretations can be made for the other rules of the model.



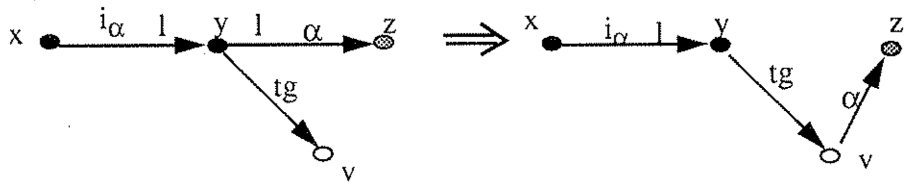
If $\exists\ (l_1, \ldots, l_n) \varepsilon N^n, (x1, \ldots, xn) \varepsilon V^n, (y, z) \varepsilon V^2$
such that $x_1(l_1, l_2, *, i_t)x_2, \ldots, x_{n-1}(l_{n-1}, l_n, *, i_t)x_n, x_n(l_n, *, *, t)y,$
$y(*, m', a', \{c\} \cup r')z, ((y\ \varepsilon\ S \wedge a'\ \varepsilon\ A_2) \vee y\ \varepsilon\ O)$ and $l \neq l'$
then $x_1(l', m', *, \{c\} \cup r')z$

Conceptually, the existence of a chain of indirection between two nodes creates a "virtual" arc in the graph between the two nodes. This "virtual" arc disappears as soon as the chain is "broken" (an arc in that chain is removed or the out-value of an arc is no longer equal to the in-value of subsequent arc). In the next subsection, we explain how the introduction of indirection in the take-grant model permits revocation.

2.3.2 Revocation

The main reason for adding indirection to the take-grant model is to permit revocation. The ability to revoke a right that has been granted is important in most systems. For instance, suppose a physical page is allocated to a process in a virtual memory computer system. When this page is reallocated to a different process, the access that the first process has to the page must be revoked. In this context, revocation is essential given the fact that computer systems have limited physical memory. As another example, consider a client-server situation. After the server completes the service, the client should be able to prevent the server from accessing the client resources that were made available to the server.

Indirection allows revocation as depicted in figure 2.3 below. In graph (a), x has α rights to z since there is a chain of indirection between x and z. If the arc y[*, α]z is removed from the graph (and assuming no other node has direct α rights to z), then no node in the graph has α rights to z, and, as we will see later, no node will ever have α rights to z. If y grants to v the α rights to z before removing the arc y[*, α]z, x still cannot obtain rights to z because the path xyvz is not a chain of indirection. Thus, y effectively revoked the access that x had to z.



(a) Figure 2.3 (b)

In general, a node x in a protection graph can have the access it has to another node y revoked if any arc along the chain of indirection between x and y is removed or has its out-value changed. A subject that can thus "break" the chain of indirection can therefore revoke the access that x has to y.

By choosing to relabel an arc (instead of removing it) along that chain of indirection so that the out-value is changed, a subject can allow other nodes to indirect through the arc.. This *selective* revocation is desirable. Consider a subject that provides the access to a given resource to several nodes. This subject can revoke that access to the resource that a node has and grant it to another node without the first node being able to access the resource any longer. In a computer system, for example, after deallocating a page from a process, the page can be reallocated safely if the right that the first process has to that page is revoked.

2.4 Confinement

Confinement must be ensured in a computer system where a server-client-like mechanism is provided. Confinement is ensured if the rights to objects passed as parameters by the client cannot be given away by the server.

In order to model policies where confinement is to be ensured, we introduced generic rights and colored arcs as extensions to the take-grant model. In particular, we want to model systems that allow subroutine-like call chains in which access rights to a resource can be passed along as parameter from one subject to another and removed as the called subjects return.

In the take-grant model, when a subject grants rights to another subject these rights cannot be taken back. This is a result of the fact that a subject cannot distinguish between arcs that were granted and arcs that it had. As a consequence, parameter passing cannot be represented in that model.

In our model, we force this distinction of the arcs by labelling them with a nature (or color). To make parameters available to others, a subject labels its arcs to these parameters as formal. When taking rights to these parameters, subjects in the protection graph obtain rights "confined" to be on arcs labelled local or formal. In fact, subsequent subjects in the call chain can only obtain local or formal rights to the parameters. In that sense and because we have a way to specify that rights to parameters have to be given back, confinement can be enforced. The reason why we can specify that rights to parameters have to be given back is that subjects "know" whether they access parameters (local and formal arcs) or private resources (static arcs). As a result, we can model the releasing of rights to parameters by having a subject remove all its local and formal arcs.

Confinement is not always desirable, however. When granted, the access rights to a resource do not necessarily have to be confined. For this reason, we introduced the copy right c in the model. If the rights granted to a subject include the copy right, the subject obtains an arc that can be colored static. Thus, by deleting the copy right on parameters, the “caller” can control whether or not confinement is enforced.

Chapter 3 The WM protection mechanism

In chapter 2, we presented a model for verifying security of a protection system. Before deriving theoretical results for the model, we show in this chapter how the model can be used to model formally the protection mechanism of a capability-based machine: the WM. A complete description of the WM protection mechanism can be found in [WUL91b]. In this chapter, we briefly review some aspects of the mechanism and show how they can be modelled. The WM is of special interest since it is a capability-based machine that provides with a protection mechanism that can easily be represented using our model.

3.1 Objects

In this section, we describe how the different objects of the WM mechanism are modelled. The WM mechanism supports several types of *segments* referred to as nodes in our model. Because they contain information essential to the construction of the graph, two of them are examined: *Tasks* and *C-Lists*. Other WM segment types (pages for example) may be thought of as objects without out-arcs.

3.1.1 Tasks

Tasks are threads of execution in the system. They are active entities and are represented by subjects in the model. With each task is associated a set of special registers that can hold *capabilities*. A capability is a reference to a segment in the system. Capabilities are modelled as arcs in a protection graph. Hence, tasks have out-arcs in the model. The capability registers are divided into three sets: the *static*, *local* and *formal* registers. Capabilities stored in capability registers are represented by arcs with nature:

- static if the capability is in a static register
- local if the capability is in a local register
- formal if the capability is in a formal register

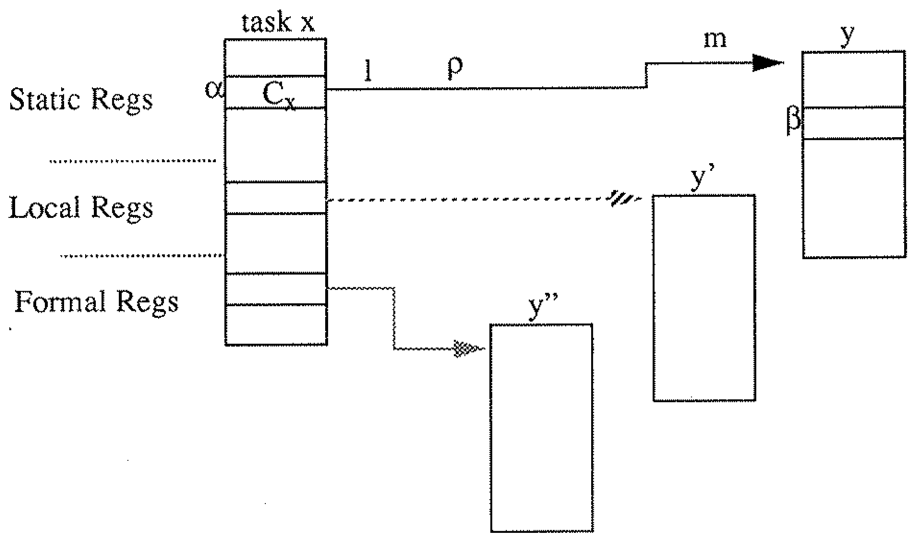


Figure 3.1

More precisely, let us see how a capability is associated with an arc in the graph. Figure 3.2 gives the representation of a WM capability. The labels on the corresponding arc are determined from the different fields of the capability:

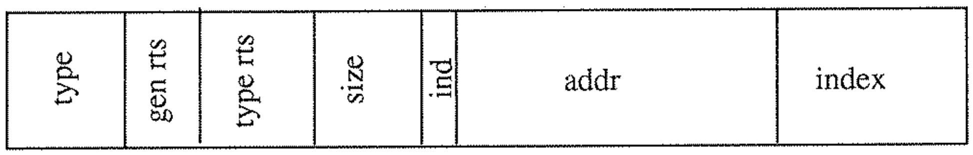


Figure 3.2

- the *gen rts* and *type rts* fields together with the *ind* field represent the set of rights on the arc. If the ind field is set to 0, all the type rights are interpreted as a subset of {t, g, call, r, w}. If the ind field is set to 1, the type rights are interpreted as a subset of $\{i_t, i_g, i_{call}, i_r, i_w\}$.
- the generic rights (gen rts field) are not affected by the value of the ind field. They represent the copy right and the delete right.
- the type rts field specifies the type of the segment referenced to by the capability.

This field determines the interpretation of the type rights field. For example, if the capability names a task, the type field corresponds to a subset of {t, g, call, i_t, i_g, i_{call}}. If it names a page, the type field corresponds to a subset of {r, w, i_r, i_w}.
- the index field indicates the in-value of the arc.
- the out-value is obtained by determining the position of the capability in the set of capability registers.

3.1.2 C-Lists

C-Lists are special WM segments that can only hold capabilities. They are represented by objects in the model. The out-value of an arc corresponding to a capability contained in a C-List is equal to the number of the slot within that C-List where the capability is stored. The in-value of the arc is indicated in the index field (figure 3.2).

3.2 Operations

Operations of the WM protection mechanism involve transformations of the protection graph associated with the state of the system in our model. In what follows we describe some of these operations: the load capability, store capability, call and reply operations.

3.2.1 Load capability operation

The load capability operation is performed by the LdCapa instruction. LdCapa moves a capability from a C-List or capability register to a capability register. Type and rights checking is done prior to the execution of the instruction.

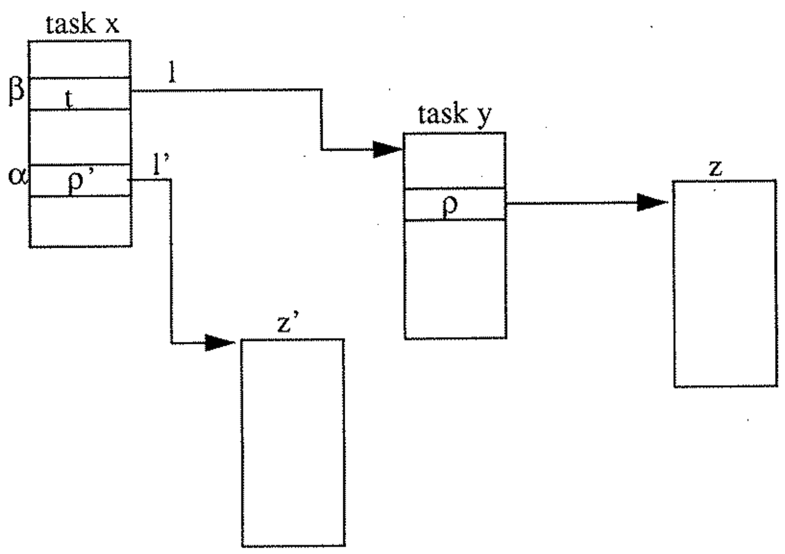


Figure 3.3

In the case of a task loading a capability from another task (figure 3.3), we show how to represent the execution of the load instruction in the model. In "LdCapa α, β, μ", α indexes the capability register in which to load the capability for task y. β is a general virtual address that indicates the capability register in y containing the capability to be loaded. The corresponding graph is given in figure 3.4. Once "LdCapa α, β, μ" is executed, and if all the conditions are satisfied, the following will happen (as illustrated in figure 3.4):

- x removes its arc to z' (restrict rule)
- x takes y[formal, ρ]z from y
- x restricts its rights to z to be ρ masked by μ (also written ρ/μ)

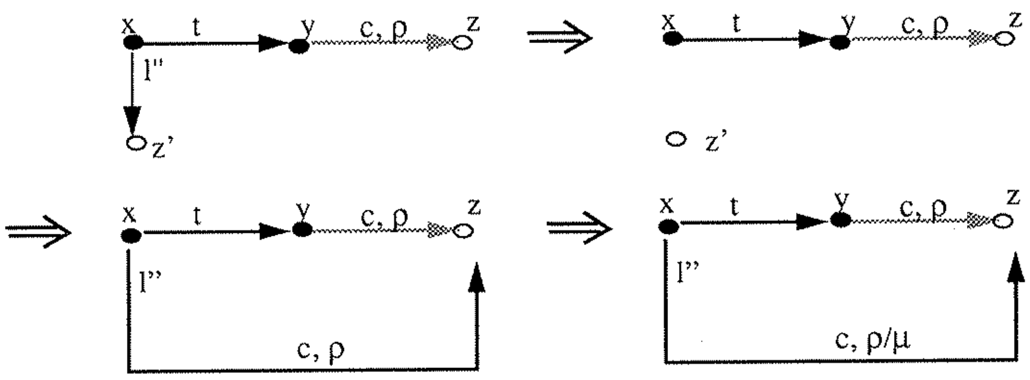


Figure 3.4

3.2.2 Store operation

The store capability operation is performed by executing the StCapa instruction. StCapa moves a capability from a capability register to a C-List or capability register. Type and rights checking is done prior to the execution of the instruction.

StCapa α, β, μ is interpreted in terms of graph rewriting rules as:

- x removes y[formal, ρ]z (restrict rule)
- x grants [formal, ρ']z to y
- x restricts those rights to be ρ'/μ

We can see that both the load and the store operations involve several rules in the graph. The implications of this are discussed in section 5.1.1.

3.2.3 Call operation

The call operation is performed by executing the call instruction (in terms of graph rewriting rules, the subject associated with the caller initiates the call rule). When a task calls another one, the called task obtains a capability for the caller in its local capability registers. This capability allows the called task to load the formal capabilities of the caller. To make these capabilities available to another task (a subsequent called task), the called task should load them into its formal registers. Otherwise, it should load them into its local registers.

The reason why the called task obtains a *local* capability for its caller is twofold. First, it ensures that no other task can take the capability. Second, the capability will be removed when the called task replies as explained in the next section.

3.2.4 Reply operation

The reply operation is performed by executing the reply instruction. The effect of that instruction is to remove all capabilities from the local and formal capability registers of the called task before giving control to the caller. In terms of the model and although there is no reply rule, the subject associated with the called task simply removes all of its local and formal arcs (restrict rule).

3.3 Summary

In the previous sections, we sketched how the model presented in chapter 2 can be used to model the WM protection mechanism. Although there is no one-to-one correspondence between the two, the correspondence is tight. In particular, different states of the machine are represented by different graphs and any transfer of rights corresponds to a sequence of graphs derived from one another using graph rewriting rules of the model. The model is therefore useful in that if an access flow is possible in the WM, it is also possible in the model. It is conceivable however that a particular transfer of rights that can occur in the model has no equivalent in the protection mechanism. This has no significant impact since for the formal theorems to be applied to the WM, the WM mechanism need only be more restrictive than the model.

Throughout the rest of this thesis (except for section 5.1.1 on page 26), we do not consider the WM protection mechanism and focus on the model. We establish important results for the model. These results can be applied with slight modifications to the WM.

Chapter 4 Decidability issues

The take-grant model has been studied in depth and many useful properties are known about it. In particular, it is possible to determine whether a particular transfer of rights can occur in a system modelled by it. It is not clear *a priori* whether these results still hold for the extended model presented in chapter 2.

In what follows, we give a new proof to show that it is decidable whether one subject/object can acquire rights to another in the extended model. For that, in this chapter we establish the equivalence of two versions of the model: the one defined in chapter 2 and a related one that allows a node to have reflexive rights (rights to itself). In the latter model, sharing is a trivially decidable problem.

4.1 A variant of the model with reflexive rights

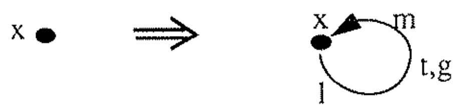
There are several variants of the take-grant model (see [JBI84]). Each of them can be represented by a pair ($\mathcal{G}$, $\mathcal{R}$), where $\mathcal{G}$ is a formal description of a graph and $\mathcal{R}$ a set of rules. Let ($\mathcal{G}_M$, $\mathcal{R}_M$) be the model defined in chapter 2. We define ($\mathcal{G}_M$, $\mathcal{R}'_M$) to be the same as ($\mathcal{G}_M$, $\mathcal{R}_M$) with the create rule in $\mathcal{R}_M$ removed and a self-grant rule introduced.

Self-grant rule:

A subject x can grant rights to itself if the out-value of the arc created differs from the out-value of any other arc emanating from x.

$x \in S \Rightarrow x\,(l, m, a,\ \{t, g\}\,)\,x$

Which graphically can be depicted, under the conditions mentioned above, as:



This model ($\mathcal{G}_M$, $\mathcal{R}'_M$) is attractive because it does not have any create rules. This means that given an initial graph G_0, the size of any derived graph (using rules in $\mathcal{R}'_M$) is bounded.

It also means that there is a finite number of graphs that can be derived from G_0 using rules in $\mathcal{R}'_M$. This makes any property associated with G_0 decidable. In particular, it is decidable whether a node in G_0 can obtain rights to another: consider all possible graphs derived from G_0 and check whether there is an arc between the two nodes.

Yet, $(\mathcal{G}_M, \mathcal{R}'_M)$ is as powerful as $(\mathcal{G}_M, \mathcal{R}_M)$ in the sense that any transfer of rights that uses rules in $\mathcal{R}_M$ can also be obtained using rules in $\mathcal{R}'_M$. The reason is that the strength of the create rule in $\mathcal{R}_M$ lies less in the creation of a node than in the creation of an arc as illustrated in the example below. In $\mathcal{R}'_M$, creation of an arc is still possible through the self-grant rule.

As an example, consider the graph in figure 4.1 a. Intuitively, it does not seem that p can obtain a rights to q since p cannot take form s and s cannot grant to p. However, the transfer of rights from s to p is indeed possible if s can take from p the ability to grant rights to a node from which p can take rights. This can be done using rules in $\mathcal{R}_M$ (figure 4.1 b) as well as rules in $\mathcal{R}'_M$ (figure 4.1 c).

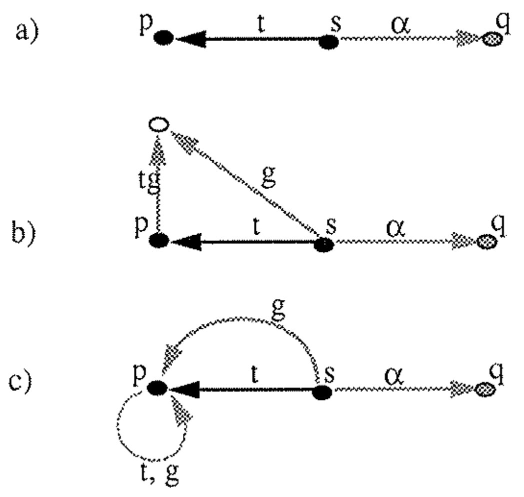


Figure 4.1

In figure 4.1.b, p obtains rights to q by the following sequence of operations.:

p creates o

s takes (g to o) from p

s grants (α to q) to o

p takes (α to q) from p

In figure 4.1.c, p obtains rights to q by the following sequence of operations.:

p self-grants take and grant rights

s takes (g to p) from p

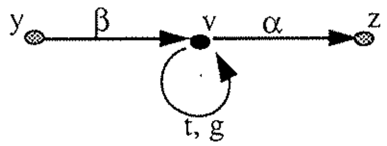
s grants (α to q) to p

4.2 Equivalence of the model

Lemma 4.1: Let p, q be vertices in a graph $G_0(V, E)$, p can obtain (α to q) in $(\mathcal{G}_M, \mathcal{R}_M)$ if and only if p can obtain (α to q) in $(\mathcal{G}_M, \mathcal{R}'_M)$.

Proof: Clearly, if p can obtain (α to q) in $(\mathcal{G}_M, \mathcal{R}_M)$ it can do so by creating subjects only. Consequently, we restrict ourselves to transformations of G_0 involving creation of subjects. We first show that two subjects connected by an edge labelled with {t, g} rights are equivalent to a single one with an arc to itself labelled {t, g} (figure 4.2 a and b).

a)



b)

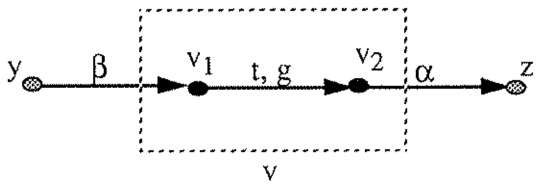


Figure 4.2

All arcs originated by v_1 can be obtained by v_2 and vice-versa. All nodes that have rights to v_1 (resp. v_2) can obtain the same to v_2 (resp. v_1). More formally, in $(\mathcal{G}_M, \mathcal{R}_M)$:

$\forall x \in V, \forall \alpha \in R$, $x[*, \alpha]v_1$ in G_0 => $\exists G$ such that G_0->G and $x[*, \alpha]v_2$ in G

$\forall x \in V, \forall \alpha \in R$, $x[*, \alpha]v_2$ in G_0 => $\exists G$ such that G_0->G and $x[*, \alpha]v_1$ in G

$\forall x \in V, \forall \alpha \in R$, $v_1[*, \alpha]x$ in G_0 => $\exists G$ such that G_0->G and $v_2[*, \alpha]x$ in G

$\forall x \in V, \forall \alpha \in R$, $v_2[*, \alpha]x$ in G_0 => $\exists G$ such that G_0->G and $v_1[*, \alpha]x$ in G

In fact, in section 5.2 on page 30 we will see that a group of connected subjects can be thought of as a single node. (v_1, v_2) fall into this category and conceptually, can be considered as a single node. The existence of an arc between v_1 and v_2 translates to an arc from v to itself.

p can obtain (α to q) in ($\mathcal{G}_M$, $\mathcal{R}_M$) means that there exists a graph G derived G_0 such that p[*, α]q. During the derivation of G, we look at the steps where a subject v_2 is created by a subject v_1. We group v_1 and v_2 into one subject v as described above. It can easily be seen that the resulting graph could have been obtained by using rules in $\mathcal{R}'_M$ which implies that, in ($\mathcal{G}_M$, $\mathcal{R}'_M$), there exists a graph G derived from G_0 such that p[*, α]q.

By a similar argument the converse is also true. Whenever a self-grant rule is activated in ($\mathcal{G}_M$, $\mathcal{R}'_M$) we expand the node v that activated it to (v1, v2) as shown in figure 4.1 without loss of power. If p can obtain (α to q) in ($\mathcal{G}_M$, $\mathcal{R}'_M$) we could therefore derive a graph G' from G_0 in ($\mathcal{G}_M$, $\mathcal{R}_M$) such that p[*, α]q in G'.

Theorem 4.1: Let p, q be vertices in a graph G_0(V, E), p can obtain (α to q) in ($\mathcal{G}_M$, $\mathcal{R}_M$) is decidable.

Proof: In ($\mathcal{G}_M$, $\mathcal{R}'_M$), we can easily verify whether p can obtain (α to q) by computing the transitive closure $\overline{G}_0$ of G_0. If an arc of the form p[*, β]q exists in $\overline{G}_0$, where β contains α it means that p can obtain (α to q) in ($\mathcal{G}_M$, $\mathcal{R}'_M$). It also means that p can obtain (α to q) in ($\mathcal{G}_M$, $\mathcal{R}_M$) as a result of lemma 4.1.

In the next chapter, we give more details about how to characterize that a node can obtain rights to another node and elaborate on the methodology. In chapter 6, we describe an algorithm to verify whether p can obtain (α to q) based on this characterization.

Chapter 5 Transfer of rights

In chapter 4, we showed that it is decidable whether a node in a protection graph can obtain rights to another using rules in our model. In this chapter, we derive conditions for the transfer of rights. In order to establish the theorem that gives those conditions, we make a number of assumptions that facilitate the formulation of the theorem.

5.1 Assumptions

When we say that a node p can obtain rights to a node q, we assume all nodes but q "conspire" for this purpose. Whether or not the transfer of rights from one node s (with rights to q) to another p can be achieved, given that some nodes in the graph can be *trusted*, is discussed later in chapter 7. In this section, we describe some of the basic aspects of how a subject can *behave* in a way that makes a transfer of right possible. Throughout the rest of the chapter, we will assume that all subjects (but q) have the characteristics of a *conspirator* described below.

The first observation is that a subject can manage its own arcs through the duplicate rule so that they can have any out-value. Hence, we will assume that the out-value of an arc originated by a "non-trusted" subject (all but q) is irrelevant. The second observation is that objects can also have arcs with any out-value. Since a subject can take arcs from an object and grant it back with a different out-value (as illustrated in figure 5.2 below), in- and out-values do not play any part in the sharing of rights.

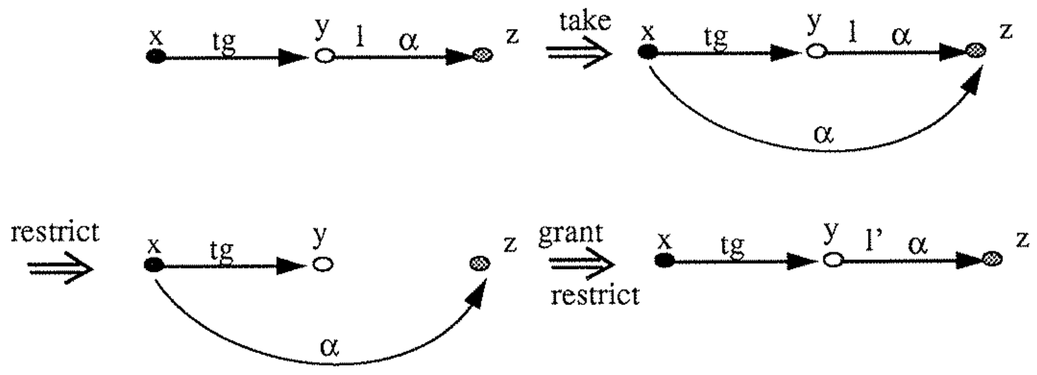


Figure 5.2

Figure 5.2 shows how a *non-trusted* subject can modify the out-value of an arc so as to allow one node to obtain rights to another. In this graph, y has α rights to z. Let α contain the copy right. The arc between y and z is labelled with an out-value l. Assume that there is a node w that has indirect rights to y. Let l' be the in-value of the arc between w and y. Unless l=l', w cannot access z. This is where the *participation* of x is needed. By initiating a sequence of graph rewriting rules, x can change l to be l' as depicted in figure 5.2.

5.2 Subject-only graph

In this section, we want to determine how a subject p in a subject-only graph G can acquire rights to another subject q. For this, we first show that it is possible only if some group s of subjects already has the rights to q, and second that for p to obtain those rights there has to be a path of a given form between p and s in G.

To facilitate the description of paths involved in a transfer of rights, we introduce a few notations. Two vertices that are connected by a path in which all edges are labelled with rights contained in a set of rights P are defined to be P-connected. More formally:

Definition 5.1: Two vertices x and y in a graph G are said to be P-connected, where P is a set of rights, if there exists an undirected path $x[*, \alpha_1]x_1[*, \alpha_2]x_2...[*, \alpha_{n-1}]x_{n-1}[*, \alpha_n]y$ in G where for all i, α_i is a set of right such that $\alpha_i \cap P \neq \emptyset$

Definition 5.2: A vertex is said to be P-connected to a set of vertices, S, if x is P-connected to every vertex in S.

Given these definitions, we establish a lemma that shows how rights can be transfered from one subject to another to which it is R_C-connected. Recall that R_C is a set of special rights as defined in section 2.1.2 on page 8. $R_C = \{t, g, call, i_t, i_g, i_{call}\}$.

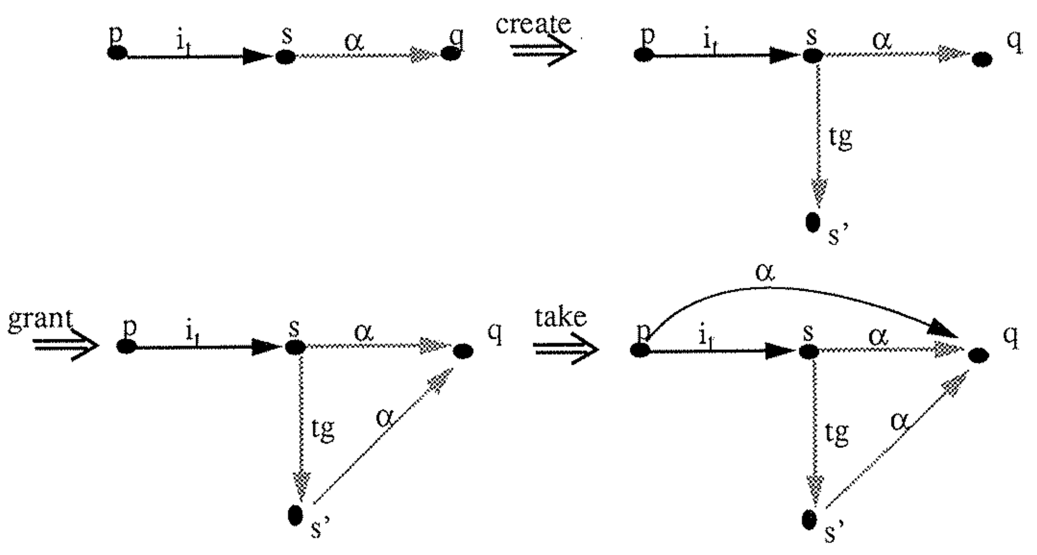
Lemma 5.1: Let G_0 be a subject-only graph, $p, q, s \in G_0$
(p and s are R_C-connected) and s[a, α]q => $\exists G, G_0 \rightarrow G$ and p[*, α]q in G.

Proof: We will prove the lemma by induction on the length *l* of the path between p and s (this path exists because p and s are R_C-connected). The basis step will be a case analysis on the different possible rights ρ supported by the only edge in the path (length *l* = 1). The induction is trivial.
basis: *l* = 1.

If p and s are R_C-connected, there has to be an arc between p and s labelled with rights ρ, such that ρ contains at least one of the following: t, g, call, i_t, i_g, i_{call}. We consider all six possibilities and show that for each one of them p can obtain the rights that s has to q.

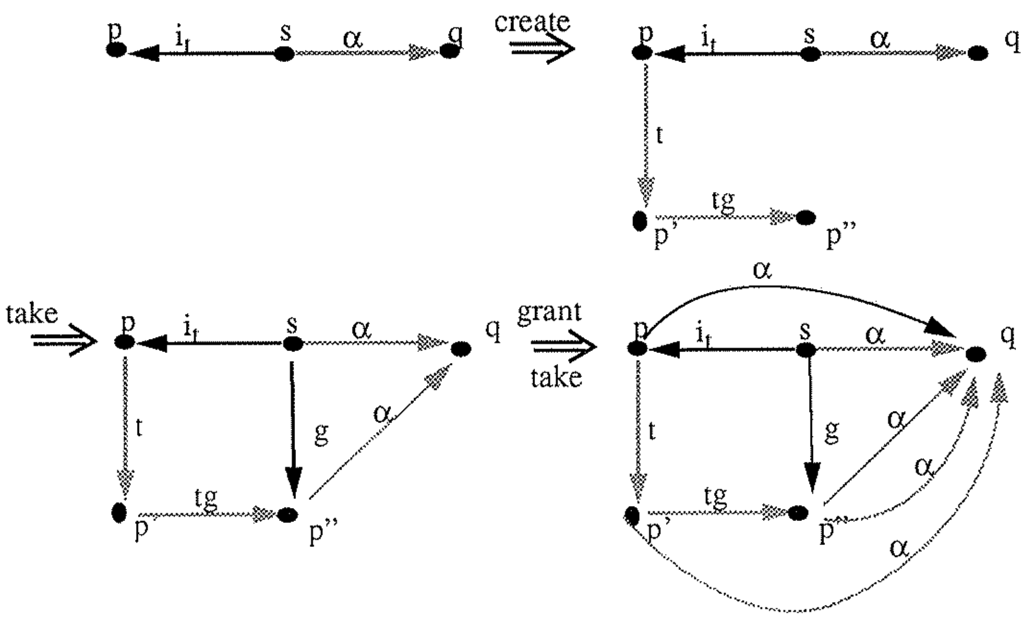
Case 0: The proofs for ρ in {t, g} can be found in [JLS76].

Case 1: $\rho = i_t$



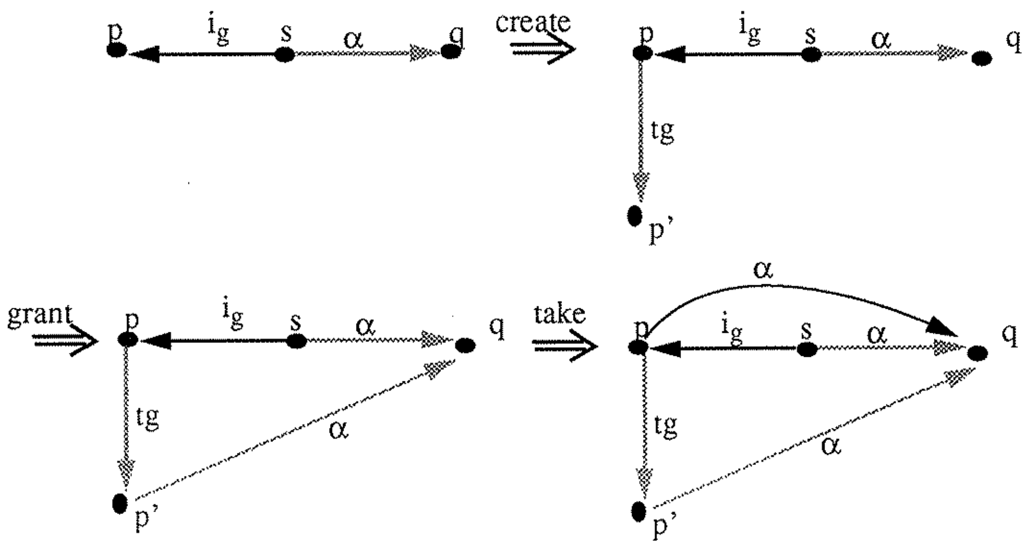
s starts by creating a subject for which it has take-grant rights and then grants to s' its α rights to q. By choosing the out-value of the arc created appropriately and by having this arc be a formal arc, s allows p to take rights from s'.

If the arc is reversed so that s has i_t rights to p, there exist a similar kind of construction that enables p to obtain α rights to q.



p creates p'.
p' creates p''.
s takes (g to p'') from p' using the chain of indirection s, p, p'.
s grants (α to q) to p''.
p' takes (α to q) from p''.
p takes (α to q) from p'.

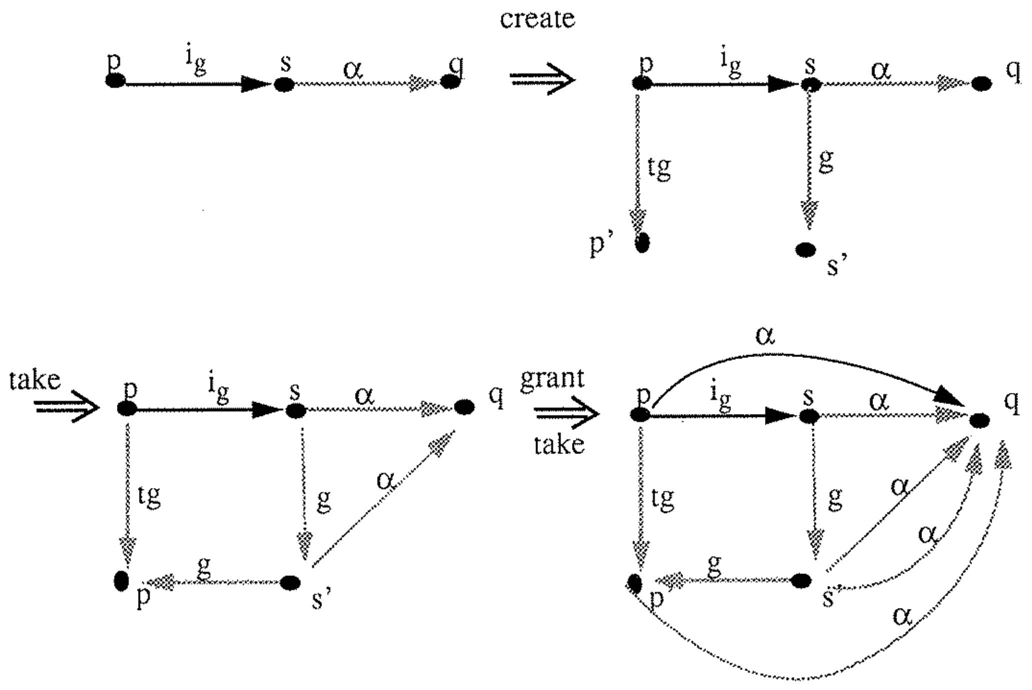
<u>Case2</u>: $\rho = i_g$



p creates p'.

s grants (α to q) to p' using the chain of indirection s, p, p'.

p takes (α to q) from p'.



p creates p'.

s creates s'.

p grants (g to p') to s' using the chain of indirection p, s, s'.

s grants (α to q) to s'.

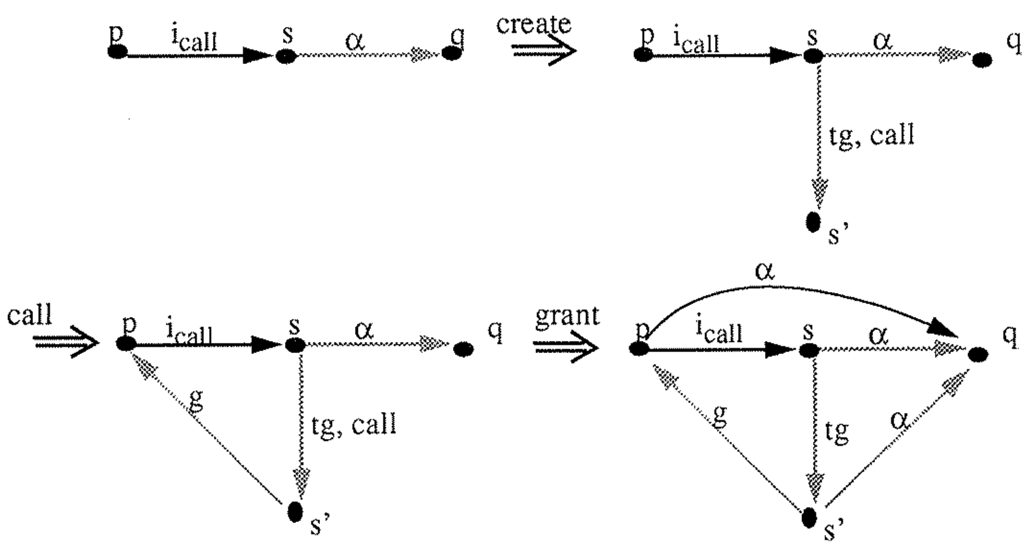
s' grants (α to q) to p'.

p takes (α to q) from p'.

Case3: ρ=call

In this case, by applying the call rule, an arc between p and s with {t, g} rights gets added to the graph. It follows from case0 that p can α q.

Case4: $\rho=i_{call}$

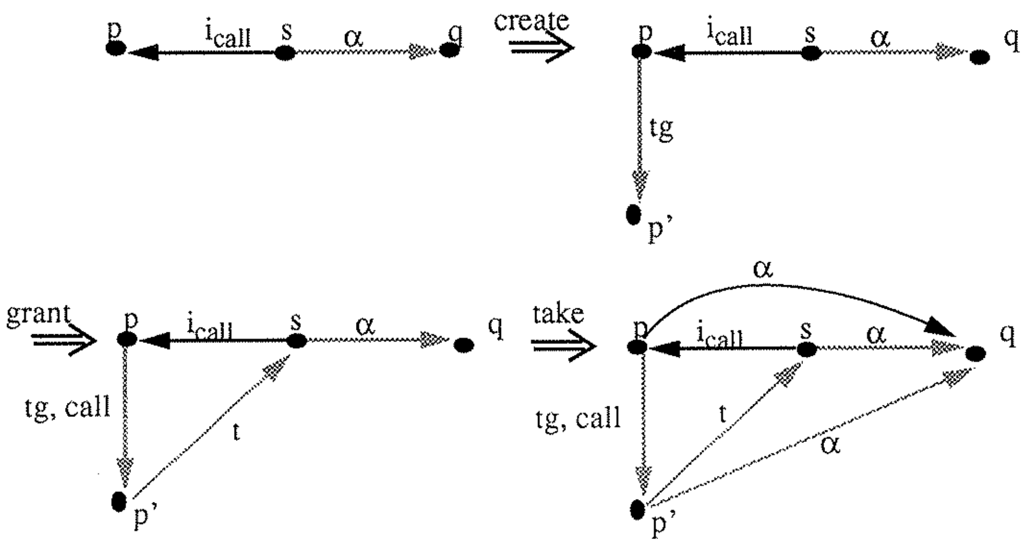


s creates s'.

p calls s' using the chain of indirection p, s, s'.

s grants (α to q) to s'.

s' grants (α to q) to p.



p creates p'.
s calls p' using the chain of indirection s, p, p'.
p' takes (α to q) from s.
p takes (α to q) from p'.

This concludes the case analysis and proves that two R_C-connected neighbors in a subject-only graph can share rights.

induction: assume that the lemma holds for any two nodes R_C-connected by a path of length l. We must show that the lemma holds for any two nodes R_C-connected by a path of length l+1. Let $px_1...x_ls$ be such a path.

$px_1...x_ls$ is R_C-connected implies that all edges along the path are labelled with rights in R_C (by definition). In addition, all edges of $x_1...x_ls$ are also edges of $px_1...x_ls$ which means that x_1 and s are R_C-connected. By induction hypothesis, there exists a graph G derived from G_0 in which $x_1[*, \alpha]q$.

On the other hand, p and x_1 are RC-connected by a path of length 1. Therefore, we can apply the results obtained earlier for the basis: There exists a graph G' derived from G in which $p[*, \alpha]q$ and the lemma holds for two nodes R_C-connected by a path of length l+1.

This concludes the proof as it was shown by induction that, for two R_C-connected nodes p and s in a graph G_0 such that s[*, α]q, there exists a graph G derived from G_0 in which p[*, α]q.

Next we state a lemma that shows that for a node to obtain rights to another, these rights had to exist in the initial graph.

Lemma 5.2: Let G_0 be a graph, $p \in G_0$, $q \in G_0$, $\alpha \in R$

$\exists G, G_0 \rightarrow G$ and p[*, α]q in G => $\exists (s \in G_0)$, s[*, α]q

Proof: It is an immediate result of the fact that the only way to create a right to a node in the model is to use the create rule which operates on *not-yet-existent* nodes. In fact, no rule introduces a new right to an existing node: rather, rules create copies of existing rights.

We establish now that two subjects connected by a path with edges labelled in R_C can share rights.

Theorem 5.1: Let G_0 be a subject-only graph, $p \in G_0$, $q \in G_0$, the following propositions are equivalent:

1- $\exists G, G_0 \rightarrow G$ and p[*, α]q in G

2- $\exists \alpha_i$, $i = 1 \ldots n$, $\bigcup_i \alpha_i = \alpha$ and $\exists (s_i \in G_0)$, s_i[*, α_i]q and (p and s_i are R_C-connected)

Proof: We first show that p[*, α]q in G implies the existence of a set of subjects in G_0 with α rights to q. Then, we prove the converse.

Proposition 1 implies proposition 2: If there was no s_i with α_i rights to q, no node would ever get those rights (Lemma 5.2). This contradicts condition 1 since α contains α_i. For a given s_i, if there is no R_T-path between p and s_i, p cannot obtain rights from s_i (see [LIP77]). Let us consider the set of all subjects s_i in G_0 such that s_i[*, α_i]q, with α_i contained in α and such that p and s_i are {t, g, call}-connected. Lemma 5.1 shows that p can then obtain $\bigcup \alpha_i$ rights to q. Now if $\bigcup \alpha_i \neq \alpha$, let $\beta = \alpha - \bigcup \alpha_i$. β is contained in α and p cannot obtain it which contradicts proposition 1.

Proposition 2 implies proposition 1: For all (s_i, α_i), lemma 5.1 shows that

$\exists G_i, G_0 \to G_i$ where p can α_i q. Therefore, given the fact that no node, arc or right is being removed to derive G_i from G_0, there exist a succession of graphs $G'_1, ..., G'_n$ such that $G_0 = G'_0 \to G'_1 \to ... \to G'_n$ and G'_i is derived from G'_{i-1} the same way G_i is derived from G_0. For $G=G'_n$, proposition 1 holds since p[*, α_i]q for all i in {1, ..., n} and $\bigcup \alpha_i = \alpha$.

We now have a way to know if a subject p in a subject-only graph can obtain rights to another subject q: there have to be subjects in the graph that have those rights and that are R_C-connected to p.

Notice that the fact that q is a subject did not occur in the proof. Hence, as a corollary we have the following result which will be used in the next section.

Corollary 5.1: Let G_0 be a subject-only graph, $p \in G_0$. Let q be an object and $G'_0=G_0 \cup \{q\}$. For α in R, p can obtain (α to q) in G'_0 if and only if there exist a set of subjects S in G_0 such that S[*, α]q and p and S are R_C-connected.

5.3 Subject-object graph

In the previous section, we determined conditions under which a particular transfer of rights is possible in a subject-only graph. We now consider the more general case of a subject-object graph and state a theorem that gives necessary and sufficient conditions for that transfer to be possible. But first, we introduce generalizations of some definitions found in [SNY81a]. These definitions are useful in stating the theorem and in reasoning about transfer of rights.

5.3.1 Definitions

An island is a subject-only subgraph of that graph with a particular structure.

Definition 5.2: An *island* in G_0 is a maximal R_C-connected subgraph of subjects.

A bridge is a path that connects two islands and along which rights can be transferred. More formally, if we associate with each arc in the graph letters taken from the set of rights combined with the direction of the arc.

Definition 5.3: We say that there is a *bridge* between islands I and J if there exist p in I, s in J and $o_1, ..., o_n$ objects in G_0 such that $po_1...o_ns$ forms a path with associated word in the language L, $L = L_0 \cup L_1 \cup L_2 \cup L_3$, where:

$$L_0 = (\overrightarrow{i_t} + \overrightarrow{t})^*$$

$$L_1 = (\overleftarrow{t} + \overleftarrow{i_t})^*$$

$$L_2 = (\overrightarrow{i_t}^* \; \overrightarrow{t})^* \; \overrightarrow{i_g}^* \overrightarrow{g} \; (\overleftarrow{i_t}^* + \overleftarrow{i_g}^*)(\overleftarrow{t} \; \overleftarrow{i_t}^*)^*$$

$$L_3 = (\overrightarrow{i_t}^* \; \overrightarrow{t})^* (\overrightarrow{i_t}^* + \overrightarrow{i_g}^*) \overleftarrow{g} \; \overleftarrow{i_g}^* (\overleftarrow{t} \; \overleftarrow{i_t}^*)^*$$

An initial span is a path along which the first node can transmit rights.
A terminal span is a path along which the first node can obtain rights.

Definition 5.4:

x in G_0 *initially spans* to y if there is a path between x and y with associated word in the language generated by: $I = (\overrightarrow{i_t} + \overrightarrow{t})^* \overrightarrow{i_g}^* \overrightarrow{g}$

x in G_0 *terminally spans* to y if there is a path between x and y with associated word in the language generated by: $T = (\overrightarrow{i_t}^* \; \overrightarrow{t})^*$

We define a predicate that asserts whether a node can obtain rights to another.

Definition 5.5: Let α be in 2^R, p and q in G_0, *Can-Share(α, p, q, G_0)* is said to be true if there exist a graph G such that G_0->G and p[*, α]q in G.

5.3.2 A necessary and sufficient condition for sharing rights

In this section, we will show how a node p in a protection graph can obtain rights to another node q; the answer is that it is possible if and only if there exist a group of vertices in G_0 that already have those rights, and they are connected to p by a particular form of path. We state a theorem that explains how p and q should be connected in order for p to obtain rights to q. This theorem is similar to the one stated in [SNY81a].

5.3.2.1 Statement of the condition

For a node in a protection graph to obtain rights to another, it is sufficient and necessary that there exist a path between the two nodes with a structure defined by the theorem below:

Theorem 5.2: Let G_0 be a general graph, $p, q \in G_0$ and α in 2^R, Can-Share(α, p, q, G_0) is true if and only if the following three conditions hold simultaneously:

1- $\exists \alpha_i \in 2^R, s_i \in G_0, i = 1 \ldots n$, such that $s_i[*, \alpha_i]q \wedge \bigcup_i \alpha_i = \alpha$

2- $\exists p'_i \in S_0, i = 1 \ldots n$, such that $p'_i = p \vee p'_i$ is connected to p by an initial span.
$\exists s'_i \in S_0, i = 1 \ldots n$, such that $s'_i = s \vee s'_i$ is connected to s by a terminal span.

3- There exist islands I_{ij} in G_0, $1 \le i \le n$, $1 \le j \le m$, such that $p'_i \in I_{i1}, s'_i \in I_{im}$ and there is a bridge between I_{ij} and $I_{i,j+1}$.

The reason why we need a set of s_i is because α can include different rights that can be owned by distinct vertices. Whether or not p and the s_i are subjects will make a difference in the kind of path that has to exist between them. If p, for example, is an object we will need to have a subject obtain the rights and then grant them to p.

For atomic rights the theorem above can be re-stated more simply:

Theorem 5.3: Let G_0 be a general graph, $p, q \in G_0$ and α in R, Can-Share(α, p, q, G_0) is true if and only if the following three conditions hold simultaneously:

1- $\exists s \in G_0, st, s[\ldots, \alpha]q$

2- $\exists p' \in S_0, st, p' = p \vee p'$ is connected to p by an initial span.
$\exists s' \in S_0, st, s' = s \vee s'$ is connected to s by a terminal span.

3- There exist islands I_j in G_0, $1 \le j \le m$, such that $p' \in I_1, s' \in I_m$ and there is a bridge between I_j and I_{j+1}.

In the next two sections we will consider this last statement of the theorem. The more general case can be deduced from the following lemma.

Lemma 5.3: Can-Share(α, p, q, G_0) if and only if Can-Share(α_i, p, q, G_0) for all i, where $\bigcup \alpha_i = \alpha$.

Proof: Sufficiency: trivial since $\alpha_i \subset \alpha$.

Necessity: comes from the fact that no restrict rule is used for p to actually obtain

(α_i to q).

5.3.2.2 Sufficiency

Assuming s, p', s' that satisfy the conditions stated in the theorem, we will prove that Can-Share(α, p, q, G_0) is true. For this, we first establish two preliminary results in the form of a lemma and a corollary.

The lemma shows how a right can be transferred over a bridge. The corollary shows how a right can be transferred from island to island. We will deduce from that that p' can acquire the rights that s' has. We will then only have to show how s' can obtain rights from s and how p' can give its rights to p.

Lemma 5.4: Let $po_1...o_ns$ be a bridge between islands I and J,
s[*, α]q in G => $\exists G', G \rightarrow G'$ and p[*, α]q

Proof: $po_1...o_ns$ is a bridge implies that p and s are subjects and there are four possible cases to study. The bridge can be in L_0, L_1, L_2 or L_3.

Case 1: the bridge is in $L_0 = (\vec{i_t} + \vec{t})^*$

In this case, p can take successively from the objects along the path to finally obtain either take or indirect-take rights to s. this results in either p taking the α rights to q from s or s creating an object through which p is going to indirect to take the α rights to q.

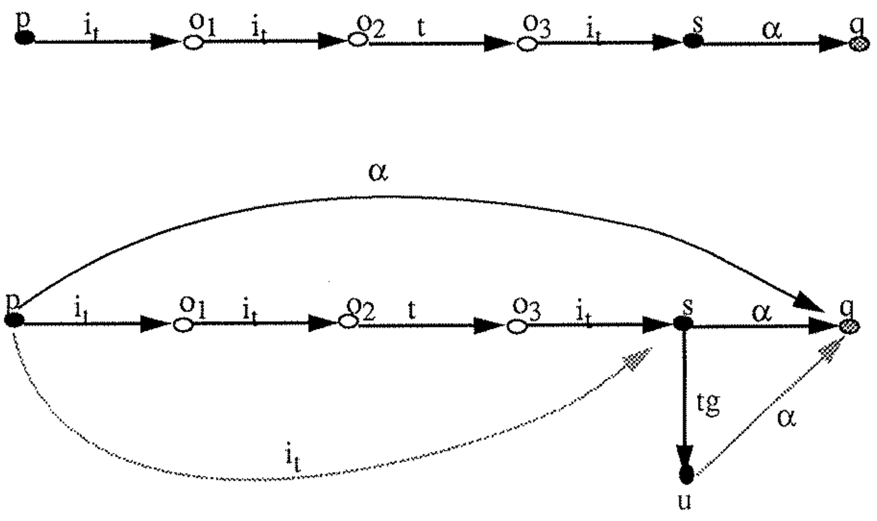


Figure 5.3

In the example above,
s creates u
s grants (α to q) to u
p takes (i_t to s) from o_3 using the chain of indirection $po_1o_2o_3$
p takes (α to q) from u using the chain of indirection psu

Case 2: the bridge is in $L_1 = (\overleftarrow{t} + \overleftarrow{i_t})^*$

Although it may seem that s can only take from p, it is actually possible for s to *grant* rights to p. Subject p simply has to create an object to which s is going to grant rights and from which p can take rights. If s can only indirect-take from p then p has to create an object u with take rights to it so that now s can take from u. Subject u can, in its turn, create an object w to which s will grant rights.

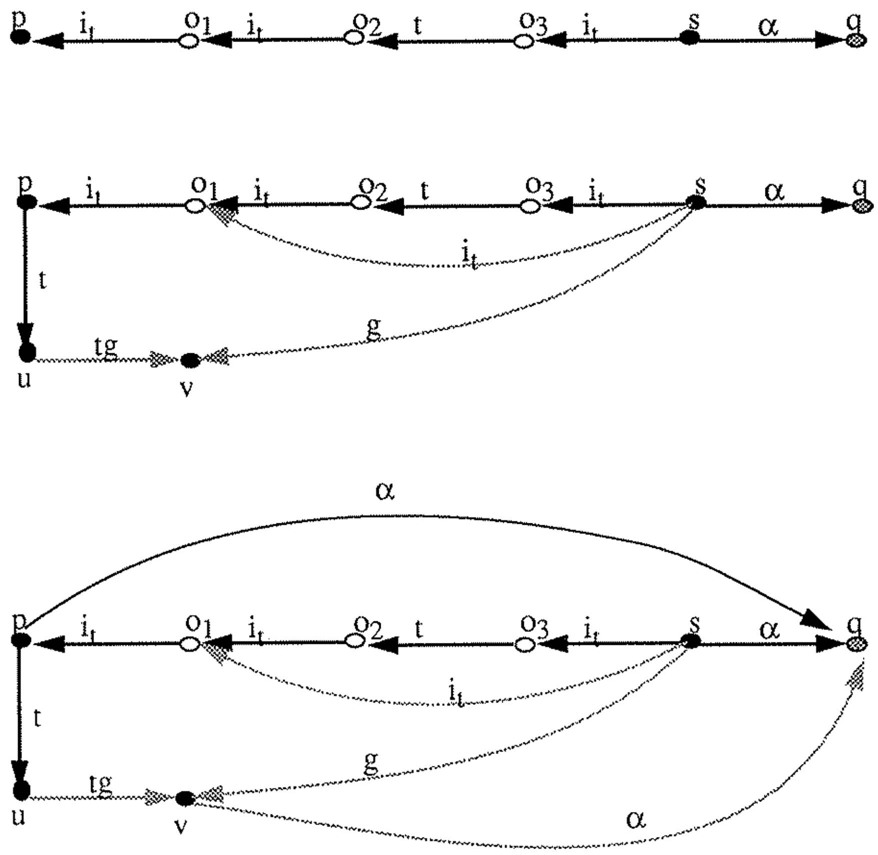


Figure 5.4

Let us see how this is done through an example (fig 5.4):

p creates u

u creates v

s takes (i_t to o_1) from o_3 using the chain of indirection so_3o_2

s takes (g to v) from u using the chain of indirection so_1pu

s grants (α to q) to v.

u takes (α to q) from v

p takes (α to q) from u

Case 3: the bridge is in $L_2 = (\overrightarrow{i_t}^* \; \overrightarrow{t} \;)^* \; \overrightarrow{i_g}^* \overrightarrow{g} \; (\overleftarrow{i_t}^* + \overleftarrow{i_g}^* \;)(\; \overleftarrow{t} \quad \overleftarrow{i_t}^* \;)^*$

This case is more complicated. p is going to create an object u and grant g rights to it to an object along the bridge. s can then take that right and grant to u the α rights to q. p's only remaining task is to take those rights from u. Notice that u has to be a subject if s[*, $\bar{c}$]q. Otherwise s wouldn't be able to grant (α to q) to u. If u is subject, this can be done under the condition that the arc created is a formal one: u[formal, $\bar{c}$α].

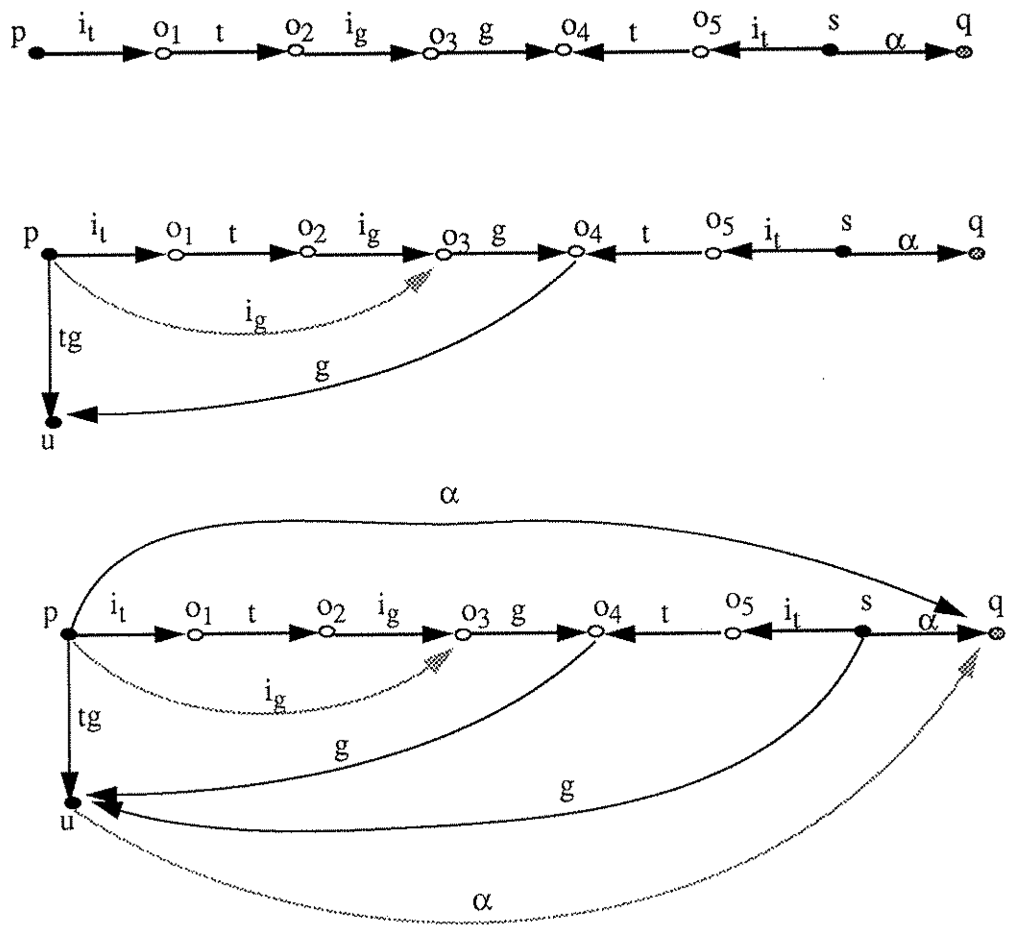


Figure 5.5

In the example shown in fig 5.5,

p creates u

p takes (i_g to o_3) from o_2 using the chain of indirection po_1o_2

p grants (g to u) to o_4 using the chain of indirection po_3o_4

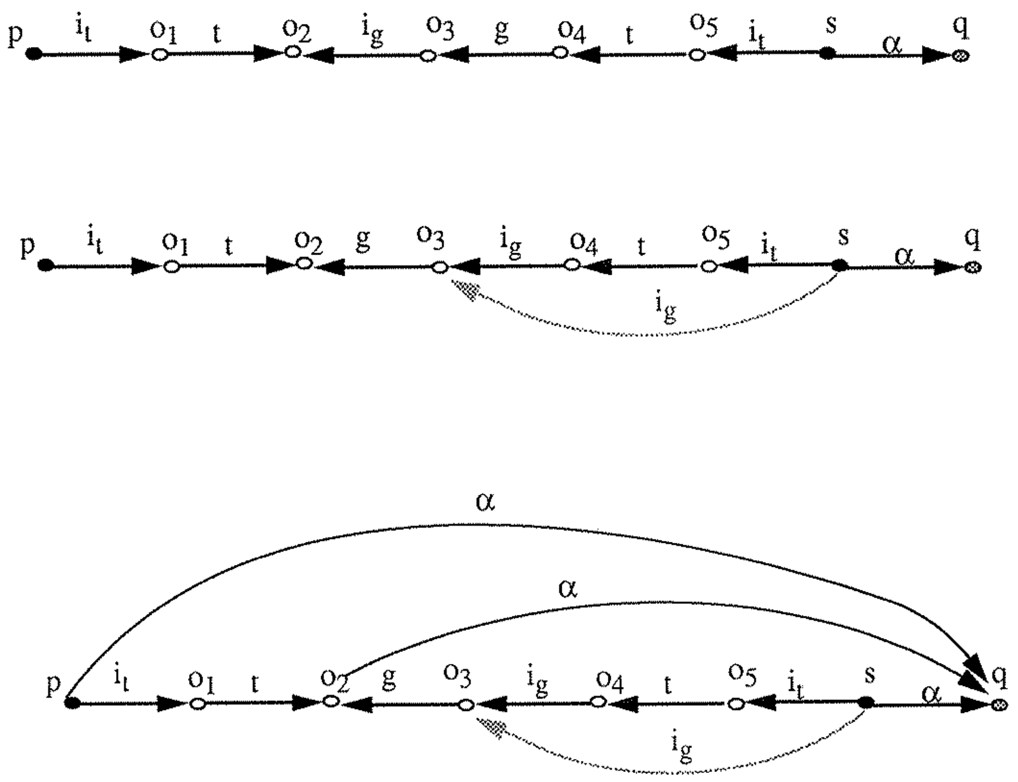
s takes (g to u) from o_4 using the chain of indirection so_5o_4

s grants (α to q) to u.

p takes (α to q) from u

<u>Case 4</u>: the bridge is in $L_3 = (\overrightarrow{i_t}^* \; \overrightarrow{t}\,)^*(\overrightarrow{i_t}^* + \overrightarrow{i_g}^*)\,\overleftarrow{g}\;\overleftarrow{i_g}^*(\,\overleftarrow{t} \;\; \overleftarrow{i_t}^*)^*$

Intuitively, s is will take the ability to grant to an object to which p can obtain take rights. This can only be done if s[*, c]q.



<u>Figure 5.6</u>

In the example above,
s takes (i_g to o_3) from o_4 using the chain of indirection so_5o_4
s grants (α to q) to o_2 using the chain of indirection so_3o_2
p takes (α to q) from o_2 using the chain of indirection po_1o_2

The case analysis has shown that if $xo_1...o_ny$ is a bridge then x can obtain the rights that y has. The arc acquired by x will be formal or local however, unless the arc originated by y had the copy right. Next in corollary 5.2, we show that if p is in I and s in J and if there is a bridge between I and J then p can obtain the rights that s has.

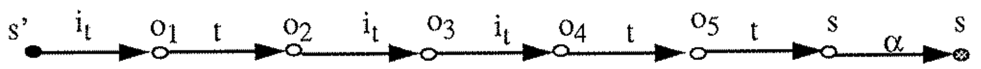
Corollary 5.2: Let p be in an island I and s in an island J in G_0. If s[*, α]q and there is a bridge between I and J then $\exists G, G_0 \rightarrow G$ and, in G, p[*, α]q.

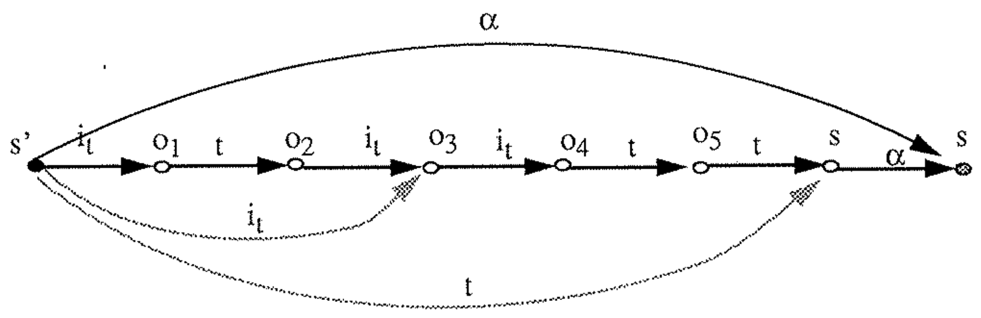
Proof: If I and J are connected by a bridge then there exist x in I and y in J such that x and y are the two ends of a bridge. s and y are two elements of a same island J which can be considered as a subject-only subgraph of G_0. Therefore we can apply the results of section 5.2 on page 30 and there exist a graph G' such that G_0->G' and y[*, α]q. Moreover since x and y are the two ends of a bridge, there exists G" such that G'->G" and x[*, α]q. x and p being two elements of the same island I we deduce that there exist a graph G such that G"->G and in which p[*, α]q.

Proof of the sufficiency for theorem 5.3:

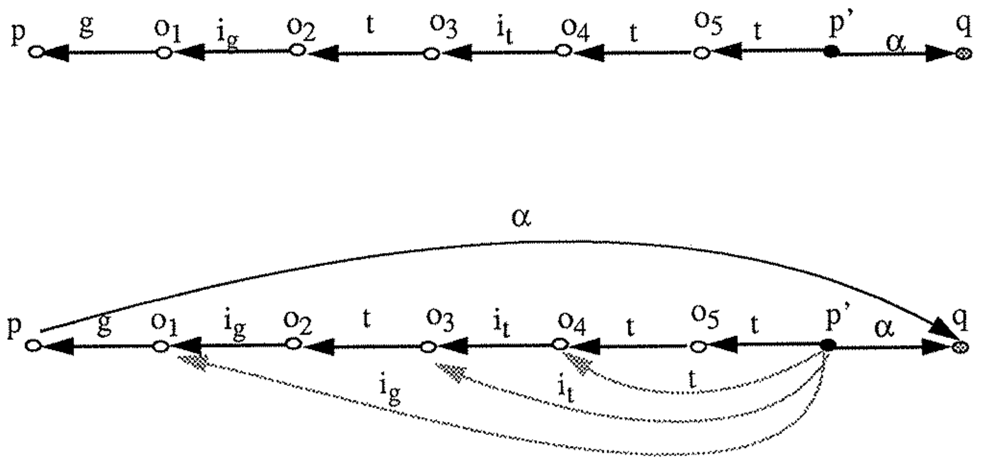
Condition 3 in the theorem states that there is a chain of islands (I_1, ..., I_m) connected by bridges, such that p' is in I_1 and s' in I_m. As a consequence of the corollary, the rights that s' has can "propagate" from island to island (I_m -> I_{m-1} -> ... -> I_1) to p'. One can actually think of an island as a single subject node since any right that a vertex in an island has can be obtained by all other vertices in the same island.

Let us show now how s' can obtain rights from s and p from p'.





s' terminally spans to s: s' can take successively take rights to objects along the path that connects it to s until it finally obtains take right to s. Notice that if s is a subject then the path between s and s' is a bridge and we know already that rights can be transferred along a bridge. The interesting case is when s is an object.



p' initially spans to p: again here, if p was a subject, the path between them would be a bridge and we wouldn't need any subject to initially span on p. If p is an object, p' has to be able to grant rights to p.

In conclusion, we see that if the three conditions in the theorem hold then it is possible for p to get the α right to q and thus Can-Share(α, p, q, G_0) is true.

5.3.2.3 Necessity

Let G_0 be a graph and p and q two vertices in that graph. Let α be an atomic right in R. Let us assume also that Can-Share(α, p, q, G_0) is true. We want to show that there must be a path in G_0 between p and q such that conditions 1 through 3 in theorem 5.2 hold.

Suppose there is no such path in G_0. Because Can-Share(α, p, q, G_0) is true, there exists a graph G derived from G_0 in which p[*, α]q. The arc [*, α] between p and q constitutes a path in G that satisfies the conditions of theorem 5.2. To show this last statement, we consider two cases (illustrated in figure 5.7): p is either a subject or an object.

If p is a subject, we choose s=p'=s'=p. We have that s[*, α]q and the path between p' and s' is in the language defined by the bridges since they include the empty string.

If p is an object and p[*, $\overline{\alpha}$]q in G_0, then there has to be a subject p' with α right to q and who can grant rights to p. This is because we know that in G p[*, α]q, which means that p has to obtain this right somehow (being an object, p cannot take any right). We choose s=s'=p'. p' initially spans to p and all the conditions are satisfied.

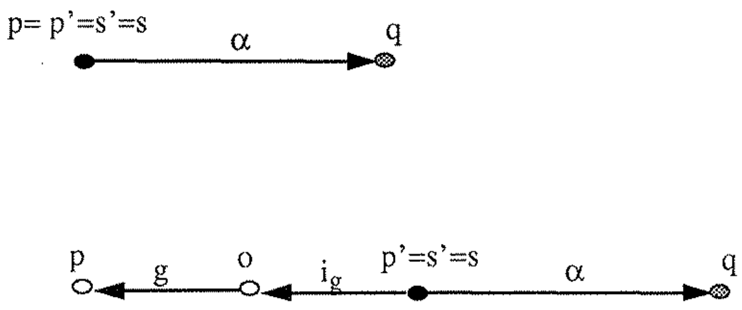


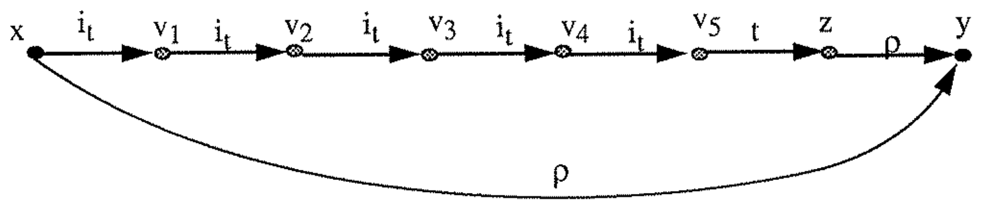
Figure 5.7

Let G_0, G_1, ..., G_n be graphs such that G_0 -> G_1 -> ... -> G_n and G_n=G, where in G p[*, α]q. There exist such graphs since Can-Share(α, p, q, G_0) is true. Let P(H) be the proposition that the three condition hold in graph H.

By assumption, P(G_0) is false, whereas, in both the case where p is an object and p is a subject, P(G) is true (see figure 5.7). Now if we assume that no restrict rule is used, then it is clear that if P(i) is true then $\forall j \geq i, P(G_j)$ is true.

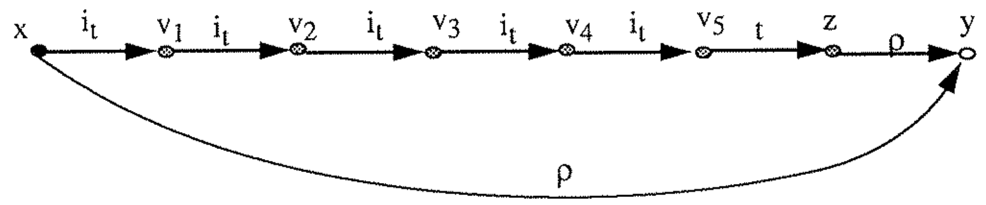
Therefore $\exists i, \forall j \leq i, P(G_j)$ is false and $\forall j > i, P(G_j)$ is true. In particular, P(G_i) is false whereas P(G_{i+1}) is true. Let us consider the path in G_{i+1} such that the conditions in the theorem hold. One and only one of the edges along that path doesn't exist in G_i. Let x and y be the vertices connected by that edge e. There are different possibilities for e to be created. Because all of them imply that P(G_i) is true, there is a contradiction. Therefore P(G_0) is true.

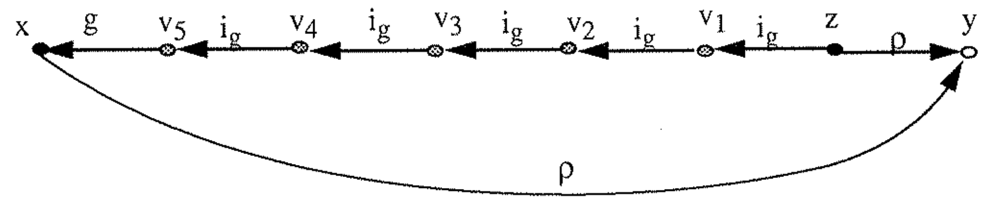
Case 1: x and y are subjects which means that in G_{i+1}, x and y are in the same island.



Let z be a vertex that has ρ right to y: either x can take from z or z can grant to x. The only way x can take from z is when x initially spans to z. Now, any word in the language defining an initial span is also in the one defining the bridges. This remark, added to the fact that $\overrightarrow{i_t}^{*}\ \overrightarrow{t}\ \rho$ is a valid bridge for all ρ's, shows that $xv_1v_2...v_nzy$ is a valid path, which implies that $P(G_i)$ is true.

<u>Case 2</u>: x is a subject and y an object => in G_{i+1}, e is the first element of a bridge, an initial span or a terminal span.



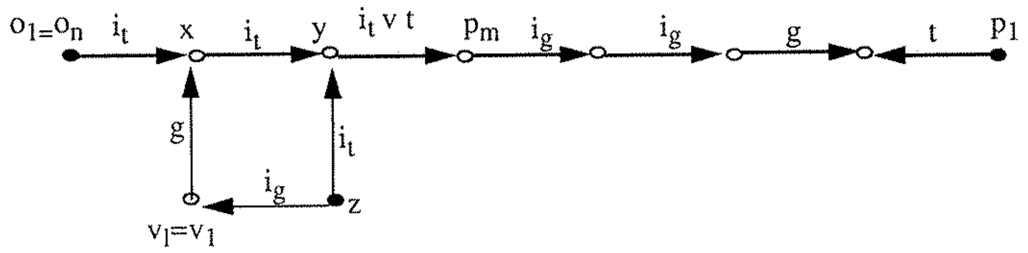


There are two possibilities: x, as a subject, can take rights or z can grant rights to x.

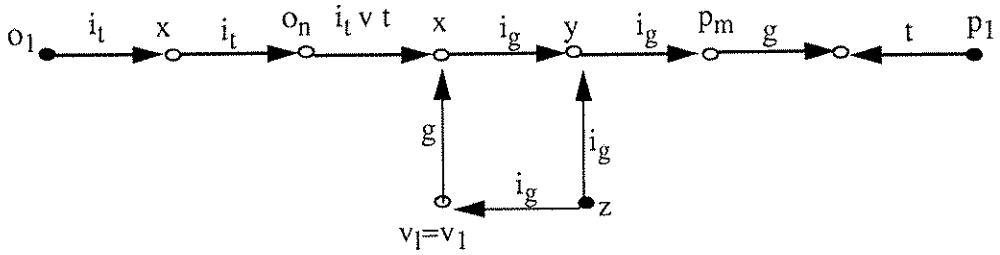
In the first figure, let v be the last subject along the path $xv_1...v_n$, where v_n=z. $xv_1...v$ is a succession of subjects connected by paths in $(t + i_t)^*$. These are legitimate bridges. On the other hand, v...y is going to replace xy in G_{i+1} which is equivalent to inserting a pattern of the form $(t + i_t)^*$ at the beginning of a bridge. This preserves the language defining the bridges.

In the second, z has to be a subject and again $xv_n...v_1z$ is a legitimate succession of islands and bridges. The path in G_{i+1} initiated at zy will be the same as the one in G_i initiated at xy.

<u>Case 3</u>: x and y are objects => x and y are two consecutive elements of the same bridge in G_{i+1} and z has to be a subject with take rights to x or y. This is the more complicated case. Again, we are going to show that if $o_1...o_nxyp_m...p_1$ is that bridge then $o_1...o_nx$-$v_l...v_1z$ is a bridge and $zyp_m...p_1$ is a bridge. Because z is going to grant x rights to y, there is an edge between z and y in G_i. This edge is the same as the one between x and y in G_{i+1}. We draw the different figures for what that edge can be.



Here, $o_1...o_nxv_l...v_1z$ is in the language defined by:
$zyp_m...p_1$ is in the language defined by:
Both of them are legitimate bridges.



The same applies here for $o_1...o_nxv_l...v_1z$ in the language defined by:
and $zyp_m...p_1$ in the language defined by:

The other possibilities can be obtained by symmetry.

So basically, we have shown that if i is the first integer greater than 0 for which $P(G_i)$ holds then $P(G_{i-1})$ holds too. This contradicts the definition of i unless i is equal to 0. As a result, if Can-Share(α, p, q, G_0) is true then the three conditions mentioned in the theorem have to hold.

5.4 Example

Let us illustrate the use of the Can-Share predicate through an example we take from the literature: the “reference monitor”. Consider a reference monitor RM_1 that works as shown in figure 5.8. $x_1 \ldots x_n$ are client subjects to which RM_1 has grant rights. $y_1 \ldots y_m$ are objects to which RM_1 has rights. These objects are resources that $x_1 \ldots x_n$ share.When a client x wants to access an object y, RM_1 decides whether or not to grant the access to y. For example, the y_i's could be pages. In that case. the α_i's would be rights {r, w}.

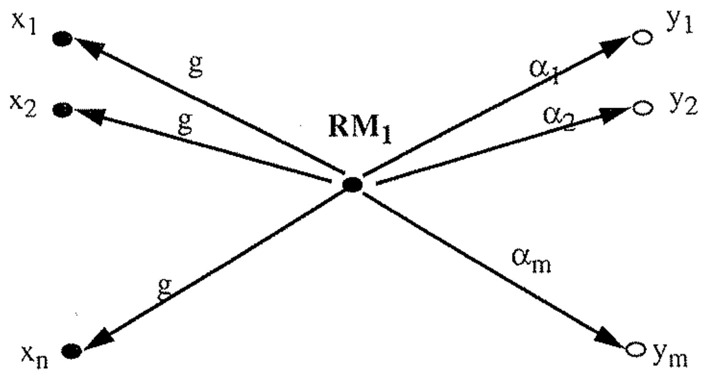


Figure 5.8

If we examine the graph G_0 in figure 5.8, we see that $x_1 \ldots x_n$ and RM_1 form an island. Thus, corollary 5.1 applies and any subject in this island can obtain the rights that RM_1 has. More formally, Can-Share(α_j, x_i, y_j, G_0) for all i and j. This simply says that each subject has the potential to access any object, which is desirable. However, this access is controlled. In fact, RM_1 is required to cooperate for this transfer of rights to be performed. The reason is, if we consider RM_1 to be an object, there are no paths in G_0 that satisfy the conditions of theorem 5.3. Similarly, no subject can obtain rights from another subject if RM_1 is an object. The strength of the theorem is that we do not need to determine all possible paths in derivation graphs. It is sufficient to look at the initial graph to establish whether or not a node can have access to another. No matter how many nodes are created and how may rights transferred, if the initial graph does not contain an access-flow path, no graph derived from that initial graph will ever contain such a path.

Before granting a right to a client, the reference monitor can determine whether the resulting graph will be "safe". In figure 5.9, $\alpha_1=\{t, g\}$ and RM_1 granted (α_1 to y_1) to x_1. Subject x_1 now wants to obtain g rights to y_1. If the request is granted, there will be a access-flow path between x_1 and x_2 that does not involve RM_1. This would allow x_1 and x_2 to share resources. If this situation is forbidden by the policy, RM_1 should refuse the request.

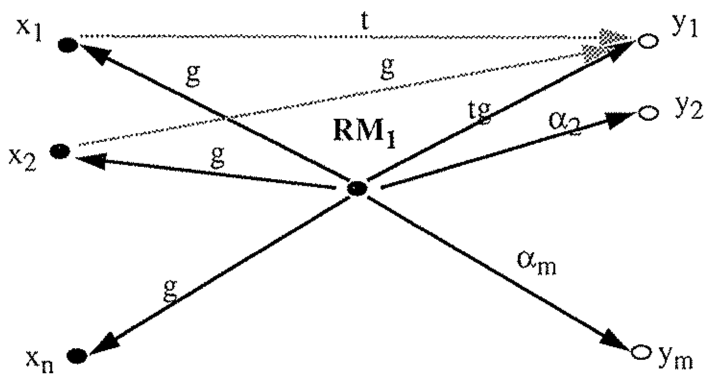


Figure 5.9

An interesting point to note is that RM_1 has some control over the rights it grants to a subject x through the use of the c rights (cf section 2.2.2 on page 12 and section 2.2.6 on page 15). Depending on whether or not the rights RM_1 grants to x contain the c right, x will have an arc of static or formal nature. x in turn will not be able to store these rights in an object y unless they contain the c right. x could however pass the rights on to another subject with the restriction that the resulting arc should be formal. These considerations have an impact on the decision to be made the reference monitor.

One problem in the implementation of RM_1 is that once a right to a node is granted, it is difficult to have it back. The different natures of the arcs can be used to address this problem as long as these natures are associated with some notion of permanence. If the designer interprets formal arcs as arcs for which there is some confidence that they are temporary, RM_1 can grant such arcs when necessary. Another solution would be to have the reference monitor consist of two parts: a subject RM_1 and an object RM_2 (figure 5.10).

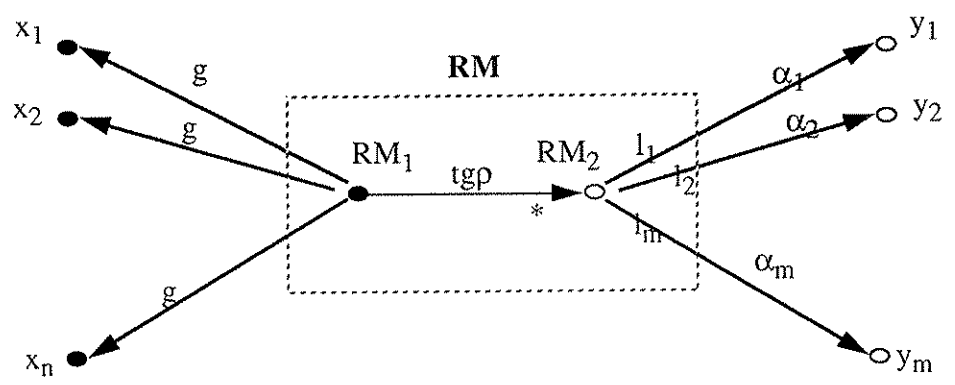


Figure 5.10

There is an edge between RM_1 and RM_2 labelled with rights ρ and in-value *. ρ is the set of all possible indirect rights and * means that the arc can take on any in-value. When RM_1 is invoked, it will grant an arc with the in-value that it chooses. If, for example, x_1 wants (α_1 to y_1) it will obtain an arc for RM_2 with rights $i_{\alpha 1}$ and in-value l_1.

The advantage of this scheme is that revocation can take place efficiently. When RM wants to revoke the access that x_1 has to y_1, RM_2 changes the out-value of the arc it has to y_2. In fact, RM_1 initiates the take, grant and delete rules so that $RM_2(l_1, *, *, \alpha_1)y_1$ becomes $RM_2(l'_1, *, *, \alpha_1)y_1$.

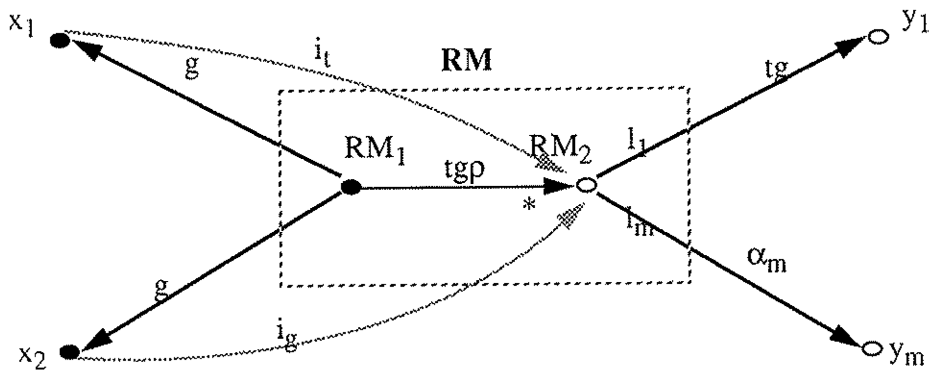


Figure 5.11

Again, because RM_1 is a subject, x_1 ... x_n can obtain all the rights that RM_1 has, which gives them the potential to access all objects in the graph. Also, two subjects can still share rights the way they could with a single-subject reference monitor. In figure 5.11, x_1 obtained indirect-take rights to y_1 and x_2 indirect-grant rights to y_1. Although $x_1RM_2x_2$ does not constitute a valid access-flow path, $x_1RM_2y_1RM_2x_2$ has an associated word ($\overrightarrow{i_t}\ \overrightarrow{t}\ \overleftarrow{g}\ \overleftarrow{i_g}$) which is in the language defining the bridges. Hence, x_1 and x_2 can share rights. This could be avoided provided that the in- and out-values are properly chosen and not modified by RM. Therefore, some degree of trust has to given to the reference monitor. The implications will be addressed in part in chapter 7.

To conclude, we see how the theorems in section 5.0 give us the tools to analyze a protection graph in the model ($\mathcal{G}_M$, $\mathcal{R}_M$) and predict whether a given transfer of right is possible. In the next chapter, we give an algorithm for verifying the truth of the conditions stated in those theorems.

Chapter 6 Linear-time algorithm

In this section we will prove that, given a labelled directed graph, it is possible to search in linear time for a path whose labels form a word in a given regular language. The result obtained can therefore be applied to different problems.

In particular, it proves the statement made in [BIS79] that it is possible to find bridges in the take-grant model in linear-time since these bridges are defined by regular expressions.

We may therefore extend the take-grant model to include other rights without having to prove the linearity of the search every time. In fact, any problem that can be stated in terms of a search for a path described by a word in a regular expression can be solved in linear time. The Can-Share predicate described in section Chapter 5 on page 29 falls into this category.

6.1 Algorithm

Let G be a graph where the edges are labelled with a letter from an alphabet Σ. We will establish that it is possible to search for a path between two end points of G with associated word in a given regular language L over Σ in linear time. The implications of this result are discussed in the next section.

6.1.1 Equivalence with a DFA

Before actually going through the search phase, we use an important result from Automata Theory and Languages: the equivalence of finite automata and regular expressions. Let L be a regular language, there is a Deterministic Finite Automaton $M=(Q, \Sigma, \delta_0, s_0, F)$ that defines the language L. That is, starting at the initial state, M will reach a final state only when reading words in the language L. For M,

Q is the set of states of the machine

Σ is the alphabet

δ is the state transition function

s_0 is the initial state
F is the set of final states

We apply this result to the language L defining the paths we are searching for. If two vertices are connected, the path between them can be written as a sequence of edges labelled with letters from the alphabet Σ. The word formed by the concatenation of those letters is tested to be in L. Because L is equivalent to a DFA, M, this test can be performed by M.

To perform the test, we associate each word w in L with a state s of M in the following manner: $s = \delta(s_0, w)$. s is the state M will be in after reading the string w starting in the initial state s_0. We use this mapping to test whether w is in L: $w \in L \Leftrightarrow s \in F$.

6.1.2 Description of the algorithm

Essentially, the algorithm consists of a depth-first search of a graph G. Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we reach the destination q successfully or until all vertices that are accessible from the source p are discovered.

If we do not take precautions, a vertex could be visited indefinitely. To avoid this situation, vertices are colored, where each color is associated with a state of M. We keep with each node v a record C[v] of all the different ways it has been colored. Every time a node is discovered, M is in a state s_i and we check whether color i is in C[v]. If it is, we backtrack because we have already explored the graph from v in state q_i. If it is not we add it to C[v] and continue the search. Initially, all C[v]'s are set to nil.

From this coloring we derive the linearity of the algorithm. In fact, a DFA has a finite number of states; therefore by stamping each vertex we reach with the state (color) at which we arrive and not "visiting" a vertex twice in the same state, each node in the graph will not be "visited" more than $|Q|$ times, where $|Q|$ is the number of states.

To understand better the function of the coloring, notice that at the end of the search, it is possible (and actually probable) that not all vertices have been fully colored. First of all, if we stop as soon as we encounter q in a final state, an entire subgraph of G might not have been explored yet and therefore not colored at all. Even if we did explore the whole graph (q was not reachable or we wanted all the bridges from p to q), nothing guarantees that a particular vertex can be reached from p in a given state. The coloring should not be interpreted as a way to determine whether a graph has been completely explored. Rather, it is simply a way to avoid visiting a node from which we know in advance the destination would not be reachable.

Following is an implementation of the algorithm described above:

```
explore(α, p, q, G)
For each vertex u in G
    do C[u] <- NIL
C[p] <- 0
Visit(p)
If C[q] contains a color i such that $s_i$ is in F
     Then return TRUE
     Else return FALSE

Procedure Visit(u)
For each v adjacent to u
    do determine state $s_i$ of M
    If v <> q OR $s_i$ is not in F
        Then if C[v] does not contain i
                Then C[v] <- i
                     Visit(v)
```

This algorithm actually finds all possible paths between p and q that are in L and simply returns TRUE if the number of paths is not zero (q has been colored with a color associated with a final state). It would be possible to record those paths as they are found, which would provide us with a way to determine all paths between p and q that are in the language.

There are two important aspects of the algorithm that deserve mention:

- first, observe that the paths considered in the algorithm are “logical” paths in the sense that they may contain several instances of the same node. This means that the number of paths between two nodes is either 0 (the two nodes are not connected) or infinity (the two nodes are connected).

For example, in the case of the take-grant model, an edge labelled with take rights is associated with words in the language $(\vec{t}\ \overleftarrow{t}\ )^*\vec{t}$, which is infinite. It is necessary to be able to consider these kind of paths in the algorithm since nothing proves that they will not result in a word in the language.

- second, cycles have to be allowed too. In fact, there are graphs for which the only paths that generate words in the language contain cycles. Figure 6.0 gives such a graph in the case of the take-grant model. xzy is not a valid bridge whereas xzuvzy is. It may also be necessary to loop several times in the cycle in order to reach a final state.

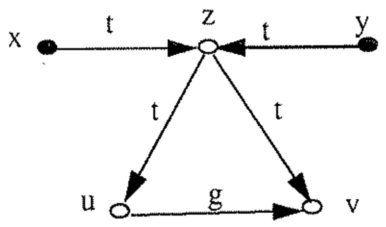


Figure 6.0

The algorithm takes both cases into account. When a vertex is reached, all its neighbors are explored including the one from which we come. This raises the possibility of an eventual infinite loop: it is avoided in the algorithm described above by coloring the arcs. No node will be visited more than the number of states of the equivalent DFA. Therefore, all possible paths are considered even though they are not necessarily explored which preserves the linearity of the algorithm.

6.1.3 Evaluation

Because of the condition on the coloring of the nodes, each node is visited at most |Q| times. Each time, at most Δ edges adjacent edges are traversed, where Δ is the maximum degree of G.

The cost of finding a DFA for a given regular language is a constant that does not depend on the size of the graph and, as a consequence, is not included.

Overall, the cost of the algorithm is O(|V| + |E|), where E is the set of edges of G and V is the set of vertices of G.

6.1.4 Example

In order to clarify the preceding, we show here how to apply the algorithm to the search of a simplified version of the bridges defined in section 5.3.1 on page 36:

$B = B_0 \cup B_1 \cup B_2 \cup B_3$, where

$B_0 = \overrightarrow{t}^*$

$B_1 = \overleftarrow{t}^*$

$B_2 = \overrightarrow{t}^* \overrightarrow{g} \overleftarrow{t}^*$

$B_3 = \overrightarrow{t}^* \overrightarrow{g} \overleftarrow{t}^*$

Given a graph G_0, p and q in G_0, we look for the existence of a path between p and s with associated word in the language B. For this example we will not worry about the fact that all nodes along the path have to be objects (except for p and s) in order for the path to be a bridge. We simply want to illustrate how to conduct the search for a path of a form given by a regular expression.

Since it is easier to derive an NFA (Non-deterministic Finite Automaton) from a regular language, the first step is to write an NFA describing B. In the next step, wee will convert the NFA to a DFA so that to apply the algorithm. Let us consider the NFA defined by $(Q, \Sigma, \delta, q_0, F)$, where:

$Q = \{q_0, q_1, q_2\}$ is the set of states

$\Sigma = \{t, t, g, g\}$ is the alphabet

δ is the transition function defined by the figure below (fig)

q_0 is the initial state

$F = \{q_1, q_2\}$ is the set of final states

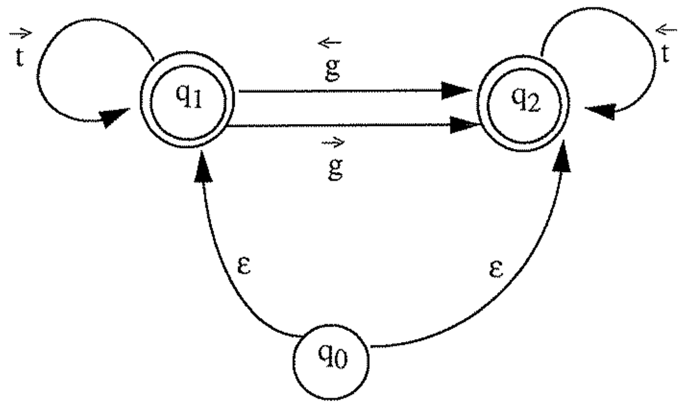


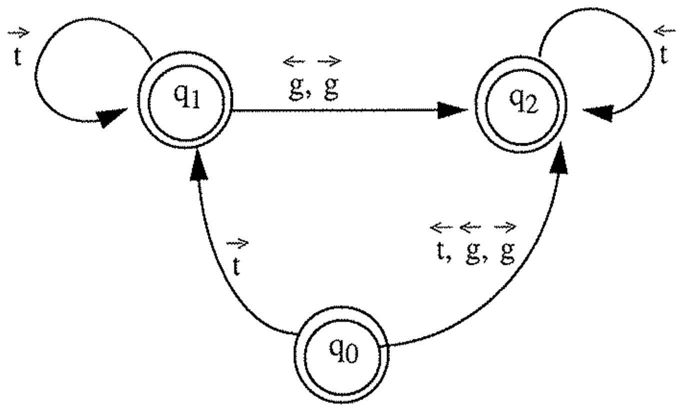
Figure 6.1

Clearly, B can be characterized by this NFA. We also know that there is an equivalence between NFA's with and without ε-moves and we have a method to derive one from the other. Let us apply it to derive M', NFA without ε-moves.

States \ Inputs	$\overrightarrow{t}$	$\overleftarrow{t}$	$\overrightarrow{g}$	$\overleftarrow{g}$
q_0	$\{q_1\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
q_1	$\{q_1\}$	$\varnothing$	$\{q_2\}$	$\{q_2\}$
q_2	$\varnothing$	$\{q_2\}$	$\varnothing$	$\varnothing$

Figure 6.2

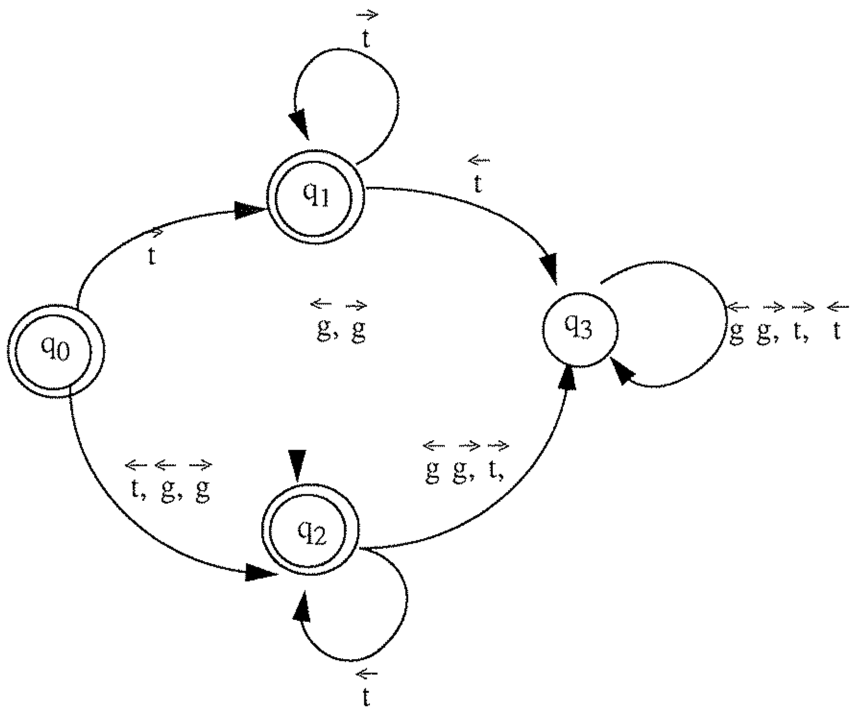
Because in M, ε-closure(q_0) = $\{q_1, q_2\}$ contains states in F, q_0 has to be a final state in M'. The transition function for M' is shown in figure.



$\overrightarrow{t}$
$\overleftarrow{g}, \overrightarrow{g}$
$\overleftarrow{t}$
q_1
q_2
$\overrightarrow{t}$
$\overleftarrow{t}, \overleftarrow{g}, \overrightarrow{g}$
q_0

Figure 6.3

Next, we obtain the equivalent DFA M'' by adding a new state q_3 and have, for each input symbol, exactly one transition out of each state (figure). In general, there are algorithms to perform these tasks. But this has to be done only once for a given regular expression (or basically for a given type of search). Every time we want to verify whether there exists a bridge between two vertices we can use the same description M'' of B.



$\overrightarrow{t}$
q_1
$\overleftarrow{t}$
$\overrightarrow{t}$
$\overleftarrow{g}, \overrightarrow{g}$
q_0
q_3
$\overleftarrow{g}\ \overrightarrow{g}, \overrightarrow{t}, \overleftarrow{t}$
$\overleftarrow{t}, \overleftarrow{g}, \overrightarrow{g}$
$\overleftarrow{g}\ \overrightarrow{g}, \overrightarrow{t},$
q_2
$\overleftarrow{t}$

Figure 6.4

Let us see how the search can be conducted through an example. Let G_0 be the graph shown in figure 6.5. We perform a depth first search of G_0, starting from p. Initially, we are in state q_0. We choose one of the edges intersecting p. po_1 corresponds to the letter $\overrightarrow{t}$ in Σ. We end up being in the state q_1. Hence, we store color 1 in the stack $C[o_1]$ associated with o_1.

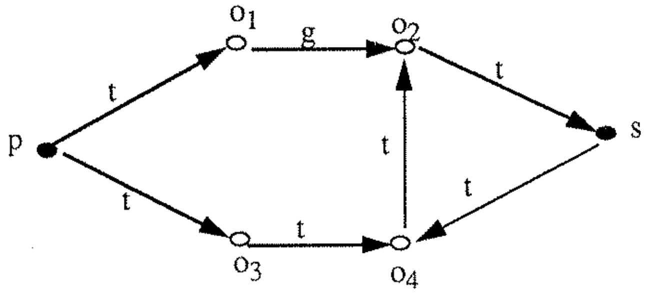


Figure 6.5

Although q_1 is a final state, we did not reach s yet. Next, we do not have any choice but go to o_2 which brings us to state q_2. q_1 is then stored in S_2. From there, there are two possibilities. Suppose we choose the path that leads to s. We did so by reading the letter $\overrightarrow{t}$. We are therefore in state q_3 which is not a final state. We now go back to o_2 and proceed to o_4. As far as M' is concerned, we stayed in state q_2. From o_4, we can reach s and again stay in state q_2 which is a final state. $po_1o_2o_4s$ is thus a bridge.

Suppose that, from o_4, we had chosen to go to o_3 and then p. We would have reached p with state q_2. If we could have gone through another cycle (not possible in this case because q_3 is the only state accessible from p at this point) and reached p with same state q_2, we would not have explored the path we were in any longer. This prevents a situation of infinite loop ($\overrightarrow{t}^*$ and $\overleftarrow{t}^*$ in this case) from happening.

Suppose now we want to find another bridge between p and q. We are back to p and we go to o_3, o_4 and o_2. At this point, we are in state q_1. o_2 had previously only been marked with state q_2. We can therefore go on and reach s from o_2. We are now in a final state, q_2. $po_3o_4o_2s$ is a bridge.

The following diagram shows the steps described above.

path	word	state	action
po_1		q_1	color 1 stored in $C[o_1]$
po_1o_2		q_2	color 2 stored in $C[o_2]$
po_1o_2s		q_2	back to o_2
$po_1o_2o_4$		q_2	color 2 stored in $C[o_4]$
$po_1o_2o_4s$		q_2	$po_1o_2o_4s$ is a bridge
po_3		q_1	color 1 stored in $C[o_3]$
po_3o_4		q_1	color 1 stored in $C[o_4]$
$po_3o_4o_2$		q_1	color 1 added to $C[o_2]$
$po_3o_4o_2s$		q_1	We reach s in the final state q_1

The analysis of the graph is not complete yet (other paths need to be explored) but we can see that:

1- no node will be visited more than four times.

2- we have to allow loops but we know they will not be traversed more than 4 times.

3- no edge will be visited more than four times.

Clearly, we obtain that there are three different bridges between p and q: po_1o_2s, $po_1o_2o_4s$ and $po_3o_4o_2s$.

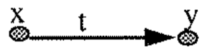
6.2 Application to the Can-Share predicate

In the context of the model described in chapter 2, what we want to prove is slightly different. Given a graph G_0, two vertices p and q, we want to find a path between p and q that can be decomposed in a succession of islands and bridges preceded by an initial span and succeeded by a terminal span.

Before applying the algorithm described above, we have to be careful. First, in G_0

edges are directed. This means that if x and y are vertices in G_0, the path between x and y is different from the one from y to x. In figure 6.1 a) x is connected to y by an edge labelled $\overrightarrow{t}$ whereas y is connected to x by an edge labelled $\overleftarrow{t}$. The direction of the edges have to be combined with the set of rights to form an alphabet we can use to conduct the search. As a consequence, the input alphabet will include { $\overrightarrow{t}$, $\overrightarrow{g}$, $\overrightarrow{i_t}$, $\overrightarrow{i_g}$, $\overleftarrow{t}$, $\overleftarrow{g}$, $\overleftarrow{i_t}$, $\overleftarrow{i_g}$ }. In figure 6.1 b) x is connected to y by a path of the form $\overrightarrow{t}\ \overleftarrow{i_g}\ \overrightarrow{i_t}\ \overrightarrow{t}$.

a)



b)

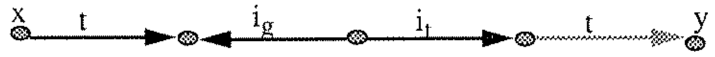
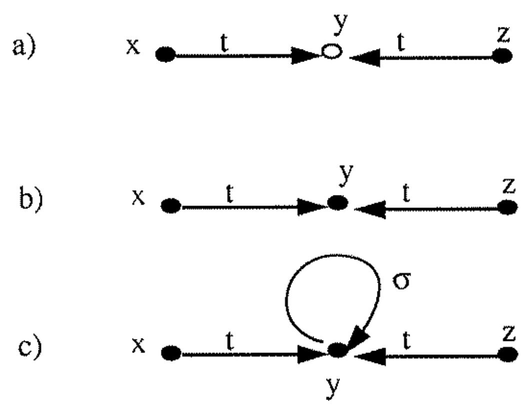


Figure 6.6

Another assumption made in the establishment of the algorithm is that the edges in the graph are labelled with single letters, which is not true for a general graph. To account for this, we construct a new graph G_0' where rights are decomposed into their atomic components. If x[*, ρ]y in G_0, where $\rho=\rho_1\rho_2...\rho_n$ and the ρ_i's are atomic, then we connect x and y in G_0' with n arcs labelled respectively with ρ_1, ρ_2, ..., ρ_n. The number of arcs of G_0' is less than (number of arcs in G_0) x (max number of atomic rights).

Recall the objective is to show we can determine the truth or falsehood of the Can-Share predicate in linear time. This can be achieved using two corollaries of the result we are going to prove in 6.2. First, it is possible to determine the existence of a bridge between two islands in linear time. Second, by applying the results of section 6.1 on page 53 to a graph G_0''' derived from G_0' we can prove or disprove the existence of an access-flow path between two vertices in G_0.

For this, and because a word associated with a path does not say anything about the nature (subject/object) of the nodes traversed by the path, we introduce a new right σ. The presence of this right translates to "a subject has just been traversed". In fact, sometimes the information carried by the right itself is relevant only if we know "who owns that right". Let us take the following example: x[*, t]y and z[*, t]y, where a and z are subjects and y is either.



In both cases (a and b), the path between x and y is: $\overrightarrow{t}\overleftarrow{t}$. If z is a subject, x can get rights from z whereas if z is an object x cannot. If we introduce a right σ we can differentiate between the two cases (a and c) by looking at the paths between x and z ($\overrightarrow{t}\overleftarrow{t}$ vs. $\overrightarrow{t}\sigma\overleftarrow{t}$).

Claim: Let L be a regular language. There is an algorithm to determine the existence of a path in L between two vertices of a graph that operates in linear time.

Proof: The algorithm is described in section 6.1 on page 53

Corollary 6.1: There is an algorithm to determine the existence of a bridge between two islands that operates in linear time.

Proof: We can derive from G_0' a graph G_0" where all the edges between two subjects have their right replaced by σ. A path in an island can therefore be described by the regular expression σ*. There is a bridge between I_1 and I_2 if given x_1 in I_1 and x_2 in I_2 there is a path between x_1 and x_2 in σ*Lσ*, where L is the language defining the bridges (section 5.3 on page 36). The rest is immediate considering that the four possible expressions defining a bridge are regular expressions and that the union and concatenation of regular expressions are still regular expressions.

Corollary 6.2: There is an algorithm to determine the existence of an access-flow path between two vertices of a general graph G_0 that operates in linear time.

Proof: We modify G_0" to be the graph G_0"' where each subject s in G_0" has now an arc pointing to itself and labelled σ. The search of an access flow path in G_0 is then equivalent to that of a path in G_0''' with associated word in the language defined by the following regular expression: $(I^R + \varepsilon)\ (\sigma^+\ (L_0 + L_1 + L_2 + L_3)\ )^*\ \sigma^+\ T$, where I, L_0, L_1, L_2, L_3, T are defined in section 5.3.1 on page 36 and I^R is the language corresponding to all reverse strings w^R, w in I.

Corollary 6.3: There is an algorithm to test for Can-Share(α, p, q, G_0) that operates in linear time.

Proof: Can-Share(α, p, q, G_0) is true if and only if there exist a path from p to q with associated word in the language defined by the following regular expression:
$(I^R + \varepsilon)\ (\sigma^+\ (L_0 + L_1 + L_2 + L_3)\ )^*\ \sigma^+\ T\ i_\alpha{}^*\ \alpha$.

Chapter 7 Trust

In chapter 5, we showed how to determine whether one node in a protection graph can obtain rights to another. To derive this result, we assumed that all subjects in the graph were conspirators. In this chapter, we consider the case where some subjects are "trusted". We first define what we call trust. Then we show how the trust of some subjects can affect the result of chapter 5.

The notion of trust is closely related to that of theft and conspiracy as introduced in [SNY81b] for the take-grant model. We briefly discuss these issues in this chapter and see how the modifications made to the model affect them.

In [SNY81b], theft occurs when a subject gains rights to a node without any other subject having to grant them. This can only happen if the thief can 'take' them. As a consequence, *Can-Steal*(α, *p*, *q*, G_0) holds if Can-Share(t, p, s, G_0) holds for some s such that s[*, α]q. Although this definition of theft seems logical, it gives rise to situations where the truth of the Can-Steal predicate is counter intuitive. If Can-Steal holds, it does not mean that no subject s owning the rights did not participate in the theft as observed in [SNY81b]. In fact, s can 'help' other subjects to have take rights to s, which is sufficient for the theft to be accomplished. This stresses the need for a formal definition of the notion of theft, as it is not clear nor intuitive what should be considered to be theft and what should be considered to be mere acquisition.

If we decide that for a subject to be trusted, it should not participate in any way in the theft then the problem becomes hard. Suppose we do not allow s to initiate any rule (s acts as an object). If p can still obtain α rights to q (Can-Share(α, p, q, G_0) holds), it means that p does not need the participation of s to obtain the rights. If on the other hand, Can-Share(α, p, q, G_0) does not hold when s is an object and does hold when s is a subject, the situation becomes delicate. s has to know what actions it should not perform to prevent the theft. Presumably any arc added by s could be that missing edge in an access-flow path between p and s (figure 7.1).

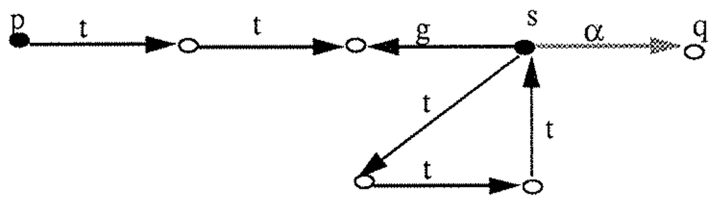


Figure 7.1

With these considerations in mind, a subject s is said to be trusted if the only way a subject p can obtain the rights that s has is to 'steal' them from s. More formally, s is trusted to preserve property P if for each rule s initiates in a graph G derived from G_0, (P is true in G => P is true in G'), where G' is obtained from G, after s initiated the rule.

This definition is the most general one. It suffers though from the fact that it requires future knowledge of the actions performed by s. Indeed, s has the potential to have p obtain the rights it needs. It would be very difficult to 'trust' s without being given an indication of what its behavior will be.

Rather, we characterize the behavior that s should not have, which in general is not easy to do. In some cases however, there is a simple way to ensure that a particular access flow will not happen given that a subject fulfills some requirements. The model ($\mathcal{G}_M$, $\mathcal{R}_M$) provides with a number of tools to make that possible.

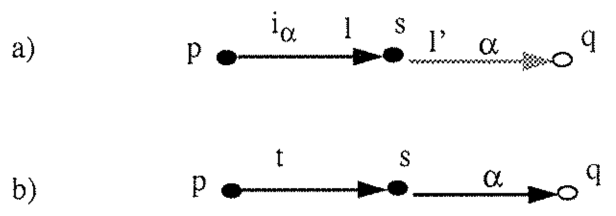


Figure 7.2

The example on figure 7.2 a illustrates how to ensure that rights will not be transferred along a given path by exploiting properties of indirection. If we trust s not to change the out-value l' of the arc s(l', *, formal, α)q and not to grant away the α rights to q, p will never obtain them. The reason is twofold:

- Since no node has take rights to s in the initial graph G_0, the same will hold for all graph derived from G_0 (no rule creates arcs to an existing node).

- Since no arc of the form (*, l', *, i_α)s exists in G_0, the same will hold for all graph derived from G_0 (no rule creates in-values for arcs referencing existing nodes).

In figure 7.2 b, we trust s not to change the nature of the arc (l', *, static, α)s and not to grant away the α rights to q. This will prevent p from ever being able to access q.

Therefore, we can see that the notion of indirection and that of nature of an arc enhances the ability to associate with a given subject some level of "trust". To establish the truth of the Can-Share predicate, however, we assume that all nodes in the protection graph conspire. Trusting some nodes in the graph would mean that a particular transfer of rights may not be possible even though the Can-Share predicate holds.

For both the graphs in figure 7.2, Can-Share(α, p, q, G_0) holds. Yet, if we trust s not to change l' (figure 7.2 a) or the nature of its arc (figure 7.2 b), p cannot obtain α rights to q. This means that under reasonable conditions, we can prevent a transfer of rights from being possible. The Can-Share predicate does not capture the formulation of those conditions: we would need to define another predicate.

Chapter 8 Conclusion

In order to model policies that enforce revocation and confinement, we introduced the notion of indirection and that of colored arcs in the take-grant model. We showed that the interesting properties that led us to base our work on the take-grant model still hold for the extended model. In particular, we proved that it is possible to determine whether rights can be obtained in a protection system represented in our model. We also propose an algorithm, operating in linear-time, that explores paths along which those rights can be transferred.

Several problems remain open. Is there a simple scheme to determine whether a subject can be trusted not to participate in a transfer of rights? We already saw that such a scheme should require a prior (partial) knowledge of the behavior of the subject to be trusted.

What are the predicates of interest? What is the set of policies that can be captured by the model? How can they be expressed in such a way that they can be verified in linear-time? We saw that if they can be expressed in terms of a search of a path with a given structure in the graph, there is a linear-time algorithm to verify them.

Another issue is to determine whether there is a way to generalize the proofs throughout this work so that they apply to different predicates or different models. It seems that there is a lot in common in the process of writing those proofs for different extensions of the take-grant model.

REFERENCES

BIS79 M. Bishop and L. Snyder.
The Transfer of Information and Authority in a Protection System.
Proceedings of the 8th Symposium on Operating Systems Principles: 45-54 (December 1979).

BIS81 M. Bishop.
Hierarchical Take-Grant Protection Systems.
Proceedings of the 7th Symposium on Operating Systems Principles: 107-123 (December 1981).

BIS84 M. Bishop
Practical Take-Grant Systems: Do They Exist?
Ph.D. Thesis, Purdue University (1984).

BIS88 M. Bishop
Theft of Information in the Take-Grant Protection Model
Technical Report PCS-TR88-137 (revised) (1988).

COG82 J. A. Coguen and J. Meceguer.
Security Policies and Security Models.
IEEE 1982

DEN76 D. Denning.
A Lattice Model of Secure Information Flow.
CACM 19(5): 236-243 (May 1976).

HRU76 M. A. Harrisson, W. L. Ruzzo and J. D. Ullman.
On Protection in Operating Systems.
CACM 19(8): 461-471 (August 1976).

JBI84 J. Biskup.

Some Variants of the Take-Grant Protectiom Model
Information Processing Letters 19(3): 151-156 (October 1984).

JLS76 A. K. Jones, R. J. Lipton and L. Snyder.
A Linear Time Algorithm for Deciding Security.
Proceedings of the 17th Annual Symposium on Foundations of Computer Science, 1976.

JON78 A. K. Jones and R. J. Lipton.
The Enforcement of Security Policies for Computation.
CSS 17(1): 35-55 (January, 1978).

LAN81 Carl E. Landweihr
Formal Models for Computer Security
Computing Surveys, Vol 13, No. 3: 247-278 (September 1981)

LIP77 R. Lipton and L. Snyder.
A Linear Time Algorithm for Deciding Subject Security.
J. ACM. 24, 3: 455-464 (July 1977).

NEE77 R. M. Needham and R. D. H. Waker
The Cambridge CAP Computer and its protection system.
Proceedings of sixth ACM symposium on Operating Systems Principles: 1-10 (November 1977).

SAN88 R. Sandhu.
The Schematic Protection Model: its Definition and Analysis for Acyclic Attenuating Schemes.
J. ACM. 35, 2: 404-432 (April 1988).

SNY77 L. Snyder.
On the Synthesis and Analysis of Protection Systems.
Proceedings of the 6th Symposium on Operating Systems Principles: 141-150 (November 1977).

SNY81a L. Snyder.

Formal Models of Capability-Based Protection Systems.
IEEE Transactions on Computers C-30, 3: 172-181 (March 1981).

SNY81b L. Snyder.
Theft and Conspiracy in the Take-Grant Protection Model.
JCSS 23, 3: 333-347 (December 1981)

WUL91a Wm. A. Wulf
A Proposal for WM Interprocess Communication
Computer Science Report No. TR-91-04 (March 1991)

WUL91b Wulf, W. and Jones, A.
WM Protection: The Base Meechanism
Technical Report
U. Virginia (March 1991)