2.3. Protection State Transitions

As processes execute operations, the state of the protection system changes. Let the initial state of the system be $\mathbf{X_0} = (\mathbf{S_0}, \mathbf{O_0}, \mathbf{A_0})$. The set of state transitions is represented as a set of operations $\tau_1, \tau_2, \ldots$. Successive states are represented as $\mathbf{X_1}, \mathbf{X_2}, \ldots$, where the notation λ |–, and the expression

$$X_i \vdash_{\tau_{i+1}} X_{i+1}$$

means that state transition τ_{i+1} moves the system from state $\mathbf{X_i}$ to state $\mathbf{X_{i+1}}$. When a system starts at some state **X** and, after a series of state transitions, enters state **Y**, we can write

$$\mathbf{X} \vdash^{*} \mathbf{Y}.$$

The representation of the protection system as an access control matrix must also be updated. In the model, sequences of state transitions are represented as single commands, or **transformation procedures**, that update the access control matrix. The commands state which entry in the matrix is to be changed, and how; hence, the commands require parameters. Formally, let $\mathbf{c_k}$ be the **k**th command with formal parameters $\mathbf{p_{k,1}}, \ldots, \mathbf{p_{k,m}}$. Then the **i**th transition would be written as

$$X_i \vdash_{c_{i+1}(p_{i+1,1}, \ldots, p_{i+1,m})} X_{i+1}.$$

Note the similarity in notation between the use of the command and the state transition operations. This is deliberate. For every command, there is a sequence of state transition operations that takes the initial state $\mathbf{X_i}$ to the resulting state $\mathbf{X_{i+1}}$. Using the command notation allows us to shorten the description of the transformation as well as list the parameters (subjects, objects, and entries) that affect the transformation operations.

We now focus on the commands themselves. Following Harrison, Ruzzo, and Ullman [450], we define a set of **primitive commands** that alter the access control matrix. In the following list, the protection state is (**S**, **O**, **A**) before the execution of each command and (**S'**, **O'**, **A'**) after each command. The preconditions state the conditions needed for the primitive command to be executed, and the postconditions state the results.

1. Precondition: $\mathbf{s} \notin \mathbf{O}$
   Primitive command: create subject **s**

Postconditions: $\mathbf{S'} = \mathbf{S} \cup \{\, \mathbf{s} \,\}$, $\mathbf{O'} = \mathbf{O} \cup \{\, \mathbf{s} \,\}$, $(\forall \mathbf{y} \in \mathbf{O'})[\mathbf{a'}[\mathbf{s}, \mathbf{y}] = \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S'})[\mathbf{a'}[\mathbf{x}, \mathbf{s}] = \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S})(\forall \mathbf{y} \in \mathbf{O})[\mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
This primitive command creates a new subject **s**. Note that **s** must not exist as a subject **or an object** before this command is executed. This operation does not add any rights. It merely modifies the matrix.

2. Precondition: $\mathbf{o} \notin \mathbf{O}$
   Primitive command: create object **o**
   Postconditions: $\mathbf{S'} = \mathbf{S}$, $\mathbf{O'} = \mathbf{O} \cup \{\, \mathbf{o} \,\}$, $(\forall \mathbf{x} \in \mathbf{S'})[\mathbf{a'}[\mathbf{x}, \mathbf{o}] = \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S'})(\forall \mathbf{y} \in \mathbf{O})[\mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
   This primitive command creates a new object **o**. Note that **o** must not exist before this command is executed. Like create subject, this operation does not add any rights. It merely modifies the matrix.

3. Precondition: $\mathbf{s} \in \mathbf{S}$, $\mathbf{o} \in \mathbf{O}$
   Primitive command: enter **r** into **a**[**s**, **o**]
   Postconditions: $\mathbf{S'} = \mathbf{S}$, $\mathbf{O'} = \mathbf{O}$, $\mathbf{a'}[\mathbf{s}, \mathbf{o}] = \mathbf{a}[\mathbf{s}, \mathbf{o}] \cup \{\, \mathbf{r} \,\}$, $(\forall \mathbf{x} \in \mathbf{S'})(\forall \mathbf{y} \in \mathbf{O'})[(\mathbf{x}, \mathbf{y}) \neq (\mathbf{s}, \mathbf{o}) \rightarrow \mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
   This primitive command adds the right **r** to the cell **a**[**s**, **o**]. Note that **a**[**s**, **o**] may already contain the right, in which case the effect of this primitive depends on the instantiation of the model (it may add another copy of the right or may do nothing).

4. Precondition: $\mathbf{s} \in \mathbf{S}$, $\mathbf{o} \in \mathbf{O}$
   Primitive command: delete **r** from **a**[**s**, **o**]
   Postconditions: $\mathbf{S'} = \mathbf{S}$, $\mathbf{O'} = \mathbf{O}$, $\mathbf{a'}[\mathbf{s}, \mathbf{o}] = \mathbf{a}[\mathbf{s}, \mathbf{o}] - \{\, \mathbf{r} \,\}$, $(\forall \mathbf{x} \in \mathbf{S'})(\forall \mathbf{y} \in \mathbf{O'})[(\mathbf{x}, \mathbf{y}) \neq (\mathbf{s}, \mathbf{o}) \rightarrow \mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
   This primitive command deletes the right **r** from the cell **a**[**s**, **o**]. Note that **a**[**s**, **o**] need not contain the right, in which case this operation has no effect.

5. Precondition: $\mathbf{s} \in \mathbf{S}$
   Primitive command: destroy subject **s**
   Postconditions: $\mathbf{S'} = \mathbf{S} - \{\, \mathbf{s} \,\}$, $\mathbf{O'} = \mathbf{O} - \{\, \mathbf{s} \,\}$, $(\forall \mathbf{y} \in \mathbf{O'})[\mathbf{a'}[\mathbf{s}, \mathbf{y}] = \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S'})[\mathbf{a'}[\mathbf{x}, \mathbf{s}] = \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S'})(\forall \mathbf{y} \in \mathbf{O'})[\mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
   This primitive command deletes the subject **s**. The column and row for **s** in **A** are deleted also.

6. Precondition: $\mathbf{o} \in \mathbf{O}$
   Primitive command: destroy object **o**
   Postconditions: $\mathbf{S'} = \mathbf{S}$, $\mathbf{O'} = \mathbf{O} - \{\, \mathbf{s} \,\}$, $(\forall \mathbf{x} \in \mathbf{S'})[\mathbf{a'}[\mathbf{x}, \mathbf{o}] =, \varnothing]$, $(\forall \mathbf{x} \in \mathbf{S'})(\forall \mathbf{y} \in \mathbf{O'})[\mathbf{a'}[\mathbf{x}, \mathbf{y}] = \mathbf{a}[\mathbf{x}, \mathbf{y}]]$
   This primitive command deletes the object **o**. The column for **o** in **A** is deleted also.

These primitive operations can be combined into commands, during which multiple primitive operations may be executed.

EXAMPLE: In the UNIX system, if process **p** created a file **f** with owner read (**r**) and write (**w**) permission, the command capturing the resulting changes in the access control matrix would be

command create•file(p,f)

```
  create object f;
  enter own into a[p,f];
  enter r into a[p,f];
  enter w into a[p,f];
end
```

Suppose the process **p** wishes to create a new process **q**. The following command would capture the resulting changes in the access control matrix.

```
command spawn•process(p,q)
  create subject q;
  enter own into a[p,q];
  enter r into a[p,q];
  enter w into a[p,q];
  enter r into a[q,p];
  enter w into a[q,p];
end
```

The **r** and **w** rights enable the parent and child to signal each other.

The system can update the matrix only by using defined commands; it cannot use the primitive commands directly. Of course, a command may invoke only a single primitive; such a command is called **mono-operational**.

EXAMPLE: The command

```
command make•owner(p,f)
  enter own into a[p,f];
end
```

is a mono-operational command. It does not delete any existing owner rights. It merely adds **p** to the set of owners of **f**. Hence, **f** may have multiple owners after this command is executed.

2.3.1. Conditional Commands

The execution of some primitives requires that specific preconditions be satisfied. For example, suppose a process **p** wishes to give another process **q** the right to read a file **f**. In some systems, **p** must own **f**. The abstract command would be

```
command grant•read•file•1(p,f,q)
  if own in a[p,f]
  then
    enter r into a[q,f];
end
```

Any number of conditions may be placed together using **and**. For example, suppose a system has the distinguished right **c**. If a subject has the rights **r** and **c** over an object, it may give any other subject **r** rights over that object. Then

```
command grant•read•file•2(p,f,q)
  if r in a[p,f] and c in a[p,f]
  then
    enter r into a[q,f];
end
```

Commands with one condition are called **monoconditional**. Commands with two conditions are called **biconditional**. The command **grant•read•file•1** is monoconditional, and the command **grant•read•file•2** is biconditional. Because both have one primitive command, both are mono-operational.

Note that all conditions are joined by **and**, and never by **or**. Because joining conditions with **or** is equivalent to two commands each with one of the conditions, the disjunction is unnecessary and thus is omitted. For example, suppose the right **a** enables one to grant the right **r** to another subject. To achieve the effect of a command equivalent to

```
if own in a[p,f] or a in a[p,f]
then
  enter r into a[q,f];
```

define the following two commands:

```
command grant•write•file•1(p,f,q)
  if own in a[p,f]
  then
    enter r into a[q,f];
end
```

```
command grant•write•file•2(p,f,q)
  if a in a[p,f]
  then
    enter r into a[q,f];
end
```

and then say

```
grant•write•file•1(p,f,q); grant•write•file•2(p,f,q);
```

Also, the negation of a condition is not permitted—that is, one cannot test for the **absence** of a right within a command by the condition

```
if r not in A[p,f]
```

This has some interesting consequences, which we will explore in the next chapter.