Formal Specification of Operating System Operations

Dan Zhou
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
dan@cse.fau.edu

Paul E. Black
National Institute of Standards and Technology
100 Bureau Drive, Stop 8970
Gaithersburg, Maryland 20899-8970
paul.black@nist.gov

This paper describes the development of a formal specification for a secure operating system architecture and its security-related operations in higher-order logic theorem prover, HOL. We specify operating system entities as user defined data types and security conditions as predicates that restrict system operations. We also provide a uniform environment for system commands that change the security state of systems.
Keywords: Formal specification, Secure operating system, Higher-order logic.

1. Motivation

An operating system (OS) is an important piece of the puzzle that provides system security as it enforces security policies of applications. The correctness of the operating system itself is paramount in ensuring the security of applications.

To facilitate high-assurance system development, we formally specify operating system security policies in the higher-order logic theorem prover, HOL98 [Gordon (1987)]. Formal specifications enables us to describe OS policies clearly and precisely. It also could benefit us in two more ways: (1) formal analysis using HOL theorem proving capability [Zhou and Chin (1999)], and (2) automated test generation based on formal specifications. In related work, we are investigating methods of automatically generating tests from HOL specifications.

Some previous works focus on abstract models of security protocols, which abstract away details that pertain to a real system requirement (e.g., [Jacob (1988)]); this type of specification and analysis is useful to protocol designers. Other work has specified high-level security policy models that can be used to validate real systems without getting into the details of actual systems [McLean (1990); Boswell (1995)].

In this work we model the security features of an OS at a level that is close to system requirements, which makes the model useful for system development. An OS at this level is large and complex, attributes which are major obstacles in specification efforts. We overcome this obstacle partly by borrowing object modeling technologies: dividing security policy into data types, functions on these data types, and operations and constraints on these operations that move the system from state to state.

We specify concrete entities that are components comprising the objects of operating systems. We also specify security conditions for system operations. We then describe security-related operations which present a user's view of operating systems. Specifically, we define different types of entities as HOL types and define security checks on these entities as HOL functions. We then present a uniform view of UNIX shell commands by providing an operating environment as lists of bindings of variable names, types, and values.

The specification of a secure operating system is part of a much larger effort in the late-development testing and post-development accreditation process. The specification developed in this work will be validated vigorously against a higher-level security policy model and will be used for automated test generation.

In the rest of the paper, we first describe entities in an OS in Section 2; we then describe security conditions for system operations in Section 3. In Section 4 we specify system states and the state transitions. We summarize in Section 5.

2. Operating System Entities

The subject of our case study is PITBULL [Argus (1999)], a security product that enhances the security of Solaris 7. It implements a restricted version of the Bell-LaPadula (BLP) model [Bell and LaPadula (1973)].

A security policy defines the rules that govern the operations of systems. A policy model provides a framework for defining rules. The BLP model addresses the security concerns of multi-level information systems. Abstract entities in the model are subjects and objects. Objects model resources to be controlled. Subjects are active entities that seek access, such as *read* or *write*, to objects.

The BLP model prevents unauthorized access to information through mandatory access control (MAC) mechanism, where the access control is mandated by the system. Subjects and objects are classified according to clearance and sensitivity level, respectively. They are also assigned compartments to which they belong. Subjects are granted access rights to objects based on their clearance and need-to-know categories.

In addition to MAC, PITBULL also provides standard UNIX discretionary access control (DAC), where owners of information can restrict or grant access to the information by using permission bits.

The only subjects in an OS are processes: normal processes and system processes. There are many different types of objects in an OS, such as regular file system objects (files, directories, etc), processes, and X-window objects. In this work we define subjects and objects by their security-related attributes: DAC attributes, MAC attributes, identifications, and privileges. PITBULL uses the security-related attributes of subjects, objects, and the system to determine whether to grant or deny access rights of subjects to objects.

2.1 DAC Attributes

UNIX-based systems provide DAC by assigning ownership to subjects and objects and permission bits to objects based on the identities of their owners. Users are clustered into groups for easy management. Permission bits on objects indicate the access rights (read, write, and execute) for the owner of the object (user), the group that the user belongs to (group), and the rest of the world (other). The HOL record type *PBS*

```
PBS = <|r: bool; w: bool; x: bool|>
```

models permission bits. The HOL record type *DAC*

```
DAC = <| up: PBS; gp: PBS; op: PBS|>
```

represents permission bits for the user, group and the world. In HOL definitions, the symbol `<|  ...  ...  |>` represents a record and ";" is a field delimiter.

Modern UNIX-based systems have many additional permission bits, such as set process ID to owner on execute (set UID), set set group ID on execute, mandatory file locking, and remove/rename file privilege (the "sticky" bit). We plan to add these, and other details, as we elaborate the model.

2.2 MAC Attributes

To enforce MAC, PITBULL provides hierarchical classification and clearance and non-hierarchical compartment security attributes to objects and subjects.

MAC Labels PitBull provides four types of MAC labels: sensitivity label (SL), clearance label (CL), information label (IL), and integrity label (TL). SLs and CLs consist of classification and compartment. SL is used for access control, CL is used to control operations on MAC labels. The HOL record type *SCLabel* represents both SLs and CLs:

```
SCLabel = <|class: Class; comp: Comp set |>
```

where type *Class* represents classifications and type *Comp* represents compartments. The construct *set* is a type constructor. The HOL types *ILabel* and *TLabel* represent IL and TL in a similar way.

We define a type *Label* to encompass all types of MAC labels:

```
Label = scLabel of SCLabel | iLabel of ILabel |
        tLabel of TLabel
```

Function *compOL* retrieves the compartment information of a label:

```
  (compOL (scLabel scl) = scl.comp) /\
  (compOL (iLabel il) = il.comp)
```

This can be read as, "For an *SCLabel scl*, the compartment is the *comp* field of *scl*, and for an *ILabel il*, the compartment is the *comp* field of *il*.

We define function *classOL* to retrieve the classification of a label similarly.

Classifications System administrators can set the values of classifications and compartments. As an example, we define a particular classification *Class* as an enumeration type with three values—*classTS*, *classS*, *classU*:

```
Class =  classTS | classS | classU
```

We intend classification *classTS* to be higher than *classS*, which in turn is higher than *classU*. In HOL we define two relations on type *Class*: *classGT* and *classGTE*, suggesting "Greater Than" and "Greater Than or Equal to". Predicate *classGT c1 c2* is true if and only if *c2* is higher than *c1*:

```
  (classGT classTS classS = T) /\
  (classGT classS  classU = T) /\
  (classGT classTS classU = T) /\
  (classGT _ _ = F)
```

where `classGT _ _ = F` states that the default value is $False$. Predicate *classGTE c1 c2* is true if and only if *c1* is higher than or equal to *c2*:

```
classGTE c1 c2 = (classGT c1 c2 \/ (c1 = c2))
```

Compartments We define *Comp* as an enumeration type with four values *NIST*, *ITL*, *FAU*, and *CSE* for compartments:

```
Comp = NIST | ITL | FAU | CSE
```

Since compartments are not in a hierarchy, but are entirely non-overlapping, there is no corresponding *compGT* or *compGTE*.

Relation among MAC Labels MAC decisions are made based on the relationship among MAC labels. Label *l1* dominates label *l2* if the classification of *l1* is at least as high as that of *l2* and the compartment set of *l1* is a superset of that of *l2*. If one of the labels is a TL, only the classifications of labels are compared. HOL function *RLLDOM* describes this relation:

```
(RLLDOM (tLabel tl1) l3 =
     classGTE tl1.class (classOL l3))     /\
(RLLDOM l4 (tLabel tl2) =
     classGTE (classOL l4) tl2.class)     /\
(RLLDOM l5 l6 =
     (classGTE (classOL l5) (classOL l6)) /\
     (compOL l6) SUBSET (compOL l5))
```

Function *RSLSLDOM* defines the dominance relation between two sensitivity labels:

```
RSLSLDOM sl1 sl2 = RLLDOM (scLabel sl1) (scLabel sl2)
```

Informally, if *RLLDOM l1 l2* is true, then MAC label *l1* dominates *l2*. If *RSLSLDOM s1 s2* is true, then sensitivity label *s1* dominates *s2*.

2.3 Privileges

The BLP model disallows write-down operations, some of which are necessary for the day-to-day operation of computer systems. Subjects privileges are the means to bypass restrictions the BLP model imposes and to perform otherwise unauthorized operations. A subject can obtain privileges by executing an executable file that contains special privileges. We describe privileges in detail to illustrate the complexity of an OS security policy.

There are three groups of privileges implemented on the PitBull system: general privileges, X-Window privileges, and superuser privileges (denoted by *Priv*, *XPriv*, and *SUPriv*, respectively). The general privileges are organized into eight subgroups (Table 1). Table 1 lists the number of privileges in each subgroup. Each subgroup contains privileges related to a particular aspect of system security. We define one data type in HOL for each group to denote the group-specific privileges.

These are enumerated types that list all group-specific privileges. For example, the subgroup for DAC privilege contains the twelve privileges that subjects can possess for bypassing DAC-related constraints (Table 2). The *DACPriv* represents the DAC privilege group:

```
DACPriv =
    PV_DAC           | PV_DAC_GID       |
    PV_DAC_O         | PV_DAC_O_NS      |
    PV_DAC_R         | PV_DAC_R_NS      |
    PV_DAC_SIG       | PV_DAC_UID       |
    PV_DAC_W         | PV_DAC_W_NS      |
    PV_DAC_X         | PV_DAC_X_NS;
```

Other general privilege subgroups are defined similarly.

A general privilege is either one of the privileges listed in Table 1 or PV_ROOT. It is defined in HOL as follows:

```
Priv = PV_ROOT                   |
       auPriv    of AUPriv       |
       azPriv    of AZPriv       |
       dacPriv   of DACPriv      |
       fsPriv    of FSPriv       |
```

Table 1: General System Privileges

Name	Description	No. of Privileges
$AUPriv$	Audit privileges allowing operations related to audit system	7
$AZPriv$	Authorization privileges allowing operations related to process authorization	4
$DACPriv$	DAC privileges allowing processes to bypass DAC-related restrictions	12
$FSPriv$	File system privileges related to file systems	2
$LABELPriv$	Label privileges related to access of labels such as information label and sensitivity label	17
$MACPriv$	MAC privileges allowing processes to bypass MAC restrictions	13
$ASNPriv$	Network, driver, and STREAM privileges	5
$PVPriv$	Privileges allowing processes to modify the privilege sets of files or processes	2
$SRPriv$	Privileges related to all other types of system resources	6
$MiscPriv$	Miscellaneous privileges	7

```
labelPriv of LABELPriv |
macPriv   of MACPriv   |
asnPriv   of ASNPriv   |
pvPriv    of PVPriv    |
srPriv    of SRPriv    |
miscPriv  of MiscPriv.
```

The HOL type definition

```
ty_new = C1 of ty1 | C2 of ty2 | ... | Cn of tyn | Cn+1
```

constructs a union type *ty_new* from types *ty1*, *ty2*, $\cdots$ *tyn* using type constructors *C1*, *C2* $\cdots$ *Cn*. Constant *Cn+1* is a null construct for type *ty_new*, which constructs an element of type *ty_new* from null. We call *ty1*, *ty2*, $\cdots$ *tyn* component types of *ty_new*.

X-Window privileges *XPriv* and superuser privileges *SUPriv* are defined similarly to for DAC privileges.

A privilege is either a general privilege, an X-Window privilege, or a superuser privilege. The HOL definition for a privilege is:

```
PV = priv of Priv    |
     xPriv of XPriv  |
     suPriv of SUPriv.
```

Privilege Hierarchy Privileges are organized hierarchically as a forest, with each group as a rooted tree (Figure 1). A privilege of a higher level in the hierarchy contains all the privileges that are lower in the tree. For example, PV_ROOT is the root of the hierarchy tree of general privileges, and PV_DAC is the root of DAC privileges. A process with privilege PV_DAC has all the privileges related to DAC; a process with privilege PV_ROOT has all the general privileges, including PV_DAC.

Table 2: DAC Privileges

Name	Description
PV_DAC	Equivalent to all of the other DAC privileges combined
PV_DAC_GID	Allows a process to change its GID
PV_DAC_UID	Allows a process to change its UID
PV_DAC_O	Bypass DAC ownership restrictions
PV_DAC_O_NS	Bypass DAC ownership restrictions for non-system files only
PV_DAC_R	Bypass DAC READ restrictions
PV_DAC_R_NS	Bypass DAC READ restrictions for non-system files only
PV_DAC_SIG	Allows a process to send a signal to an unrelated process
PV_DAC_W	Bypass DAC WRITE restrictions
PV_DAC_W_NS	Bypass DAC WRITE restrictions for non-system files only
PV_DAC_X	Bypass DAC EXECUTE/SEARCH restrictions
PV_DAC_X_NS	Bypass DAC EXECUTE/SEARCH restrictions for non-system files

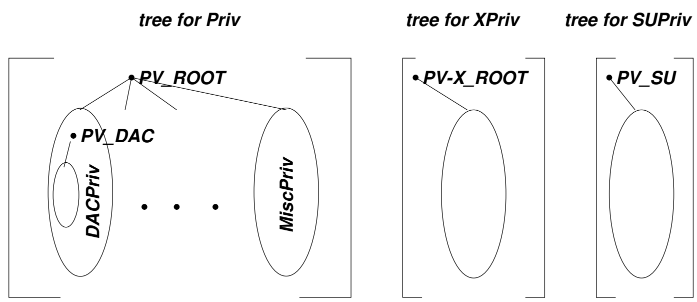

Figure 1: Privilege Hierarchy

To check if a subject has a particular privilege *pv*, we need to find out if *pv* or a privilege higher in hierarchy than pv (i.e., an ancestor privilege) is in the subject's privilege set. Two functions are defined to help this checking: a parent function that finds the parent privilege given a particular privilege, and a privilege checking function that recursively checks if a privilege or any of its ancestors is in a given privilege set.

We define a function $hrPV : PV \rightarrow PV$ as the parent function that reports the parent (i.e., the next higher-level privilege in hierarchy) of a given privilege. If a given privilege is the root of a hierarchy, its parent is itself. The function $hrPV$ is defined on data type PV by using its structure. In particular, we define a parent function for each subgroup privilege of the general privileges (see Table 3). For example, the function $hrDACPriv : DACPriv \rightarrow DACPriv$ is a parent function for $DACPriv$:

```
(hrDACPriv PV_DAC_R_NS = PV_DAC_R)     /\
(hrDACPriv PV_DAC_W_NS = PV_DAC_W)     /\
(hrDACPriv PV_DAC_X_NS = PV_DAC_X)     /\
(hrDACPriv PV_DAC_O_NS = PV_DAC_O)     /\
(hrDACPriv PV_DAC_SIG  = PV_DAC_O_NS) /\
(hrDACPriv _           = PV_DAC)
```

where $hrDACPriv\ \_ = PV\_DAC$ states that the default parent is PV_DAC.

We define a parent function $hrPriv$ for the general privileges based on the parent functions for the subgroup privileges. This definition uses the structure of data type $priv$ by calling the par-

Table 3: Parent Functions of Privilege Groups

Privilege Group	Parent Function	Privilege Group	Parent Function
$AUPriv$	hrAUPriv	$PVPriv$	hrPVPriv
$AZPriv$	hrAZPriv	$SRPriv$	hrSRPriv
$DACPriv$	hrDACPriv	$MiscPriv$	hrMiscPriv
$FSPriv$	hrFSPriv	$Priv$	hrPriv
$LABELPriv$	hrLABELPriv	$XPriv$	hrXPriv
$MACPriv$	hrMACPriv	$SUPriv$	hrSUPriv
$ASNPriv$	hrASNPriv	PV	hrPV

ent functions of its component types. Each subgroup has an "if" statement to return *PV_ROOT* if the subgroup root is passed.

```
(hrPriv (auPriv aupv) =
   if (aupv = PV_AU) then PV_ROOT
   else (auPriv (hrAUPriv aupv)))       /\
(hrPriv (asnPriv asnpv) =
   if (asnpv = PV_ASN) then PV_ROOT
   else (asnPriv (hrASNPriv asnpv)))    /\
(hrPriv (dacPriv dpv) =
   if (dpv = PV_DAC) then PV_ROOT
   else (dacPriv (hrDACPriv dpv)))      /\
(hrPriv (labelPriv lpv) =
   if (lpv = PV_LABEL) then PV_ROOT
   else (labelPriv (hrLABELPriv lpv)))  /\
(hrPriv (macPriv mpv) =
   if (mpv = PV_MAC) then PV_ROOT
   else (macPriv (hrMACPriv mpv)))      /\
(hrPriv (srPriv spv) =
   if (spv = PV_SR) then PV_ROOT
   else (srPriv (hrSRPriv spv)))        /\
(hrPriv _ = PV_ROOT)
```

Parent functions are defined for X-Window privileges and superuser privileges similarly. Finally, we define a parent function $hrPV$ for all privileges as follows:

```
(hrPV (priv ppv)   = priv (hrPriv ppv))     /\
(hrPV (suPriv spv) = suPriv (hrSUPriv spv)) /\
(hrPV (xPriv xpv)  = xPriv (hrXPriv xpv))
```

We can now define a recursive function $hasPV : PV \rightarrow PV\ set \rightarrow bool$ as a privilege checking function to check if a privilege pv is in privilege set $pvSet$. The function $hasPV$ first looks for pv in $pvSet$, then looks for $(hrPV\ pv)$, and then for $(hrPV\ (hrPV\ pv))$, until it looks for PV_ROOT, PV_X_ROOT or PV_SU_EMUL in $pvSet$. The function $hasPV$ is defined in HOL as follows:

```
hasPV (priv PV_ROOT) pvSet
      = (priv PV_ROOT) IN pvSet     /\
hasPV (xPriv PV_X_ROOT) pvSet
      = (xPriv PV_X_ROOT) IN pvSet  /\
hasPV (suPriv PV_SU) pvSet
```

```
       = (suPriv PV_SU) IN pvSet     /\
hasPV pv pvSet
       = (pv IN pvSet \/ hasPV (hrPV pv) pvSet)
```

where IN is a function in HOL's *set* library. Expression $(e\ IN\ s)$ returns *true* if e is an element of set s, otherwise it returns *false*.

3. Security Conditions

Security conditions are the rules that system operations obey. They can be rather complex. Sometimes they are also confusing because of the many factors contributing to a decision making process. The complexity of the conditions warrants formal specification and analysis to ensure the system behaves as we wish.

As an example, we show the condition for modifying an object's sensitivity level (SL). A process can change the SL of an object or subject if and only if all these conditions are met.

- The process has either privilege PV_SL_FILE (object only) or PV_SL_PROC (subject only)
- The process has DAC WRITE access to the object (object only)
- The process has MAC WRITE access to the object or subject
- The process has DAC OWNER access to the object or subject (downgrade only)
- The process has either privilege PV_SL_UG (upgrade only) or PV_SL_DG (downgrade only)
- The process has privilege PV_MAC_CL (if either the new SL or the existing SL is not dominated by the acting process's CL)
- The new SL dominates the IL and is dominated by the CL (subject only).

The condition for modifying the SL of an object is defined as *procMdSLCond* in HOL:

```
procMdSLCond s p os sl =
(if (isSubjObj os) then
    (hasPV_PSP s p /\
     RLLDOM (scLabel sl)
             (iLabel (ilMACObj os)) /\
     RLLDOM (scLabel (clMACObj os))
             (scLabel sl))
 else (hasPV_PSF s p /\ hasRT_DW p os))  /\
hasRT_MW p os      /\
(if ~(RSLSLDOM (slMACObj os) sl)
 then (hasRT_DO p os /\ hasPV_PSD s p)
 else (hasPV_PSU s p))      /\
(if (~(RSLSLDOM p.MAC.CL sl) \/
   ~(RSLSLDOM p.MAC.CL (slMACObj os)))
 then hasPV_PMC s p
 else T)
```

4. System Operations

As examples, we use UNIX commands typed at the UNIX prompt which are executed by system processes. These operations change an operating system from state to state.

4.1 System States

A system state is the collection of entities that are active in the system and the values of these entities. A state is represented as a list of entity names bound to their values. Names and values are represented uniformly as unions containing the type information of the actual entities.

In this simplified state machine model, we assume that there are three types of entities in the operating system: system, subjects, and files (denoted by *sysType*, *subjType*, and *fileType* respectively). We use a string plus a keyword to identify an entity, for example, *fileVar "paper.tex"* for a file named paper.tex and *sysVar "p1"* for a process named p1. The type *VarType* represents the entity names in HOL as follows:

```
VarType =
    sysVar  of string |
    subjVar of string |
    fileVar of string
```

The type *ValType* represents the possible values of entities of type *VarType*:

```
ValType =
    sysVal  of SysType  |
    subjVal of SubjType |
    fileVal of FileType
```

We use a pair to represent the binding of the name and value of an entity. This is denoted as type *bindTy = (VarType # ValType)* in HOL, where # constructs a pair type. Functions *varB* and *valB* retrieve the name and value of the binding of a given entity. Both the name and value contain the actual type of the entity (see Table 4).

We define a *store* as a list of bindings that represent the operating environment of an operation, or the state of the operating system. The type of a store is *storeTy = (VarType # ValType) list*. A change to the environment is represented by prepending a binding to the *store*. To evaluate a variable, we scan the *store* from the beginning until we find a match, at which point the corresponding value is returned. Table 4 lists auxiliary functions that evaluate values of entities.

4.2 State Transitions

A security policy has two components: if an operation should be permitted (authorization), and how it changes the system state if the operation is executed (state transition). Security rules determine the circumstances under which a request should be permitted or rejected.

For example, the execution of a command to set the sensitivity label of a file is divided into two parts. First, the authorization function *setSLFileCond* uses *procMdSLCond* to check the access right of the process making the change:

```
setSLFileCond st p f sl =
procMdSLCond (sysE st system) (procE st p)
   (fsoObj (fileFSO (fileE st f))) sl
```

where *sl* is the new sensitivity label to be set on file *f*. Second, the state transition function *setSLFile* describes how the system state is changed:

Table 4: Evaluation of System Operations

name	description	definition
varB	return a name given a name-value binding	`varB (a, b) = a`
valB	return a value given a name-value binding	`valB (a, b) = b`
varE	return an entity given a name	`varE (h::t) x = if (varB h = x) then valB h else varE t x`
unpkSys	return a system given a value	`unpkSys (sysVal s) = s`
unpkFile	return a file given a value	`unpkFile (fileVal f) = f`
unpkProc	return a process given a value	`unpkProc (subjVal p) = p`
sysE	return a system from a store	`sysE st s = unpkSys (varE st (sysVar s))`
fileE	return a file from a store	`fileE st f = unpkFile (varE st (fileVar f))`
procE	return a process from a store	`procE st p = unpkProc (varE st (subjVar p))`

```
setSLFile (f:FileType) (sl:SCLabel) =
(f with MAC := (f.MAC with SL := sl))
```

Function *setSLFileEnv* models the operation of a process attempting to set the sensitivity label of a file. This operation may or may not change the state of the system, depending on if the process has the right to perform the operation.

UNIX commands are accompanied by arguments to the system such as the new label or the file name. To model the arguments that are passed to these commands, we define an argument type:

```
ArgType = strArg of string | sclArg of SCLabel
```

where *strArg* and *sclArg* are two type constructors for type *ArgType*. An argument is either a string representing an entity whose value can be retrieved from a store or a new sensitivity label to be set.

```
setSLFileEnv st
   [strArg p; strArg f; sclArg sl] =
if setSLFileCond st p f sl
then update st (fileVar f)
     (fileVal (setSLFile (fileE st f) sl))
else st
```

Function $update$ prepends new values to the *store*. We can similarly define $setSLProc$ as a command to change process sensitivity labels.

The system evolves as commands are entered and executed. The *nextstate* function models this evolution:

```
(nextstate st1 set_sl_file arglist1 =
           setSLFileEnv st1 arglist1) /\
(nextstate st2 set_sl_proc arglist2 =
           setSLProcEnv st2 arglist2)
```

4.3 Example

Let's look at a UNIX command that changes file sensitivity labels. Suppose we have a file *f1* whose uid is 1 and gid is 10. Its permission bits are `rw-r--r--`. Its classification is *classTS* and its compartment set is {*NIST*}.

We have a process *p1* whose euid is 1 and egid is 10. It only has privilege PV_DAC. Its sensitivity label (SL) is as follows: clearance level is of *classTS* and the compartment set is {*NIST*}. Its clearance label (CL) is the same as its SL.

Process *p2* has the same attributes as *p1*, but also has privileges PV_SL_FILE and PV_SL_DG.

The clearance labels that *system* is authorized to handle are between *classS* and *classTS*. The compartments that *system* is authorized to process are *NIST* and *FAU*.

The initial state of OS is stored as a list:

```
state1 = [(sysVar "system", sysVal system);
          (fileVar "f1", fileVal f1);
          (subjVar "p1", subjVal p1);
          (subjVar "p2", subjVal p2)]
```

Suppose *sl2* is a sensitivity label with clearance level *classS* and compartment set {*NIST*}. A request by process *p2* to set the sensitivity label of *f1* to *sl2* is decided by predicate

```
setSLFileCond state1 "p2" "f1" sl2.
```

Process *p2* is allowed to perform the operation based on the conditions listed in *setSLFileCond*. The state of OS changes to

```
state2  = [(sysVar "system", sysVal system);
          (fileVar "f1", fileVal f1new);
          (subjVar "p1", subjVal p1);
          (subjVar "p2", subjVal p2)],
```

where *f1new* is the *f1*, except that its classification is *classS*.

5. Experience

We specified part of a security operating system architecture. This is a complex structure with a large number of types of entities and security conditions enforced on the system operations.

The HOL language is suitable for describing large systems because of its expressibility. HOL provides a variety of ways to define data types, similar to those that can be defined using a BNF format. The ability to define types and functions recursively makes specification writing similar to that of writing in a high-level programming language. Unlike many high-level programming languages, HOL is a strongly-typed language and HOL expressions are precise.

This project started after PITBULL had been developed. It was desirable to have a formal model for accreditation purpose. Informal security policy description and security architecture were available to us. We ran into a few difficulties. The main difficulty came from the complexity of the system. We spent a lot of time familiarizing ourselves with the inter-weave of requirements that appear in different places. As a specification aid, we used the object-oriented modeling technology UML to help organize the formal specifications [Zhou (2000)]. A second difficulty came from the ambiguity of informal descriptions that are available to us. We found a few inconsistencies and mistakes in the published informal descriptions. On other occasions, we spent significant amounts of time explaining our confusion to system developers to get a clear answer.

Formal specifications are subject to mistakes if there are no validations. For example, we once mistakenly wrote a *less than* instead of a *greater than* in a relationship definition. A formal evaluation would help as would a formal analysis of specifications. We plan to do just that in a follow-up study.

We also plan to use these formal specifications to generate tests and evaluate the design against security criteria.

References

Argus (1999). *Trusted Facility Manual, Argus Security Solutions for Solaris 7, Fortify/PitBull*. Argus Systems Group, Inc.

Bell, D. E. and LaPadula, L. J. (1973). Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE Corporation, Bedford, MA.

Boswell, A. (1995). Specificaiton and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):63–68.

Gordon, M. (1987). A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer.

Jacob, J. (1988). Security specifications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 14–23.

McLean, J. (1990). The specification and modeling of computer security. *IEEE Computer*, pages 9–15.

Zhou, D. (2000). Towards the formal modeling of a secure operating system. In *Proceedings of the 23rd National Information System Security Conference*, Baltimore, Maryland.

Zhou, D. and Chin, S.-K. (1999). Formal Analysis of a Secure Communication Channel: Secure Core-Email Protocol. In *The Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, Toulouse, France.