specification at least allows to explain precisely the interpretation we have chosen, so that misunderstandings can be much easier detected than in a natural-language specification.

## 6.5 A Formal Evaluation Example

In this section we will use the formalized requirement introduced above to illustrate the formal verification process necessary to prove that a product possesses the properties of an ITSEC functionality class. As an example we will consider a UNIX implementation allowing administration of rights by means of *access control lists (ACL)*. This technique is for example described in [39].

### 6.5.1 UNIX Access Control Lists

The UNIX extensions necessary to certify F-C2 functionality[5] to this operating system include the definition of discretionary access rights by means of *access control lists (ACL)*. Every UNIX file is associated with a list of entries of the form

```
(user.group, rwx)
```

Here, `user` is a subject known to the system, `group` describes a UNIX group associated with `user` and `rwx` stands for the types *read, write, execute* of access right. For `user` and `group` a "don't-care" symbol `%` may be used. If users try to access a file, their identification and the associated groups are compared with the ACL of the file. This comparison algorithm observes the following priorities (1 = highest):

1. `(u.g,rwx)` — specific user, specific group
2. `(u.%, rwx)` — specific user, arbitrary group
3. `(%.g, rwx)` — arbitrary user, specific group
4. `(%.%, rwx)` — arbitrary user, arbitrary group

Any pair `u.g, u.%, %.g, %.%` must occur at most once in the ACL. For example, the entries `(u1.g1, r--), (u1.g1, -w-)` cannot be contained in the ACL at the same time, whereas this is possible for `(u1.g1,r--)` and `(u1.%,---)`.

As soon as a matching ACL entry allowing object access has been found, only matching ACL entries of higher priority are inspected. Access is finally granted if no matching higher-priority entry preventing object access is found. If a user or process is associated with more than one group, the access rights of all matching ACL entries are summarized by means of an OR combination.

**Example 6.1** Let a UNIX file have the ACL

```
(u2.g2,r--), (u2.%,---), (%.g1,---), (%.g3,r-x), (%.g4,-w-)
```

Suppose, user `u1` is associated with group `g1`, user `u2` with groups `g1,g2` and user `u3` with groups `g1,g3,g4`.

[5] F-C2 is the next functionality class higher than F-C1.

u1 cannot access the file, because (u1,g1) only matches ACL entry (%.g1,---), and this entry explicitly forbids object access. u2 is allowed to read the file because this user is a member of group g2; if the g2 membership were removed from u2, access would be denied. User u3 has full access, since he is a member of groups g3, g4; if he would only be in group g1, access would be denied, if he would only be in g4, he would be allowed to write onto the file.
□

### 6.5.2 Formalization of the ACL Model

For being able to prove that the ACL model fulfills the F-C1 requirements, we need a formal specification of the ACL data structures and a mathematical description of the conditions granting file access to a user in the ACL model. This is accomplished by a second Z specification representing a *concrete* security product specification, as implemented by the ACL model.

Let

$$[UXUSERS, UXGROUPS]$$

be the sets of concrete UNIX user identifications and groups, including the don't-care symbols $\%_{users}, \%_{groups}$:

$$\begin{array}{|l} \%_{users} : UXUSERS \\ \%_{groups} : UXGROUPS \end{array}$$

$$[UXFILES]$$

is the concrete representation of the general objects defined in the previous paragraph. Define

$$PERMISSION ::= r \mid w \mid x$$

as the set of UNIX access types.

Using the Z notation, an ACL entry (u.g,rwx) can be expressed as a schema

$$\begin{array}{|l} \hline \textit{ACLentry} \\ user : UXUSERS \\ group : UXGROUPS \\ perm : \mathbb{P}\, PERMISSION \\ \hline \end{array}$$

and each complete ACL is a sequence of such entries; i. e. an element of

$$ACLset == \operatorname{seq} ACLentry$$

Each object of our little UNIX universe can now be expressed by a Z schema

$$
\begin{array}{|l}
\hline
\textit{UXobject} \\
f : \mathit{UXFILES} \\
acl : \mathit{ACLset} \\
\hline
\forall\, i, j : \operatorname{dom} acl \bullet \\
\qquad (acl\ i).user = (acl\ j).user \wedge (acl\ i).group = (acl\ j).group \Rightarrow i = j \\
\hline
\end{array}
$$

Here, $(acl\ i).user$ denotes the `user`-component of the $i$th ACL entry; dom $acl$ is the domain of all indexes $1, \ldots, \#acl$ numbering the list entries.

This predicate part of the schema states that every `user.group` entry may appear only once in the ACL.

The priority ordering of ACL entries introduced above can be formally defined as a relation between pairs `(user,group)`:

$$
\begin{array}{|l}
\_ \leq_{ACL} \_ : (\mathit{UXUSERS} \times \mathit{UXGROUPS}) \leftrightarrow (\mathit{UXUSERS} \times \mathit{UXGROUPS}) \\
\hline
\forall\, u1, u2 : \mathit{UXUSERS};\ g1, g2 : \mathit{UXGROUPS} \bullet \\
\qquad (u1, g1) \leq_{ACL} (u2, g2) \Leftrightarrow \\
\qquad\qquad (u1 = u2 \wedge (g1 = g2 \vee g2 = \%_{groups})) \\
\qquad\qquad\qquad \vee\ (u1 \neq u2 \wedge u2 = \%_{users} \wedge (g1 = g2 \vee g1 = \%_{groups} \vee g2 = \%_{groups}))
\end{array}
$$

Now for example the formal specification *"UNIX user is granted read access to UNIX file"* can be mathematically described by Z schema

$$
\begin{array}{|l}
\hline
\textit{UXgrantReadAccess} \\
\textit{UXobject} \\
u? : \mathit{UXUSERS} \\
g? : \mathbb{F}_1\ \mathit{UXGROUPS} \\
\hline
\exists\, gr : g?;\ a : \operatorname{ran}\ acl \bullet \\
\qquad r \in a.perm \wedge \\
\qquad (u?, gr) \leq_{ACL} (a.user, a.group) \wedge \\
\qquad (\forall\, b : \operatorname{ran}\ acl \setminus \{ a \} \bullet \\
\qquad\qquad \neg\ ((u?, gr) \leq_{ACL} (b.user, b.group) \wedge \\
\qquad\qquad\qquad (b.user, b.group) \leq_{ACL} (a.user, a.group))) \\
\hline
\end{array}
$$

Here, $a : \operatorname{ran}\ acl$ denotes an element contained in the ACL.

In natural-language style, this can be expressed as

> *User $u?$ has read access to the UNIX file, if a group $gr \in g?$ is associated with $u?$, such that*
>
> 1. *There exists a matching ACL entry 'a' granting at least read permission to $(u?, gr)$.*
> 2. *There is no other ACL entry matching with $(u?, gr)$ and possessing a higher priority than 'a'.*

Note that the set of groups $g?$ must be non-empty. This reflects the fact that UNIX does not admit users without any associated groups.

Similar schemas *UXgrantWriteAccess* and *UXgrantExecuteAccess* can be defined by replacing the term $r \in a.perm$ by $w \in a.perm$ and $x \in a.perm$, respectively. To specify the condition for full read-, write- and execute-access, we construct the AND-combination

$$
\begin{array}{l}
UXgrantFullAccess \mathrel{\widehat{=}} \\
\quad UXgrantReadAccess \land UXgrantWriteAccess \land UXgrantExecuteAccess
\end{array}
$$

of all three schemas.

UNIX access control is a deterministic and total operation. As a consequence, the condition for rejecting read access (and analogously write and execute access) is just the negation of the predicate part of *UXgrantReadAccess*, i. e.

$$
\begin{array}{|l}
\hline
\textit{UXdenyReadAccess} \\
\hline
UXobject \\
u? : UXUSERS \\
g? : \mathbb{F}_1\ UXGROUPS \\
\hline
\forall\, gr : g?;\ a : \operatorname{ran} acl \bullet \\
\quad r \notin a.perm \lor \\
\quad \neg\, ((u?, gr) \leq_{ACL} (a.user, a.group)) \lor \\
\quad (\exists\, b : \operatorname{ran} acl \setminus \{ a \} \bullet \\
\qquad ((u?, gr) \leq_{ACL} (b.user, b.group) \land \\
\qquad\quad (b.user, b.group) \leq_{ACL} (a.user, a.group))) \\
\hline
\end{array}
$$

Analogously, *UXdenyWriteAccess, UXdenyExecuteAccess* are defined, so giving only limited access or none at all is specified by

$$
\begin{array}{l}
UXdenyFullAccess \mathrel{\widehat{=}} \\
\quad UXdenyReadAccess \lor UXdenyWriteAccess \lor UXdenyExecuteAccess
\end{array}
$$

### 6.5.3 Instantiation of the Generic Specification

To be able to verify the concrete specifications *UXgrantFullAccess* and *UXdenyFullAccess* of the product against the abstract specifications, we first have to instantiate the generic data types used in *GrantAccess* and *DenyAccess* with the concrete data structures of the product. To this end, we create a new F-C1 object specification by instantiating it with the concrete UNIX data types *UXFILES, UXUSERS, UXGROUPS*:

$$FC1objectForUnix \mathrel{\widehat{=}} Object[UXFILES, UXUSERS, UXGROUPS]$$

### 6.5.4 Abstraction Relation Between F-C1 Model and ACL Model

The process of replacing an abstract specification (in our case the formal F-C1 specification, instantiated with the concrete UNIX data types) by a concrete one (the ACL model) is a