A Verification Approach for Applied System Security

Achim D. Brucker[1], **Burkhart Wolff**[2]

[1] Information Security, ETH Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland
e-mail: brucker@inf.ethz.ch

[2] Universität Freiburg, George-Köhler-Allee 52, D-79110 Freiburg, Germany
e-mail: wolff@informatik.uni-freiburg.de

Published online: 25 January 2005

Abstract. We present a method for the security analysis of realistic models over off-the-shelf systems and their configuration by formal, machine-checked proofs. The presentation follows a large case study based on a formal security analysis of a CVS-Server architecture.

The analysis is based on an abstract architecture (enforcing a role-based access control), which is refined to an implementation architecture (based on the usual discretionary access control provided by the POSIX environment). Both architectures serve as a skeleton to formulate access control and confidentiality properties.

Both the abstract and the implementation architecture are specified in the language Z. Based on a logical embedding of Z into Isabelle/HOL, we provide formal, machine-checked proofs for consistency properties of the specification, for the correctness of the refinement, and for security properties.

Keywords: verification, security, access control, refinement, POSIX, CVS, Z

1 Introduction

These days, the *Concurrent Versions System* (CVS) is a widely used tool for version management in many industrial software development projects, and plays a key role in open source projects usually carried out by highly distributed teams [3,4]. (See http://www.cvshome.org.) CVS provides a central database (the *repository*) and means to synchronize local modifications of partial copies (the *working copies*) with the repository. The repository can be accessed via a network; this requires a security architecture establishing authentication, access control and non-repudiation. A further complication of the CVS security architecture stems from the fact that the administration of authentication and access control is done via CVS itself; i.e. the authentication table is accessed and modified via standard CVS operations.

This work emerged from our own experiences with setting up a CVS-Server for more than 80 users worldwide. Besides overcoming a number of security problems (see, e.g. http://www.cvshome.org/dev/security9706.html, we had to develop an improved CVS-Server configuration described in [1] meeting two system design requirements: first, we had to provide a configuration of a CVS-Server that enforces a role-based access control [13]; second, we had to develop an "open CVS-Server architecture", where the repository is part of the shared filesystem of a local network and the server is a regular process on a machine in this network. While such an architecture has a number of advantages, the correctness and trustworthiness of the security mechanisms become a major concern. Thus, we decided to apply formal modeling and analysis techniques to meet the challenge.

In this paper, we present the method we developed for analyzing the security problems of complex systems such as the CVS-Server and its configuration. As a result, we provide the following contributions:

1. a modeling technique that we call *architectural modeling*, which has an abstraction level in-between the usual behavioral modeling used in protocol analysis and code verification,
2. a technique to use system architecture models for defining security requirements,
3. the presentation of the mapping from security requirements to concrete security technologies as a data refinement problem,

4. mechanized proof-techniques for refinements and security properties over system transitions, and
5. reusable models for widely used security technologies.

In particular, we provide means to model a certain type of security policies, and show how security analysis can be performed not only on the abstract, but also on the concrete level.

The paper is organized as follows: After introducing some background material, e.g. CVS, our chosen specification formalism Z and the architectural modeling style, we present the model of the abstract system architecture. We proceed with the model of the POSIX filesystem as an infrastructure for the implementation architecture, and present the implementation architecture itself. Then, we describe the refinement relation between the system architecture and the implementation architecture, and the analysis of security properties at the different layers based on formal proofs in an interactive theorem prover.

2 Background

2.1 The CVS Operations

For the purpose of this paper, it is sufficient to mention only the most common CVS commands (initiated by the client). These are: login for client authenticating, add for registering files or directories for version control, commit for transferring local changes to the repository, and update for incorporating changes from the repository (e.g. fetching the latest version from the repository) into the working copy. Additionally, CVS provides functionality for accessing the history, for branching, for logging information which is out of the scope of this paper, and a mechanism for conflict resolution (e.g. merging the different versions) which is only modeled as an abstract operation. Further, in order to facilitate both the refinement and the security analysis, we will include in our CVS model a operation which is strictly speaking not part of CVS but of the operating system: the operation modify. This operation models changes of the working copy, e.g. by editing a file.

2.2 Z and Isabelle/HOL-Z

As our specification formalism, we chose Z [16] for the following reasons: first, Z fits our modeling problem since the complex states of our components suggest to use a formalism with rich theories for data-structures. Second, syntax and semantics of Z are specified in an ISO-standard;[1] for future standardization efforts of operating system libraries (e.g. similar to the POSIX [17] model in Sec. 3.3.2), Z is therefore a likely candidate. Third, Z comes with a data-refinement notion [16, p. 136], which provides a correctness notion of the underlying "security technology mapping" between the two architectures and a means to compute the proof obligations. We assume a rough familiarity with Z (the interested reader is referred to excellent textbooks on Z such as [16,18]).

As our modeling and theorem-proving environment, we chose Isabelle/HOL-Z [2], which is an integrated documentation, type-checking, and theorem-proving environment for Z specifications, which is built on top of Isabelle/HOL. Isabelle [9] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [7], a classical logic with equality. Isabelle/HOL-Z is a conservative embedding of Z into HOL (which is semantically isomorphic to Z). As a result, Isabelle/HOL-Z combines up-to-date theorem proving technology with a widespread, standardized specification formalism, and powerful documentation facilities.

2.3 Architectural Modeling

As a means to identify conceptual entities of the problem domain and to structure the overall specification, we found it useful to describe the *architecture* of the system on several abstraction layers. Following Garlan and Shaw's approach [6,15], architectures are composed by *components* (such as *clients*, *servers* or *stores* like the filesystem) and *connectors* (like *channels*, *shared variables*, etc). In this terminology it is straight forward to make the mentioned architectures more precise (as implementation architecture, we present the intended "open server architecture", see Fig. 1). We assume for each operation (such as add) a shared variable as connector that keeps all necessary information that goes to and from the components. This paves the way to formalize this architecture by describing the transition relation of the combined system by the parallel composition of the local transition relations of the components synchronized over the corresponding shared variable. Since such transition relations can be represented in Z by *operation schemas*, we can thus define, for example:

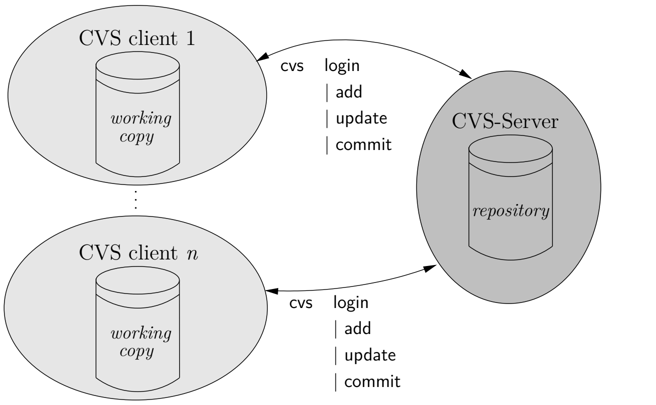
$$\begin{aligned} CVS\_add = {} & Client\_add \\ & \land Server\_add \setminus add_{\text{shared variable}} \end{aligned}$$

where $\land$ is the schema-conjunction and $\setminus$ the hiding operator (i.e. an existential quantifier). Throughout this paper, we will only present combined operation schemas and model properties over the transitive closure of their transition relations.

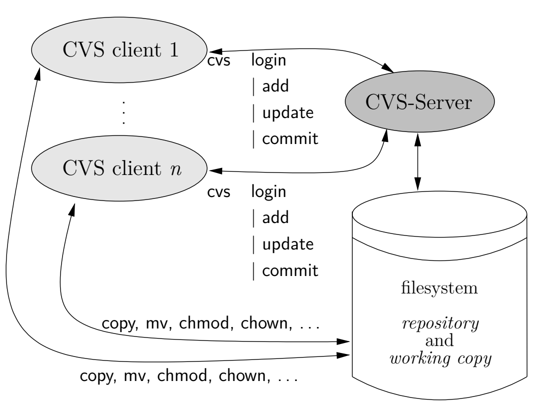
2.4 Architecture Refinement

When analyzing security architectures one can separate an *abstract security architecture* (see Sec. 3.2), which

[1] Z formal specification notation Âŋ syntax, type system and semantics, 2002. ISO/IEC 13568:2002



(a) "system"



(b) "implementation"

Fig. 1: The different CVS-Server architectures

is merely a framework for describing the security requirements, from an *implementation architecture* (see Sec. 3.3) where a mapping to security mechanisms is described (see Fig. 2). By connecting the abstract and the concrete layer formally, it is possible to reason about safety and security properties on the abstract level. Such a connection between abstract and more concrete views on a system and their semantic underpinning is well-known under the term *refinement*, and security technology mappings can be understood as a special case of this. Various refinement notions have been proposed [18, 11]; in our setting, we chose to use only a simple data refinement notion following Spivey [16].

2.5 Security Models vs. Security Technologies

Many security models distinguish between *objects* (e.g. data) and *subjects* (e.g. users). Using *role-based access control* (RBAC) [13] one assigns each subject at least one role (e.g. the role "administrator"), and access of objects is granted or denied by the role a subject is acting in. Further, roles can be hierarchically ordered, e.g. subjects in the role "administrator" are allowed to do everything other roles are allowed to. Our CVS-Server uses such a *hierarchic RBAC* model.

In an RBAC model, the decision, which roles may have access to which objects is done during system design and cannot be changed by regular users. In contrast, in a *discretionary access control* (DAC) model, every object belongs to a specific subject (its *owner*), and the owner is allowed to change the access policies at any time, hence "discretionary". For example a DAC implementation that also allows grouping users is the Unix/POSIX filesystem layer [17] access control.

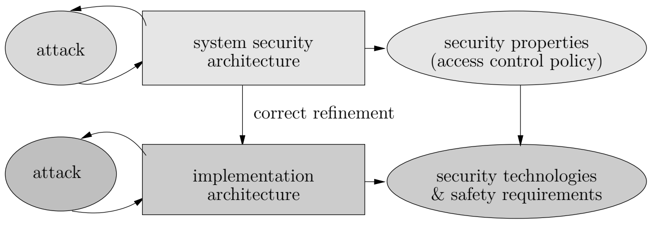

attack
system security architecture
security properties (access control policy)
correct refinement
attack
implementation architecture
security technologies & safety requirements

Fig. 2: Refining Security Architectures

Based on a DAC that supports groups, one can 'implement' an RBAC model by a special setup [12]. We use a similar technique to implement a hierarchic RBAC model for our CVS-Server on top of the POSIX filesystem layer, which is described in Sec. 3.3.3. However, we will analyze the concrete form in which DAC is implemented in POSIX and not a conceptual model thereof.

3 The CVS-Server Case Study

The Z specification of the CVS-Server [1] consists of more than 120 pages, and the associated proof scripts are about 13000 lines of code. The organization of the Z-sections follows directly the overall scheme presented in Fig. 3. The Z-sections *AbsState* and *AbsOperations* are describing the abstract system architecture of the client and the server components. The Z-section *SysConsistency* contains the consistency conditions (conservatism of axiomatic definitions, definedness of applications, non-blocking operation schemas) of the system architecture. This is mirrored at the implementation architecture level by the structures *FileSystem*, *CVS-Server*, and *ImplConsistency*. The Z-section *Refinement* contains the usual abstraction predicates relating the abstract and the concrete states, and also the proof obligations for this refinement. The security properties, together with the corresponding proof obligations, are defined in the sections *SysArchSec* and *ImplArchSec*.

3.1 Entities of the Security Model

Following the standard RBAC model, we introduce abstract types for CVS clients (users) *Cvs_Uid*, permissions

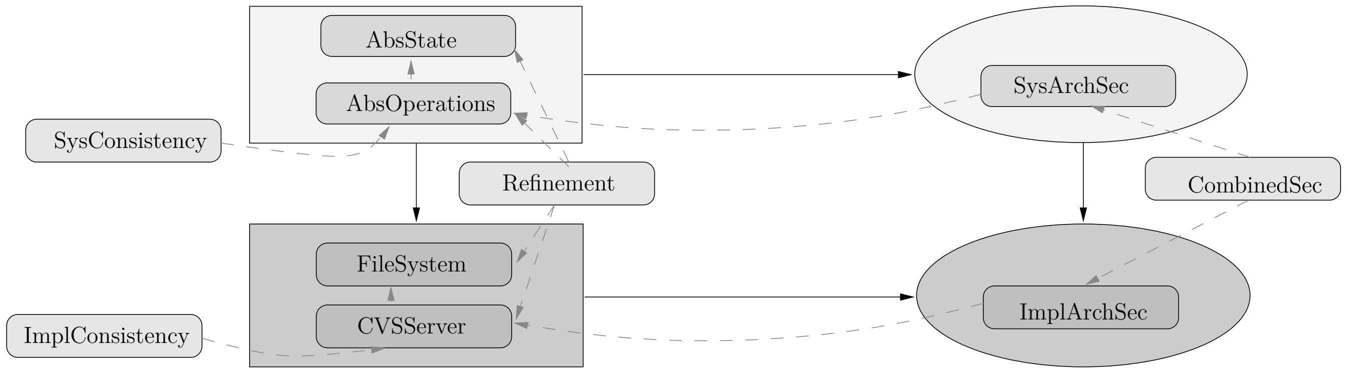


Fig. 3: Organizing the Specification into Z-sections

Cvs_Perm (which are isomorphic to roles in our setting), and CVS passwords *Cvs_Passwd* used to authenticate a CVS client for a permission:

$[Cvs\_Uid, Cvs\_Perm, Cvs\_Passwd]$

Permissions are hierarchically organized by the reflexive and transitive relation *cvs_perm_order* (over permissions *Cvs_Perm*) with *cvs_adm* as greatest element:

$$\begin{array}{|l}
cvs\_adm, cvs\_public : Cvs\_Perm \\
cvs\_perm\_order : Cvs\_Perm \leftrightarrow Cvs\_Perm \\
\hline
cvs\_perm\_order = cvs\_perm\_order^{*} \\
\forall x : Cvs\_Perm \bullet (x, cvs\_adm) \in cvs\_perm\_order \\
\forall x : Cvs\_Perm \bullet (cvs\_public, x) \in cvs\_perm\_order \\
\forall x : Cvs\_Perm \bullet (x \neq cvs\_adm) \Rightarrow \\
\qquad (cvs\_adm, x) \notin cvs\_perm\_order \\
\forall x : Cvs\_Perm \bullet (x \neq cvs\_public) \Rightarrow \\
\qquad (x, cvs\_public) \notin cvs\_perm\_order \\
\forall x : Cvs\_Perm \bullet \exists y : Cvs\_Perm \bullet \\
\qquad (x, y) \in cvs\_perm\_order
\end{array}$$

We turn now to the security entities and mechanisms of the CVS-Server and the clients: first we have to model the working copies and the repositories as maps assigning *abstract names Abs_Name* to *data Abs_Data* (both types are abstract in our model):

$[Abs\_Name, Abs\_Data]$
$ABS\_DATATAB \equiv Abs\_Name \nrightarrow Abs\_Data$

A CVS-Server provides an authorization table, which is used to control access within the repository. The server stores for each file in the repository the required permission. These tables are modeled as follows:

$$\begin{array}{ll}
AUTH\_TAB & \equiv Cvs\_Uid \times Cvs\_Passwd \\
& \nrightarrow Cvs\_Perm \\
ABS\_PERMTAB & \equiv Abs\_Name \nrightarrow Cvs\_Perm
\end{array}$$

Clients possess in their working also a table that assigns to each abstract name a CVS client and another map that associates each CVS client to the password previously used during the cvs login procedure. The interplay of these tables will be discussed later, here we just define them:

$$\begin{array}{ll}
ABS\_UIDTAB & \equiv Abs\_Name \nrightarrow Cvs\_Uid \\
PASSWD\_TAB & \equiv Cvs\_Uid \nrightarrow Cvs\_Passwd
\end{array}$$

3.2 The System Architecture

In this section, we give a brief overview on how we model the system architecture, which is divided into: the state of the server (including the repository), the state of the client (including the working copy), and a set of CVS operations working over both of them.

It is a distinguishing feature of a CVS-Server to store the authentication data inside the repository such that it can be accessed and modified with CVS operations. This implies certain formal prerequisites: we require an abstract name *abs_cvsauth* to be associated with data, which can be converted into an authentication table via a postulated function *authtab*.

$$\begin{array}{|l}
abs\_cvsauth : Abs\_Name \\
abs\_auth\_of : Abs\_Data \nrightarrow AUTH\_TAB \\
abs\_data\_of : AUTH\_TAB \rightarrow Abs\_Data \\
authtab : ABS\_DATATAB \nrightarrow AUTH\_TAB \\
\hline
\mathsf{ran}(abs\_data\_of) \subseteq \mathsf{dom}(abs\_auth\_of) \\
\forall x : \mathsf{dom}\ abs\_auth\_of \bullet \\
\qquad abs\_data\_of(abs\_auth\_of\ x) = x \\
\forall x : AUTH\_TAB \bullet \\
\qquad abs\_auth\_of(abs\_data\_of\ x) = x \\
\forall r : ABS\_DATATAB \bullet abs\_cvsauth \in \mathsf{dom}(r) \Rightarrow \\
\qquad authtab(r) = abs\_auth\_of(r\ abs\_cvsauth)
\end{array}$$

Modeling the server's state as a Z schema is straight forward. The state contains the repository *rep* and the map *rep_permtab* containing the required permissions for each file. Accessing the authentication table inside *rep* will require to have the role *cvs_adm*. *RepositoryState* is modeled as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RepositoryState}}\\
rep : ABS\_DATATAB\\
rep\_permtab : ABS\_PERMTAB\\
\hline
abs\_cvsauth \in \mathsf{dom}\, rep\\
\mathsf{dom}\, rep = \mathsf{dom}\, rep\_permtab\\
rep\_permtab(abs\_cvsauth) = cvs\_adm\\
rep(abs\_cvsauth) \in \mathsf{dom}\, abs\_auth\_of
\end{array}
$$

The state of the client component contains the working copy *wc*, the *wc_uidtab* assigning a CVS client to each file and a password table *abs_passwd* with credentials (passwords) used in previous CVS login operations (*abs_passwd* models the file .cvspass). Thus, for any data in the working copy and whenever an access to it may be processed, an individual role may be generated and validated by the server with respect to its current repository state. Further, there is a set of abstract names *wfiles* which is used as filter in update and commit operations. This filter corresponds to the concept of the *working directory* in the implementation, i.e. the effects of these operations are restricted to files stored within the working directory:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClientState}}\\
wc : ABS\_DATATAB\\
wc\_uidtab : ABS\_UIDTAB\\
abs\_passwd : PASSWD\_TAB\\
wfiles : \mathbb{P}\, Abs\_Name\\
\hline
\end{array}
$$

In what follows, we define the abstract CVS operations that model combined state transitions of the client and the repository. Due to space reasons, we only present login and commit.

The login operation simply stores the authentication data on the client-side. This is used to authenticate a CVS user for a permissions of the client. The Δ and Ξ notation is used in Z to import the schemas in two variants; one variant as a copy, the other by replacing all variables by corresponding stroked variables (e.g. wc') describing the successor state. The Ξ also introduces equalities enforcing that the components of the previous state are equal to the post state components.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{abs\_login}}\\
\Delta ClientState\\
\Xi RepositoryState\\
passwd? : Cvs\_Passwd\\
uid? : Cvs\_Uid\\
\hline
(uid?, passwd?) \in \mathsf{dom}(authtab\; rep)\\
abs\_passwd' = abs\_passwd \oplus \{uid? \mapsto passwd?\}\\
wc' = wc\\
wc\_uidtab' = wc\_uidtab\\
wfiles' = wfiles
\end{array}
$$

The commit (ci) operation usually takes a set of files as arguments (here denoted by *files?*). The case that no arguments may be passed is modeled by the possibility to set *files?* to the set of all files *ABS_NAME*.

Now we address the core of our hierarchic RBAC model of the system architecture, the *has_access* predicate. As a prerequisite, we define the shortcut is_valid_in for checking that a CVS client, together with a credential (password), represents a valid role with respect to the current repository:

$$
\begin{array}{|l}
\_\ \mathsf{is\_valid\_in}\ \_ : (Cvs\_Uid \times Cvs\_Passwd)\\
\qquad\qquad\qquad \leftrightarrow ABS\_DATATAB\\
\hline
\forall\, role : Cvs\_Uid;\ pwd : Cvs\_Passwd;\\
\qquad rep : ABS\_DATATAB \bullet\\
(role, pwd)\ \mathsf{is\_valid\_in}\ rep\\
\qquad \Leftrightarrow (role, pwd) \in \mathsf{dom}(authtab(rep))
\end{array}
$$

Further, the *has_access* predicate ensures is_valid_in and that the permissions resulting from these credentials are sufficient to access the requested file according to the role hierarchy:

$$
\begin{array}{|l}
has\_access\ \_ : \mathbb{P}(ABS\_PERMTAB\\
\qquad\qquad \times ABS\_DATATAB \times PASSWD\_TAB\\
\qquad\qquad \times Abs\_Name \times Cvs\_Uid)\\
\hline
\forall\, rep\_pt : ABS\_PERMTAB;\ rep : ABS\_DATATAB;\\
\qquad pwtb : PASSWD\_TAB;\ file : Abs\_Name;\\
\qquad role : Cvs\_Uid \bullet\\
has\_access(rep\_pt, rep, pwtb, file, role)\\
\Leftrightarrow (role, pwtb(role))\ \mathsf{is\_valid\_in}\ rep\\
\qquad \wedge\ (rep\_pt(file), authtab(rep)(role, pwtb(role)))\\
\qquad\qquad \in cvs\_perm\_order
\end{array}
$$

The commit operation consists of the construction of a new repository rep' and a new table with required permissions $rep\_permtab'$ which were constructed via the override operator $\oplus$ from previous states of these tables. For rep', three cases can be distinguished: (i) either a file in the repository does not occur in the working copy, then it is unchanged, or (ii) it occurs in the working copy but not in the repository, then it is copied provided a valid permission is available in the *wc_uid_tab* of the working copy, or (iii) the file exists both in working copy and repository, then the working copy file overrides the repository file whenever the client has access:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{abs\_ci}}\\
\Xi ClientState\\
\Delta RepositoryState\\
files? : \mathbb{P}\, Abs\_Name\\
\hline
(wfiles \cap files?) \subseteq \mathsf{dom}\, wc\\
rep' = rep \oplus (\{n : wfiles \cap files? \mid n \notin \mathsf{dom}\, rep\\
\qquad \wedge\ n \in \mathsf{dom}\, wc\_uidtab\\
\qquad (wc\_uidtab(n), abs\_passwd(wc\_uidtab\; n))\\
\qquad\qquad \mathsf{is\_valid\_in}\ rep\} \lhd wc)\\
\qquad\quad \oplus(\{n : wfiles \cap files? \mid n \in \mathsf{dom}\, rep\\
\qquad\qquad \wedge\ n \in \mathsf{dom}\, wc\_uidtab\\
\qquad\qquad \wedge\ has\_access(rep\_permtab, rep,\\
\qquad\qquad\qquad abs\_passwd, n, wc\_uidtab(n))\\
\qquad\qquad \} \lhd wc)\\
rep\_permtab' = rep\_permtab \oplus \{n : wfiles \cap files? \mid\\
\qquad n \notin \mathsf{dom}\, rep \wedge n \in \mathsf{dom}\, wc\_uidtab\\
\qquad \wedge\ (wc\_uidtab(n), abs\_passwd(wc\_uidtab\; n))\\
\qquad\quad \in \mathsf{dom}(authtab\; rep) \bullet\\
\qquad\qquad n \mapsto authtab(rep)(wc\_uidtab(n),\\
\qquad\qquad\qquad abs\_passwd(wc\_uidtab\; n))\}
\end{array}
$$

The table $rep\_permtab'$ is extended by permissions for files that are new in the repository (based on the permissions used for committing these files). Further, the table wc_uid_tab is updated by the add operation, which we omit here.

In addition to these abstract models of the CVS operations, we provide a modify operation which explicitly models interactions of users with their files via modifying the files of the working copy of the client state.

3.3 The Implementation Architecture

The implementation architecture of CVS-Server is intended to model realistically the security mechanisms used to achieve the security goals formalized in the previous system architecture. Therefore, it captures the relevant operating system environment methods, i.e. POSIX methods in our case, for accessing files and changing their access attributes. We derived our POSIX model by formalizing the specification documents [17] and detailed system descriptions [5] and by validating it by carefully chosen tests and by inspections of critical parts of the system sources. In this POSIX model, the *CVS Filesystem* will be embedded, i.e. a repository is described as some area in the filesystem, where file attributes are set in a suitable way.

3.3.1 Modeling Basic Data Structures

We declare basic abstract sorts for POSIX user IDs, group IDs, data (file contents left abstract in this model), elementary filenames and file paths.

$$[Uid, Gid, Data, Name]$$
$$Path \equiv \mathrm{seq}\, Name$$

We assume a static table *groups* that assigns to each user a set of groups he belongs to. We also describe a special user ID *root*, modeling the system administrator. As we will show later, all security goals can only be achieved for all users except *root*, because root is allowed to do (almost) everything.

$$\begin{array}{|l} groups : Uid \rightarrow \mathbb{P}\, Gid \\ root : Uid \end{array}$$

3.3.2 Modeling the POSIX Filesystem Access Control

Within POSIX, every file belongs to a unique pair of owner (user) and group, and file access is divided into access by the *user* (owner), the *group* or *other* (world). The POSIX *discretionary access control* (DAC) distinguishes access for reading (r), writing (w), and executing (x). We also model the "set group id" (sg) on directories, which affects the default group of newly created files within that directory (see [5] for more technical details about the Unix/POSIX DAC):

$$Perm ::= ru \mid wu \mid xu \mid rg \mid wg \mid xg \mid ro \mid wo \mid xo \mid sg$$

The filesystem consists of a map from a file path to file content (which is either *Data* for regular files or *Unit* for directories[2]) and of file attributes (assigning to each file or directory the permissions[3], the user ID of the owner and the group it belongs to). Our concept of file attributes may easily be extended by adding new components to its records.

$$\begin{array}{l} Unit ::= Nil \\ FILESYS\_TAB \equiv Path \nrightarrow (Data + Unit) \\ FILEATTR \equiv [perm : \mathbb{P}\, Perm;\ uid : Uid;\ gid : Gid] \\ FILEATTR\_TAB \equiv Path \nrightarrow FILEATTR \end{array}$$

We use type sums for modeling the $FILESYS\_TAB$ which are not part of the Z standard. Type sums can simulate enumerations in Z free type definitions on the fly. The two functions $Inl : X \rightarrow (X + Y)$ and $Inr : Y \rightarrow (X + Y)$ are provided for building type sums.

For testing if a directory contains a specific entry (either a file or a directory) we provide the function is_in. Further, we provide functions that test for regular files (is_file_in) and for directories (is_dir_in); their definitions are straight forward:

$$\begin{array}{|l} \_\ \mathsf{is\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit)) \\ \_\ \mathsf{is\_dir\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit)) \\ \_\ \mathsf{is\_file\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit)) \\ \hline \forall fs : (Path \nrightarrow (Data + Unit));\ f : Path \bullet \\ \quad (f\ \mathsf{is\_in}\ fs) \Leftrightarrow f \in \mathrm{dom}\, fs \\ \forall fs : (Path \nrightarrow (Data + Unit));\ d : Path \bullet \\ \quad (d\ \mathsf{is\_dir\_in}\ fs) \Leftrightarrow (d\ \mathsf{is\_in}\ fs) \\ \quad \wedge (\exists\, u : Unit \bullet fs(d) = Inr(u)) \\ \forall fs : (Path \nrightarrow (Data + Unit));\ f : Path \bullet \\ \quad (f\ \mathsf{is\_file\_in}\ fs) \Leftrightarrow (f\ \mathsf{is\_in}\ fs) \wedge \neg\, (f\ \mathsf{is\_dir\_in}\ fs) \end{array}$$

At this point we are ready to model the filesystem state, which mainly describes the map of (name) paths to their attributes. As mentioned, we require that all defined paths must be "prefix-closed", i.e. all prefix paths must be defined in the filesystem (thus constituting a tree) and point to directories.

$$\begin{array}{|l} \textit{FileSystem} \\ \hline files : FILESYS\_TAB \\ attributes : FILEATTR\_TAB \\ \hline \forall p : \mathrm{dom}\, files \bullet (p = \langle\rangle) \\ \quad \vee (front(p)\ \mathsf{is\_dir\_in}\ files) \\ \mathrm{dom}\, files = \mathrm{dom}\, attributes \\ \hline \end{array}$$

In addition to the filesystem state, we introduce a state schema *ProcessState* for client related information,

[2] We do not consider *special files*, like devices, named pipes or process files.

[3] The terms *attributes* and *permissions* are used interchangeably.

namely the current user and group ID, the client's umask (which is used to set the initial file attributes on new files) and current working directory (*wdir*). The working directory is often used as an implicit parameter to filesystem and CVS operations:

ProcessState

$$\begin{array}{l} uid : Uid \\ gid : Gid \\ umask : \mathbb{P}(Perm \setminus \{sg\}) \\ wdir : Path \end{array}$$

As a prerequisite for describing functions that do modifications on the file system, we need to model the POSIX DAC in detail. Therefore we first introduce a function *has_attrib*, which decides whether the attributes (read, write and execute) of a file are set with respect to a specific user (and the groups he is a member of). Within this function, a crucial detail of the POSIX access model is formalized, namely that file access is checked by *sequentially* testing the following conditions (leading to an overall failure if the first condition fails):

1. If the user owns the file, he can only access the file if the access attributes for users grant access.
2. If the user is a member of the group owning the file, he can only access the file if the access attributes for the group grant access.
3. Last, the access attributes for others are checked.

These requirements may lead to some unexpected consequences, e.g. assume a user u being member of the group g and owner of a file with the permissions $\langle\!| \, perm == \{rg, ro\}, uid == u, gid == g \, |\!\rangle$. Curiously, file access will be denied for him, while granted for all others in his group, because the rights specified for the user precede the rights given for the group.

$$\begin{array}{l} has\_attrib\,\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB \\ \qquad\qquad\qquad \times Perm \times Perm \times Perm) \\ \hline \forall\, uid : Uid;\ fa : FILEATTR\_TAB; \\ \quad pu, pg, po : Perm \bullet \forall\, p : \mathrm{dom}(fa) \bullet \\ has\_attrib(uid, p, fa, pu, pg, po) \Leftrightarrow \\ \quad ((uid = root) \lor \\ \quad (\forall\, m : \mathbb{P}\, Perm;\ diruid : Uid;\ dirgid : Gid \mid \\ \qquad \langle\!|\, perm \equiv m, uid \equiv diruid, gid \equiv dirgid \,|\!\rangle \\ \qquad\quad = fa(p) \bullet \\ \qquad\quad (diruid = uid \land pu \in m) \lor \\ \qquad\quad (diruid \neq uid \land dirgid \in groups(uid) \\ \qquad\quad \land pg \in m) \lor \\ \qquad\quad (diruid \neq uid \land dirgid \notin groups(uid) \\ \qquad\quad \land po \in m))) \end{array}$$

Based on *has_attrib* we introduce shortcuts for checking read, write and execute attributes (e.g. *has_w_attrib*) of files and directories as well as definitions for checking the read, write and execute access (e.g. *has_w_access*).

$$\begin{array}{l} has\_w\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ has\_r\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ has\_x\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ \hline \forall\, uid : Uid;\ p : Path;\ fa : FILEATTR\_TAB \bullet \\ \quad has\_w\_attrib(uid, p, fa) \\ \qquad \Leftrightarrow has\_attrib(uid, p, fa, wu, wg, wo) \\ \quad \ldots \end{array}$$

$$\begin{array}{l} has\_w\_access\,\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ has\_r\_access\,\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ has\_x\_access\,\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\ \hline \forall\, uid : Uid;\ p : Path;\ fa : FILEATTR\_TAB \bullet \\ \quad has\_w\_access(uid, p, fa) \Leftrightarrow \\ \qquad (\forall\, pref : Path \mid pref\ \mathsf{prefix}\ (front\, p) \bullet \\ \qquad has\_x\_attrib(uid, pref, fa)) \\ \qquad \land\ has\_w\_attrib(uid, front\, p, fa) \\ \quad \ldots \end{array}$$

As an example for our approach to specify POSIX operations, we present the (shortened) file remove specification [17], which corresponds to unlink():

The unlink() function shall fail and shall not unlink the file if:

- *A component of path does not name an existing file or path is an empty string.*
- *Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.*

This text is formalized by a Z operation schema *rm* as follows: The first condition in the body is common for most filesystem operations and requires the path of the file must be a valid one in the filesystem table. The second condition requires that the client has write permissions on the file and the working directory ("*the directory containing the directory entry to be removed*"), which is checked via the *has_w_access* predicate:

rm

$$\begin{array}{l} \Delta FileSystem \\ \Xi ProcessState \\ u? : Name \\ \hline (wdir \frown \langle u? \rangle)\ \mathsf{is\_file\_in}\ files \\ has\_w\_access(uid, wdir, attributes) \\ \quad \land\ has\_w\_access(uid, wdir \frown \langle u? \rangle, attributes) \\ files' = \{wdir \frown \langle u? \rangle\} \lhd files \\ \quad \land\ attributes' = attributes \end{array}$$

The definitions for the remaining filesystem operations are similar, see [1] for details.

3.3.3 Mapping CVS Access Control onto POSIX DAC

We turn now to a crucial aspect of the implementation of the security goals by security mechanisms provided from standard POSIX DAC: any CVS role will be mapped to a particular pair of system *owner* and a set of system *group*s. This mapping has the consequence of an inheritance mechanism for generating default roles when creating new objects in the repository. Additionally, there is a mechanisms to "down-scale" and "up-scale" the permissions in the repository for the CVS administrator (not described here).

For every CVS operation, the server determines the CVS role according to the client's CVS ID and password. These roles are then mapped to POSIX user and group IDs, and these are compared to the file attributes of the files and directories the operations operates on. This translation is done by the two functions *cvsperm2uid* and *cvsperm2gid*.

$$
\begin{array}{|l}
\mathit{cvsperm2uid} : \mathit{Cvs\_Perm} \rightarrowtail \mathit{Uid} \\
\mathit{cvsperm2gid} : \mathit{Cvs\_Perm} \rightarrowtail \mathit{Gid} \\
\mathit{users} : \mathbb{P}\,\mathit{Uid} \\
\hline
\mathit{root} \notin \mathsf{ran}\ \mathit{cvsperm2uid} \\
\mathsf{ran}\ \mathit{cvsperm2uid} \cap \mathit{users} = \varnothing \\
\mathsf{ran}\ \mathit{cvsperm2gid} \cap \bigcup\{x : \mathit{users} \bullet \mathit{groups}(x)\} = \varnothing \\
\mathsf{ran}\ \mathit{cvsperm2uid} \lhd \mathit{groups} = \{x : \mathit{Cvs\_Perm} \bullet \\
\qquad \mathit{cvsperm2uid}\ x \mapsto \\
\qquad\qquad \{c : \mathit{Cvs\_Perm} \mid (c, x) \in \mathit{cvs\_perm\_order} \bullet \\
\qquad\qquad\qquad \mathit{cvsperm2gid}\ c\}\}
\end{array}
$$

It is important to notice that CVS IDs (*Cvs_Uid*) are independent of POSIX IDs (*Uid*) and that the POSIX IDs which are used by CVS are disjoint from "normal" POSIX user IDs, i.e. it is impossible to login with such a special POSIX ID.

From these distinctness constraints follows that the POSIX system administrator and the CVS administrator may be different. Moreover, we require that the group table (administrated by the system administrator and nobody else) is compatible with *cvs_perm_order*. These requirements have to be assured during installation of a CVS server.

The CVS repository is a subtree of the normal filesystem; its root is denoted by the absolute path *cvs_rep* and all paths inside the repository are relative to the root *cvs_rep*. Further, the administrative files of CVS are stored in the *CVSROOT* directory, which is a subdirectory of *cvs_rep*, and the file that contains all authentication information is called *cvsauth* and is located inside *CVSROOT*.

$$
\begin{array}{|l}
\mathit{cvs\_rep} : \mathit{Path} \\
\mathit{CVSROOT} : \mathit{Name} \\
\mathit{cvsauth} : \mathit{Name} \\
\mathit{auth\_of} : \mathit{Data} \nrightarrow \mathit{AUTH\_TAB} \\
\mathit{data\_of} : \mathit{AUTH\_TAB} \rightarrow \mathit{Data} \\
\hline
\mathsf{ran}\ \mathit{data\_of} \subseteq \mathsf{dom}\ \mathit{auth\_of} \\
\forall x : \mathsf{dom}\ \mathit{auth\_of} \bullet \mathit{data\_of}(\mathit{auth\_of}\ x) = x \\
\forall x : \mathit{AUTH\_TAB} \bullet \mathit{auth\_of}(\mathit{data\_of}\ x) = x
\end{array}
$$

3.3.4 Modeling the CVS Filesystem

A major design decision for our specification is to enrich the *FileSystem* state by new state components relevant to CVS, or more precisely, the combined client/server component of CVS. In CVS, working copies contain specific attributes assigned to the files; we restrict ourselves to security relevant attributes, i.e. the CVS client ID and password, and the path *rep* where the file is located in the repository. This information is kept in an own table implicitly associated to the working copies.

$$
\begin{array}{ll}
\mathit{CVS\_ATTR} & \equiv [\mathit{rep} : \mathit{Path};\ \mathit{f\_uid} : \mathit{Cvs\_Uid}] \\
\mathit{CVS\_ATTR\_TAB} & \equiv \mathit{Path} \nrightarrow \mathit{CVS\_ATTR}
\end{array}
$$

Due to the space reasons, we only show some requirements of the combined POSIX and CVS filesystem:

- working copies and the repository are distinct areas of the filesystem.
- the repository contains a special directory that contains the administrative data of CVS. Certain restrictive access permissions must be ensured to this directory and its contents to preserve the system integrity.
- requirements on file attributes within the repository:
  - since the owners of files must be POSIX user IDs that are disjoint from "regular" POSIX user IDs, and the group IDs must be legal with respect to the CVS role hierarchy. This guarantees that regular users only have the rights described by the file attributes for others. Thus, our initial invariant for the base directory of the repository implies that such a user cannot do anything, using only POSIX operations, within the repository.
  - read, write and execute permissions are the same for user and group. Together with our group setup this ensures that the initial CVS role and all roles with higher precedence have the same rights to access that file.

  These invariants are formally described in the axiomatic definition:

$$
\begin{array}{|l}
\mathit{attr\_in\_rep}\ \_ : \mathbb{P}\,\mathit{FileSystem} \\
\mathit{attr\_in\_root}\ \_ : \mathbb{P}\,\mathit{FileSystem} \\
\mathit{attr\_outside\_root}\ \_ : \mathbb{P}\,\mathit{FileSystem} \\
\hline
\forall \mathit{fs} : \mathit{FileSystem} \bullet \mathit{attr\_in\_rep}(\mathit{fs}) \Leftrightarrow \\
\qquad (\forall p : \mathsf{dom}\,\mathit{fs.files} \mid (\mathit{cvs\_rep}\ \mathsf{prefix}\ p) \bullet \\
\qquad (((\mathit{fs.attributes}\,p).\mathit{uid}) \in \mathsf{ran}\ \mathit{cvsperm2uid} \\
\qquad \wedge\ ((\mathit{fs.attributes}\,p).\mathit{gid}) \\
\qquad\qquad \in \mathit{groups}((\mathit{fs.attributes}\,p).\mathit{uid}) \wedge \\
\qquad (\mathit{ru} \in ((\mathit{fs.attributes}\,p).\mathit{perm}) \Leftrightarrow \mathit{rg} \\
\qquad\qquad \in (\mathit{fs.attributes}\,p).\mathit{perm}) \wedge \\
\qquad (\mathit{wu} \in ((\mathit{fs.attributes}\,p).\mathit{perm}) \Leftrightarrow \mathit{wg} \\
\qquad\qquad \in (\mathit{fs.attributes}\,p).\mathit{perm}) \wedge \\
\qquad (\mathit{xu} \in ((\mathit{fs.attributes}\,p).\mathit{perm}) \Leftrightarrow \mathit{xg} \\
\qquad\qquad \in (\mathit{fs.attributes}\,p).\mathit{perm}))) \\
\qquad \ldots
\end{array}
$$

$$
\begin{array}{l}
\textit{cvs\_ci} \\
\hline
\Delta \textit{Cvs\_FileSystem} \\
\Xi \textit{ProcessState} \\
p? : \textit{Path} \\
\hline
\textit{has\_r\_access}(\textit{uid}, \textit{wdir} \frown p?, \textit{attributes}) \\
\textit{wdir} \in \mathsf{dom}\, \textit{wcs\_attributes} \\
\textit{files}' = \textit{files} \oplus \{q : \textit{rep\_access}(\theta \textit{Cvs\_FileSystem})((\textit{wcs\_attributes}\, \textit{wdir}).\textit{rep} \frown p?) \mid \\
\qquad\qquad \textit{has\_r\_access}(\textit{uid}, \textit{wdir} \frown \textit{cutPath}(q, (\textit{wcs\_attributes}\, \textit{wdir}).\textit{rep}), \textit{attributes}) \bullet \\
\qquad\qquad\qquad \textit{cvs\_rep} \frown q \mapsto \textit{files}(\textit{wdir} \frown \textit{cutPath}(q, (\textit{wcs\_attributes}\, \textit{wdir}).\textit{rep}))\} \\
\textit{attributes}' = \textit{attributes} \oplus \{q : \textit{rep\_access}(\theta \textit{Cvs\_FileSystem})((\textit{wcs\_attributes}\, \textit{wdir}).\textit{rep} \frown p?) \mid \\
\qquad\qquad \textit{has\_r\_access}(\textit{uid}, \textit{wdir} \frown \textit{cutPath}(q, (\textit{wcs\_attributes}\, \textit{wdir}).\textit{rep}), \textit{attributes}) \bullet \\
\qquad\qquad\qquad \textit{cvs\_rep} \frown q \mapsto \langle\!|\ \textit{perm} \equiv \{\textit{ru}, \textit{rg}\}, \\
\qquad\qquad\qquad\qquad \textit{uid} \equiv \textit{cvsperm2uid}(\textit{get\_auth\_tab}(\textit{files})((\textit{wcs\_attributes}\, q).\textit{f\_uid}, \\
\qquad\qquad\qquad\qquad\qquad \textit{cvs\_passwd}((\textit{wcs\_attributes}\, q).\textit{f\_uid}))), \\
\qquad\qquad\qquad\qquad \textit{gid} \equiv \textit{cvsperm2gid}(\textit{get\_auth\_tab}(\textit{files})((\textit{wcs\_attributes}\, q).\textit{f\_uid}, \\
\qquad\qquad\qquad\qquad\qquad \textit{cvs\_passwd}((\textit{wcs\_attributes}\, q).\textit{f\_uid}))) \\
\qquad\qquad\qquad\qquad |\!\rangle\} \\
\textit{wcs\_attributes}' = \textit{wcs\_attributes} \\
\textit{cvs\_passwd}' = \textit{cvs\_passwd}
\end{array}
$$

Fig. 4: The specification of the commit command (implementation architecture)

We turn now to a formal description of the repository *within* the filesystem. This invariant of the system is captured in the state schema *Cvs_FileSystem*:

$$
\begin{array}{l}
\textit{Cvs\_FileSystem} \\
\hline
\textit{FileSystem};\ \textit{wcs\_attributes} : \textit{CVS\_ATTR\_TAB} \\
\textit{cvs\_passwd} : \textit{PASSWD\_TAB} \\
\hline
\mathsf{dom}\, \textit{wcs\_attributes} \subseteq \mathsf{dom}\, \textit{files} \\
(\textit{cvs\_rep} \frown \langle \textit{CVSROOT}, \textit{cvsauth} \rangle\ \mathsf{is\_file\_in}\ \textit{files}) \\
\textit{attributes}(\textit{cvs\_rep}) = \\
\langle\!|\ \textit{perm} \equiv \{\textit{ru}, \textit{wu}, \textit{xu}, \textit{xg}, \textit{sg}\}, \\
\quad \textit{uid} \equiv \textit{cvsperm2uid}(\textit{cvs\_adm}), \\
\quad \textit{gid} \equiv \textit{cvsperm2gid}(\textit{cvs\_public})\ |\!\rangle \\
\textit{attr\_in\_rep}(\theta \textit{FileSystem}) \\
((\textit{attributes}(\textit{cvs\_rep} \frown \langle \textit{CVSROOT} \rangle)).\textit{gid}) \\
\qquad = \textit{cvsperm2gid}(\textit{cvs\_adm}) \\
\textit{attr\_in\_root}(\theta \textit{FileSystem}) \\
\qquad \wedge\ \textit{attr\_outside\_root}(\theta \textit{FileSystem})
\end{array}
$$

Additionally to *rep_attributes*, we impose similar requirements for the administrative area of the repository by the predicate *attr_in_root*. Further, we describe in the predicate *attr_outside_root* the requirements for the data in the repository, i.e. files that are subject to version control. Both axiomatic definitions are omitted here.

Now we have established a basis for the operations on the combined POSIX and CVS environment. As in Sec. 3.2, we present the login and commit operations in order to compare the two different architecture levels.

Before we describe the operations of the CVS-Server we need to model the access to the CVS authentication table (*get_auth_tab*) that is part of the $\textit{cvs\_rep} \frown \textit{CVSROOT}$ directory and underlies the standard access discipline of CVS-Server. In particular, the authentication table is only modifiable by the CVS administrator, but not by any other client of the system.

$$
\begin{array}{|l}
\textit{get\_auth\_tab} : \textit{FILESYS\_TAB} \rightarrow \textit{AUTH\_TAB} \\
\hline
\quad \ldots
\end{array}
$$

The login operation updates the variable *cvs_passwd*, provided that for the combination of user ID and password the authentication will succeed.

$$
\begin{array}{l}
\textit{cvs\_login} \\
\hline
\Delta \textit{Cvs\_FileSystem} \\
\Xi \textit{ProcessState} \\
\textit{cvs\_uid}? : \textit{Cvs\_Uid} \\
\textit{cvs\_pwd}? : \textit{Cvs\_Passwd} \\
\hline
(\textit{cvs\_uid}?, \textit{cvs\_pwd}?) \in \mathsf{dom}(\textit{get\_auth\_tab}\, \textit{files}) \\
\textit{cvs\_passwd}' = \textit{cvs\_passwd} \\
\qquad \oplus \{\textit{cvs\_uid}? \mapsto \textit{cvs\_pwd}?\} \\
\textit{wcs\_attributes}' = \textit{wcs\_attributes} \\
\theta \textit{FileSystem} = \theta(\textit{FileSystem})'
\end{array}
$$

In the commit operation, the current working directory *wdir* can be restricted by the parameter $p?$ to just one file or directory. All files below $p?$ for which the client has access will be committed. We use the function *cutPath* to remove a given prefix from a path.

$$
\begin{array}{|l}
\textit{cutPath} : (\textit{Path} \times \textit{Path}) \rightarrowtail \textit{Path} \\
\hline
\forall\, a, b, c : \textit{Path} \bullet \textit{cutPath}(a, b) = c \Leftrightarrow a = b \frown c
\end{array}
$$

In contrast to the system architecture specification we also must determine the POSIX file attributes of the files. The particularity of the update and the commit operation is the use of *rep_access* which computes the paths into the repository to which the client has read access according to his CVS role.

$$
\begin{array}{|l}
\mathit{rep\_access} : \mathit{Cvs\_FileSystem} \rightarrow \mathit{Path} \rightarrow \mathbb{P}\,\mathit{Path} \\
\hline
\forall\, \mathit{cfs} : \mathit{Cvs\_FileSystem};\ p : \mathit{Path} \bullet \\
\quad \mathit{rep\_access}(\mathit{cfs})(p) = \{ q : \mathit{Path} \mid p \ \mathsf{prefix}\ q \\
\qquad \wedge\ \mathit{cvs\_rep} \frown q \in \mathsf{dom}\ \mathit{cfs.files} \\
\qquad \wedge\ (\exists\, \mathit{idpwd} : \mathit{cfs.cvs\_passwd} \bullet \\
\qquad\quad \mathit{idpwd} \in \mathsf{dom}(\mathit{get\_auth\_tab}(\mathit{cfs.files})) \\
\qquad \wedge\ (\mathit{has\_r\_access}(\mathit{cvsperm2uid}(\\
\qquad\quad \mathit{get\_auth\_tab}(\mathit{cfs.files})(\mathit{idpwd})), \\
\qquad\quad \mathit{cvs\_rep} \frown q, \mathit{cfs.attributes}) \\
\qquad \vee\ (\mathit{has\_x\_access}(\mathit{cvsperm2uid}(\\
\qquad\quad \mathit{get\_auth\_tab}(\mathit{cfs.files})(\mathit{idpwd})), \\
\qquad\quad \mathit{cvs\_rep} \frown q, \mathit{cfs.attributes}) \\
\qquad \wedge\ \mathit{cvs\_rep} \frown q\ \mathsf{is\_dir\_in}\ \mathit{cfs.files})))\}
\end{array}
$$

The schema *cvs_ci* (see Fig. 4) models the commit command. We require that the client has read access for the file or directory in the current working directory and sufficiently high-ranked role to modify the repository.

4 Formal Analysis

A formal model, even if successfully type-checked, is in itself not a value of its own: it must be validated, e.g. by testing techniques or by formal proof activities as in our approach. In this section, we present a formal consistency check of the specifications, and we show that the implementation architecture is, in a formal sense, a refinement of the abstract system architecture. We specify and prove security properties of the type "no combination of user-commands will enable a user to write into the repository, except he has the required access rights".

4.1 Checking the Consistency

Two types of "sanity checks" are useful and have been carried out with HOL-Z [2] routinely:

– definedness checks for all applications of partial functions in their context, as undefined applications usually indicate that some part of the precondition of a schema context is missing, and
– checking the state invariant of all operation schemas; in particular, we require that in a schema, all syntactic preconditions (i.e. the conjuncts in the predicate part that contain occurrences of variables without stroke " ′ " and " ! " suffix) suffice to show that a successor state exists.

Violating these conditions does not result in logical inconsistencies but in unprovable statements or operation definitions with undesired semantical effects.

4.2 Establishing the Refinement

To prove that the concrete implementation architecture correctly implements the abstract system architecture, we have to define an abstraction schema R which relates the components of the abstract state to the components of the concrete state. In particular, we must map abstract names and data to paths and files in the sense of the POSIX filesystem, and the working copies and repositories of the abstract model must be related to certain areas of the filesystem, the authentication tables must be related, the user must not be *root* (the refinement simply does not work otherwise) and the file attributes in the concrete filesystem must be convertible along the mapping discussed in Sec.3.3.3.

Due to limited space, we will only show two constraints of R formally. As a prerequisite, let us define a function *Rname2path*, which maps abstract names, to file paths in the implementation model. One constraint is that *abs_cvsauth* is mapped to the right path and that the authentication tables in both models are equal:

$$
\begin{aligned}
\mathit{Rname2path}(\mathit{abs\_cvsauth}) &= \mathit{cvs\_rep} \\
&\qquad\qquad \frown\langle \mathit{CVSROOT}, \mathit{cvsauth}\rangle \\
\mathit{authtab}(\mathit{rep}) &= \mathit{get\_auth\_tab}(\mathit{files})
\end{aligned}
$$

The last constraint we present here enforces the abstract working copy to have a counterpart in the implementation working copy:

$$\mathit{Rname2path}(\!|\mathsf{dom}\ \mathit{wc}|\!) = \mathsf{dom}\ \mathit{wcs\_attributes}$$

To verify the refinement relation R, following Spivey in [16], we must prove two refinement conditions for each operation on the abstract state and its corresponding operation on the concrete state: Condition (a) ensures that a concrete operation terminates whenever their corresponding abstract operation is guaranteed to terminate, condition (b) ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate.

As an example of the refinement, we show the instantiation of conditions (a) and (b) for the CVS login operation. The refinement conditions, though, as defined in [16], assume that both operations have the same input parameters, but since we define them differently in our two models, we introduce an additional schema *Asm*, which is used to insert further assumptions into the refinement proofs (the effect could also have been achieved by a suitable renaming):

$$
\begin{array}{|l}
\textit{— Asm —} \\
\mathit{passwd?}, \mathit{cvs\_pwd?} : \mathit{Cvs\_Passwd} \\
\mathit{uid?}, \mathit{cvs\_uid?} : \mathit{Cvs\_Uid} \\
\hline
\mathit{passwd?} = \mathit{cvs\_pwd?} \\
\mathit{uid?} = \mathit{cvs\_uid?} \\
\hline
\end{array}
$$

In the case of the login operation, these assumptions are simple since the parameters are of the same type but differ in name. Instantiating condition (a) and (b) for the login operation and adding the assumption schema *Asm* leads to the following two proof obligations:

$$\begin{array}{l}
login_a \equiv \forall\, ClientState;\ RepositoryState;\\
\quad ProcessState;\ Cvs\_FileSystem;\\
\quad passwd?, cvs\_pwd? : Cvs\_Passwd;\ uid?,\\
\quad cvs\_uid? : Cvs\_Uid \bullet\\
\qquad Asm \wedge \mathsf{pre}\ abs\_login \wedge R \Rightarrow \mathsf{pre}\ cvs\_login\\
login_b \equiv \forall\, ClientState;\ RepositoryState;\\
\quad ProcessState;\ Cvs\_FileSystem;\\
\quad ProcessState';\ Cvs\_FileSystem';\ passwd?,\\
\quad cvs\_pwd? : Cvs\_Passwd;\ uid?\\
\quad cvs\_uid? : Cvs\_Uid \bullet\\
\qquad Asm \wedge \mathsf{pre}\ abs\_login \wedge R \wedge cvs\_login\\
\qquad\quad \Rightarrow (\exists\, ClientState';\ RepositoryState' \bullet\\
\qquad\qquad R' \wedge abs\_login)
\end{array}$$

The obligations for the other operations are defined analogously. So far, we proved these obligations formally for the refinement of login, add and update. These proofs considerably helped us to identify subtle side-conditions in our model and thus to get our real CVS configuration "right".

4.3 Security Properties in Architecture Layers

Specifying the security properties motivates a Z-section for the system architecture and one for the implementation architecture, both containing a classical *behavioral* specification. In *SysArchSec* we investigate security properties of the system architecture. In *ImplArchSec* we investigate the same properties and additional ones that are specific to the implementation architecture.

4.3.1 The General Scheme of Security Properties

As an interface between the operation schemas of the two architecture layers and the behavioral part allowing to specify safety properties, we convert suitably restricted operation schemas of both system layers into explicit relations over the underlying state. The purpose of these restrictions is to provide a slot for side-conditions that are related to the security model and not the functional model described in the previous sections:

$$\begin{array}{rcl}
rop_1 & = & op_1 \wedge R_1\\
 & \dots & \\
rop_n & = & op_n \wedge R_n
\end{array}$$

where each rop_i represents the operation schema op_i constrained by the restriction schema R_i. Further the schema disjunction *step* represents the overall step relation of the system, which is converted into a transitively closed relation *trans*:

$$\begin{array}{l}
\text{step} \ = rop_1 \vee \ldots \vee rop_n\\
\text{trans} = \{step \mid (\theta state, \theta state')\}^*
\end{array}$$

In the literature, three types of properties can be distinguished: One may formalize properties over the *set of reachable states*, the *set of possible transitions* or the set of *possible sequences of states (traces)* of a system. While the first two types are only sufficient for classical safety invariants ("something bad will never happen"), the latter two allow for the specification of liveness properties ("eventually something good will happen"). The general scheme for properties over reachable states and possible transitions for safety properties and the schema for liveness properties looks as follows:

$$\begin{array}{l}
\text{SP}_{\text{RS}} \ = \forall\, \sigma : \text{trans}(\!|\text{init}|\!) \bullet P\sigma\\
\text{SP}_{\text{RT}} \ = \forall (\sigma, \sigma') : \text{init} \lhd \text{trans} \bullet P(\sigma, \sigma')\\
\text{LP}_{\text{RT}} \ = \forall (\sigma, \sigma') : \text{init} \lhd \text{trans} \bullet\\
\qquad\qquad \exists (\sigma'', \sigma''') : \text{trans} \bullet P(\sigma, \sigma', \sigma'', \sigma''')
\end{array}$$

Note that the reachable states are restricted via the existential image operator or the domain restriction to the states (respectively transitions) reachable from the set of initial states init.

4.3.2 An Instance of the General Scheme: *RBAC_write*

We will exemplify the scheme SP_{RT} for a crucial security property, namely "the user may write in the repository only if he has RBAC-permissions", which we will call *RBAC_write* in the following. Moreover, we will outline the inductive proof.

As a prerequisite, we postulate two arbitrary sets *knows* and *invents*; a client "knows" a set of pairs of roles and passwords, and "invents" only files from a given set of pairs from names to data. We assume *invents* to be closed under the *merge*-operation left abstract in our model.[4] On this basis, we define a *security policy*, by providing suitable restrictions op_iR for the system operations.[5] For example, we restrict the add operation to elements in the domain of the *invents*-set, we assume login being restricted to roles and passwords the client *knows* set, the modify operation and add being restricted to data the client "invents". While these restrictions have a more technical nature, a more conceptual restriction of *abs_ci* is as follows: in the role *cvs_adm*, the authentication table may only be altered such that rights are withdrawn, not granted. A typical restriction looks as follows:

$$\begin{array}{l}
abs\_loginR \equiv abs\_login\\
\qquad\qquad \wedge\ [cvs\_uid? : Cvs\_Uid;\ passwd? :\\
\qquad\qquad\quad Cvs\_Passwd \mid\\
\qquad\qquad\quad (cvs\_uid?, passwd?) \in Aknows]
\end{array}$$

[4] This is very similar to the concept of abstract crypt-functions and the closures *analz*, *synth* and *parts* in [10]; see discussion

[5] In practice, such security policies may be based on voluntary self-restrictions of users or enforced by administrative means.

Now we define the *step*-relation and its transitive closure of the system architecture layer:

$$\begin{aligned} step &\equiv abs\_loginR \lor abs\_addR \lor abs\_ciR \\ &\quad \lor abs\_modifyR \lor abs\_up \lor abs\_cd \\ AbsState &\equiv ClientState \land RepositoryState \\ trans &\equiv \{step \bullet (\theta AbsState, \theta AbsState')\}^* \end{aligned}$$

Finally, for constructing the proof goal *RBAC_write*, we instantiate the P in our schema SP_{RT} by:

$$\begin{array}{|l} rbac\_write\_ : \ldots \\ \hline \forall\, rep, rep' : ABS\_DATATAB; \\ \forall\, rptab' : ABS\_PERMTAB \bullet \\ \quad rbac\_write(rep, rep', rptab') \Leftrightarrow \\ \qquad (\forall f : \mathsf{dom}\ rep' \bullet \\ \qquad\quad (rep(f) \neq rep'(f) \\ \qquad\qquad \land (f, rep'(f)) \in invents) \\ \qquad\quad \Rightarrow (\exists\, m : knows \bullet \\ \qquad\quad (rptab'(f), authtab(rep')(m)) \\ \qquad\qquad \in cvs\_perm\_order)) \end{array}$$

This property reads as follows: whenever there is a change in the repository, and the changed file stems from the users *invents*-set, the user must have valid permissions according to the *RBAC*-model. We observe that *rbac_write* is *true* whenever the repository does not change, i.e. $rbac\_write(r, r, rt)$ holds.

4.3.3 A Proof-Outline

We will now present an exemplary proof (performed with HOL-Z) for *RBAC_write*. The initial proof goal stating that *RBAC_write* holds is refined by unfolding elementary definitions and simplification of Z notation to the following proof state:

$$\begin{aligned} [\![\sigma_0 &= (abs\_passwd, rep, rep\_permtab, wc, \\ &\qquad wc\_uidtab, wfiles); \\ \sigma_1 &= (abs\_passwd', rep', rep\_permtab', wc', \\ &\qquad wc\_uidtab', wfiles'); \\ &AbsState\sigma_0; \\ &AbsState\sigma_1; \\ &(\sigma_0, \sigma_1) : \{step \bullet (\sigma_0, \sigma_1)\}^* \\ ]\!] &\Longrightarrow \\ &\quad \sigma_0 : init \\ &\qquad \Rightarrow rbac\_write(rep, rep', rep\_permtab') \end{aligned}$$

Over this implication, we can now apply an induction rule over the transitive closure:

$$\frac{(a,b) \in r^* \qquad \begin{matrix}[x \in \mathsf{dom}\ r] \\ \vdots \\ P\,x\,x\end{matrix} \qquad \begin{matrix}[y \in \mathsf{ran}\ r] \\ \vdots \\ P\,y\,y\end{matrix} \qquad \begin{matrix}\begin{bmatrix} P\,x\,y, \\ (x,y) \in r^*, \\ (y,z) \in r \end{bmatrix} \\ \vdots \\ P\,x\,z\end{matrix}}{P\,a\,b}$$

This leads to two base cases and the induction step; both base cases are trivially true due to observation $rbac\_write(r, r, rt)$. Now the induction steps, which looks after some massage as follows, remains to show:

$$\begin{aligned} [\![&\ldots \\ &\sigma_{00} = (abs\_passwdx, repx, rep\_permtabx, wcx, \\ &\qquad wc\_uidtabx, wfilesx); \\ &\sigma_{01} = (abs\_passwdy, repy, rep\_permtaby, wcy, \\ &\qquad wc\_uidtaby, wfilesy); \\ &\sigma_{10} = (abs\_passwdz, repz, rep\_permtabz, wcz, \\ &\qquad wc\_uidtabz, wfilesz); \\ &\sigma_{00} : init \Rightarrow rbac\_write(repx, repy, rep\_permtaby); \\ &(\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*; \\ &(\sigma_{00}, \sigma_{10}) : \{step \bullet (\sigma_0, \sigma_1)\} \\ ]\!] &\Longrightarrow \sigma_{00} : init \\ &\quad \Rightarrow rbac\_write(repx, repz, rep\_permtabz) \end{aligned}$$

Here, the point of proof refinement is the assumption $(\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*$, which can be decomposed via the definition of *step* into a disjunction of schemas, where the input variables are existentially quantified. A generic tactic strips away the disjunctions and the existential quantifiers in the assumption. The result is a case split over all operations of the system architecture and universally quantified input parameters of all operations under consideration. Now, the observation is crucial that all operations except *abs_ci* do not change the repository, and, as a consequence of observation $rbac\_write(r, r, rt)$, imply the truth of the step. We can therefore focus on the case *abs_ci*:

$$\begin{aligned} [\![&\ldots \\ &(\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*; \\ &rbac\_write(repx, repy, rep\_permtaby); \\ &abs\_ci(abs\_passwdy, abs\_passwdz, filesq, repy, repz, \\ &\qquad rep\_permtaby, rep\_permtabz, wcy, wcz, \\ &\qquad wc\_uidtaby, wc\_uidtabz, wfilesy, wfilesz) \\ &(\sigma_{00}) : init; \\ ]\!] &\Longrightarrow rbac\_write(repx, repz, rep\_permtabz) \end{aligned}$$

This is the core part of an invariance proof: the system made a transition from an initial system state (with *repx*) to another (with *repy*) performing an arbitrary combination of operations *and* the system behaved well (i.e. $rbac\_write(repx, repy, rep\_permtaby)$). Now a commit operation (*abs_ci*) occurs, and the question is if the resulting state (with *repz*) will also fulfill our safety property.

The core of this subproof is, of course, a case distinction following the definition of *abs_ci* shown in Sec. 3.2: a file may be

1. in the repository *and not* in the working copy: then *abs_ci* will change nothing,

2. in the working copy *and not* in the repository: then *abs_ci* will only change the latter if the current credentials are is_valid_in which implies *write_correct* as the *rep_permtab* was changed accordingly,
3. both in the working copy *and* the repository: then *abs_ci* will only change the file in the repository if the current credentials allow for *has_access* which implies *write_correct*.

The interested reader may note that the overall scheme of the proof follows the structure of the general scheme of the property descriptions, which allows for automated tactic support that copes with Z-related technicalities, the choice of the inductions, the decomposition of the specification and the systematic derivation of state components remaining invariant. Obviously, there is a high potential of automation for this type of proofs, such that the proof developer may be guided rather automatically to the critical questions in the induction step.

4.3.4 Other Examples

The verification of the analogous property *RBAC_read* is straight forward; files in the working copy of a client are either invented by him (via the operation modify) or stem from the repository, where the client knows a password to obtain sufficient permissions.

An important, but quite obvious liveness property in the LP_{RT}-scheme is *RBAC_do_write*: Provided the client has access, it can change a file arbitrarily and perform operations leaving the repository changed accordingly; the proof immediately boils down to *abs_ci* which is designed to fulfill this property. At first sight, *RBAC_do_write* looks very similar to *RBAC_write*, however, note that both properties are independent: one could model an absolutely secure CVS-Server that never changes the repository. Such a model trivially fulfills *RBAC_write*, but is ruled out by *RBAC_do_write*.

So far, *RBAC_write* is formalized for a single-user client/server setting. Extending the analysis to a multi-user client/server model only requires simple modifications in the definition of the *step*-relation; via renaming of the working copies and the *invents* and *knows*-sets, instances of *abs_ci*, *abs_up* and *modify* for each client with individual working copy can be generated. Adding suitable restrictions (e.g. *invents* and *knows*-sets must be pairwise disjoint), *RBAC_write* and similar properties remain valid.

It is well-known that security properties are usually not preserved under refinement (see discussion later). The reason is that *implementing* one security architecture by another opens the door to *new* types of attacks on the implementation architecture that can be completely overlooked on the abstract level. For example, on the implementation architecture, it is possible to realize an attack on the repository by combinations POSIX commands such as *rm* and *setumask* etc (see Sec.3.3.2). In principle, our method can be applied for this type of analysis of the implementation architecture as well. In this setting, the *step*-relation and the *init* is defined as:

$$\begin{aligned} step_{impl} &\equiv rm \vee setumask \vee \cdots \vee chmod \\ &\qquad \vee cvs\_login \vee \cdots \vee cvs\_update \\ init_{impl} &\equiv ConcState \\ &\qquad \wedge [wcs\_attributes : CVS\_ATTR\_TAB \mid \\ &\qquad\quad wcs\_attributes = \varnothing] \end{aligned}$$

Although the proofs on the implementation architecture have the same structure as on the system architecture, they are far more complex since concepts such as paths, the distinction between files and directories, and their permissions are involved. Moreover, they require new side-conditions (for example, the refinement can only be established for the case that the user is not root) which were systematically introduced by the abstraction predicate R.

On the other hand, the higher degree of detail on the implementation architecture makes a formalization of new types of security properties possible: For example, since the crucial concept *directory* is present on the implementation level and since the existence of files can only be established by having access to all parent directories of a file, one can express confidentiality properties such as “the user can not find out that a file with name x exists in some directory of the repository” on this level.

5 Conclusion

5.1 Discussion

We demonstrate a method for analyzing the security in off-the-shelve system components thus made amenable to formal, machine-based analysis. The method proceeds as follows: First, specify the system architecture (as a framework for formal security properties), second, specify the implementation architecture (validated by inspecting informal specifications or testing code), third, set up the security technology mapping as a refinement, and fourth, prove refinements and security properties by mechanized proofs. The demonstration of the method follows a case study of a security problem for a real system, the CVS client/server architecture. We believe that the method is applicable for a wider range of problems such as mission-critical e-commerce applications or e-government applications.

The core of our approach is based on the presentation of the security technology mapping as data refinement problem. In general, it has been widely recognized that security properties can not be easily refined — actually, finding refinement notions that preserve security properties is a hot research topic [8,14]. However, standard refinement proof technology has still its value here since it checks that abstract security requirements are indeed achieved by a mapping to concrete security technology, and that implicit assumptions on this implementation have been made explicit. Against implementation specific attacks, we believe that specialized security property refinement techniques will be limited to restricted aspects. For this problem, in most cases the answer will be an analysis on the implementation level, possibly by reusing results from the abstract level.

In our approach, the analysis is based on interactive theorem proving while security analysis is often based on model-checking techniques for logics like LTL, the μ-calculus or process algebras like CSP. While these techniques offer a high degree of automation, they possess well-known and obvious limitations: the state-space must usually be finite and in practice be very small, and the analysis tends to be infeasible for many models, in particular those imposed by system specifications. As a consequence, proof engineers tend to develop oversimplified and unsystematically abstracted system models. In contrast, in our approach technical concerns like the size of the system state-space, aesthetic concerns like naturalness of the modeling (in our example, we use architectural modeling) or methodological needs like realistic treatments of system specifications do not represent fundamental obstacles to the analysis. In particular the latter paves the way for the reuse of standard system models like POSIX. Moreover, we have the full flexibility of Z and HOL to express security properties at need.

Our case study shows that the presented technology and method makes the treatment of complex security problems possible. Naturally, the question arises how long the formalization and the proof work took. This question is hard to answer, partly because the method and technical components had been developed during the project, partly because library theorems had to be proven, partly because some contributors needed time to learn Isabelle. The overall case study took about 18 man months, including the development of tool support. An substantial part (about six man months) was the formalization and testing (i.e. reverse engineering) of the system which was done by Achim Brucker and Burkhart Wolff. The proof work was done by Frank Rittinger, Harald Hiss and Burhart Wolff. Using our tool support, we estimate that someone with experience in theorem proving would be able to solve a similar task (specifying a similarly complex system and proving the core security properties) in less than 10 months. By improving the general technology (e.g. better front-ends and tatic support) a further speed-up by a factor two seems feasible.

5.2 Related Work

Sandhu and Ahn described in [12] a method for embedding role-based access control with the discretionary access control provided by standard Unix systems. Our model used this construction for providing the static roles, but extended it to a dynamic model.

Wenzel developed a specification of the basic Unix functionality, which was done in Isabelle/HOL and is part of the actual Isabelle [9] distribution. On the file system part, only a simple access model, not supporting groups and the concepts of set-id bits, is formalized.

Our behavioral analysis is based on the same foundations as Paulson's inductive method for protocol verification [10]. Beyond the obvious difference, that Paulson research focus is on analysis (the language of protocols is deliberately small and restrictive) and not on modeling, technical differences consist merely in some details: Paulson uses specialized induction schemes which are automatically derived from the protocol-rules; these are considered as inductive rules defining the set of system traces. In contrast, we use standard induction over transitive relations, which leads to a different organization of the specification and the security properties and leads to different tactic support.

5.3 Future Work

In our opinion, amazingly little work has been addressed to the specification of the POSIX interface; due to its often not intuitive features, its importance for security implementations and its high degree of reuse, this is a particularly rewarding target. We believe that our formalization is a starting point for a comprehensive, more complete model of the filesystem related commands.

Clearly, the formal proofs established so far do not represent a *complete* analysis of the (real) CVS-Server. Many more security properties can be formulated, and, by setting up different operation restrictions R_i, "best-practice" security policies can be formally investigated. Moreover, in order to make implementation level security analysis more feasible, it could be highly rewarding to develop techniques and methods to reuse (abstract) system level proofs on the more concrete levels.

References

1. A. D. Brucker, F. Rittinger, and B. Wolff. A CVS-Server security architecture — concepts and formal analysis. Technical Report 182, Albert-Ludwigs-Universität Freiburg, 2002.
2. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
3. P. Cederqvist et al. *Version Management with CVS*, 2000. `http://www.cvshome.org/docs/manual/`.
4. K. Fogel. *Open source development with CVS.* The Coriolis Group, 1999.
5. Æ. Frisch. *Essential System Administration.* O'Reilly, 1995.
6. D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.

7. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
8. J. Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (FME)*, volume 2021 of *LNCS*. Springer Verlag, 2001.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.
10. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
11. A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.
12. R. Sandhu and G.-J. Ahn. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *National Information Systems Security Conference*, pages 486–502, 1998.
13. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
14. T. Santen, M. Heisel, and A. Pfitzmann. Confidentiality-preserving refinement is compositional — sometimes. In *ESORICS*, volume 2502 of *LNCS*, pages 194–211. Springer Verlag, 2002.
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.
17. *The Single UNIX Specification Version 3*. The Open Group and IEEE, 2002. This standard superseeds the "Single UNIX Specification Version 2" (Unix 98) and the "IEEE Standard 1003.1-2001" (POSIX.1).
18. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996. `http://www.usingz.com/`.