Technical Report 182

A CVS-Server Security Architecture
—
Concepts and Formal Analysis

Achim D. Brucker Frank Rittinger Burkhart Wolff

December 28, 2002

Institut für Informatik
Albert–Ludwigs–Universität Freiburg
Georges-Köhler-Allee 52
D-79110 Freiburg, Germany
Tel: +49 (0)761 203-8240, Fax: +49 (0)761 203-8242

{brucker,rittinge,wolff}@informatik.uni-freiburg.de

http://www.informatik.uni-freiburg.de/~{brucker,rittinge,wolff}

Acknowledgements:
We would like to thank Nicole Rauch (Uni Kaiserslautern) for many valuable comments on a draft version of this paper. Harald Hiss provided in his Studienarbeit most of the consistency proof-work done in Chapter 6. Stefan Friedrich also made comments on an early version of the report and contributed to discussions and proofs to the library of HOL-Z.

The latest version of this technical report, of the CVS-server specification, and the proof scripts can be found in the example section of the HOL-Z distribution, which is available at: `http://wailoa.informatik.uni-freiburg.de/holz/`
The underlying implementation can be downloaded at: `http://www.informatik.uni-freiburg.de/~softech`

Abstract

We present a secure architecture of a CVS-server, its implementation (i.e. mainly its configuration) and its formal analysis. Our CVS-server uses cvsauth [39], that provides protection of passwords and protection of some internal data of the CVS repository. In contrast to other (security oriented) CVS-architectures, our approach makes it possible to CVS-server on an open filesystem, i.e. a filesystem where users can have direct access both by CVS-commands and by standard UNIX/POSIX commands such as `mv`. A key feature of our implementation is that it enforces a particular access control model, namely role-based access control (RBAC).

For our secure architecture of the CVS-server, we provide a formal specification and security analysis. This is based on a refinement, mapping a system architecture on an implementation architecture abstractly describing CVS in our implementation. The system architecture describes the abstract system operations including the desired access control model RBAC and serves as backbone to describe overall security requirements formally. The implementation architecture — to be seen as an abstract program — describes the security mechanisms on the UNIX/POSIX filesystem level, namely discretionary access control (DAC).

The purpose of the formal analysis of the secure CVS-server architecture is twofold: First, we describe our implementation, in particular the access control for our open architecture. Second, we provide a method to analyze formally security implementations (beyond the code level) for realistic applications in terms of off-the-shelf security technologies.

Keywords: security, formal methods, software architecture, Concurrent Versions System (CVS), Z, refinement

Contents

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 performed by highly distributed teams [7, 10, 8]. CVS provides a central data base (the *repository*) and means to synchronize local partial copies (the *working copies*) and their local modifications with the repository. CVS 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 relevant data is accessed and modified via standard CVS operations and, thus, access to objects may change dynamically.

The current standard CVS-server distribution (based on the `pserver`-protocol) is known to be quite insecure: since passwords are not encrypted during communication, usual password-sniffing attacks can be applied; moreover, a bug in some CVS-operations can allow an arbitrary user to modify the password authentication table. Further, any CVS user (with write access) can execute any program on a server with the userid he is authenticated to. This means he can send himself a command shell. Via buffer-overflow attacks, there is the inherent danger that the CVS-server can be crashed leaving a command shell under root permissions. Additionally, the repository in itself is not protected against direct user manipulations via file system operations.

Moreover, the current CVS-server distribution inherits the usual Unix permission concept — this may be too low-level and inadequate to cope with the demands of many user-groups and their work organization. In particular, we found it useful to provide a special configuration of the CVS-server that enforces a hierarchical *role based access control* model as described in [33]. This model associates to users one or more *roles* (e.g. project supervisor, test engineer, programmer, project member, etc) that are organized in a partial order. Further, *permissions* are associated to objects in the repository; a user may authenticate for a particular role in order to get appropriate permissions for object access.

In order to install the current CVS-server in security conscious way, the traditional architecture requires the server to run on a secured machine that has exclusive access to an own file-system hosting the repository. This *closed architecture* has a number of administrative and technical costs (additional hardware, efficiency).

In order to overcome some of these problems, we propose a number of improvements of the standard CVS-server, either on the level of its implementation (via patches), its configurations (i.e. the file system, including the initial state of the repository; this extends to group-tables, file permissions, scripts started when check in or check out data into the repository) or its architecture (i.e. the particular setup of a cvs-server and its configuration in a network). Our first main goal of our work is to provide a particular configuration of CVS-server that enforces a role-based access control security policy. Our second main goal is to develop an *open* CVS-server architecture, where the repository is part of the usual shared file-system of an intranet and the server a regular process on some machine in it. While an open architecture has undoubtedly a number of advantages, the correctness and trustworthiness of the security mechanisms becomes a major concern. In order to meet these concerns, we will present in this paper formal models and an analysis of the open CVS-server security architecture.

Our contribution in this paper is fivefold:

1. We provide a formal model of the CVS-server *system architecture*, that provides a *hierarchical* role based access control model for a fixed preconceived hierarchy of CVS-roles. Our CVS-server security architecture model contains checkout and checkin operations of CVS with respect to the access control policy resulting from the CVS-role hierarchy.

2. We provide a formal model of the *implementation architecture* based on the UNIX/-POSIX file-systems and its security mechanisms. Methodologically, the implementation architecture is a refinement of the system architecture.

3. We provide semi-formal and formal (machine-assisted) proofs for the consistency of the specification and for the refinement. Moreover, we provide proofs for the security properties both on the system and the implementation level of the architecture. The proofs take into account that on the implementation level UNIX-operations like `cd`, `chmod`, `chown` can be used within attacks.

4. We provide insight into the limits of refinement based, well-known abstraction techniques for the security analysis of real-world applications. In particular, realistic security analysis involves the formal analysis of *attacks against the implementation*; thus, models have to provide sufficient detail of the security mechanisms of the implementation architecture.

5. We provide a particular *configuration* of our CVS-server architecture, that implements our open, intranet compliant architecture. Our implementation is based on standard CVS extended by *cvsauth* that incorporates SSL/TSL-support [9] for secure communication of remote CVS access.

The overall benefit of our implemented architecture consists in security against most of the known attacks as listed above: Password sniffing is prevented by encryption, gaining root access by cvs-command attacks is prohibited, the danger of gaining root access by buffer-overflow attacks is limited to the authentication phase, which is the only remaining phase where the server runs under root. Additionally, administration of our CVS-server architecture is independent from system administration (except during initialization). And, last but not least, our open CVS-server can run on an arbitrary machine (say, the file server) with standard access to the file-system, which has become possible due to our formal modeling and analysis.

As formal specification language, we use the Z-Notation [19] for specifying our model. For theorem-proving, we compile our LaTeX-based specification via a own translator to HOL-Z [22] based on Isabelle98 [30]. Since Z is typed set-theory and as such semantically equivalent to higher-order logic (HOL; modulo minor differences w.r.t. polymorphism; see [34] for details.), we treat Z as a mere syntactic front-end to HOL. As such, Z offers a compact syntax geared towards data-modeling, which is normed [19] and for which many excellent text-books for software-engineers are available. For Z, an excellent LaTeX-based literate specification environment is available ([41]; with own type-checker etc.).

We proceeds as follows: After a discussion of our notion of architecture and architectural refinement, we come to the core chapters of this paper: the system architecture model (or: abstract layer of the refinement), the Unix model as infrastructure for the implementation architecture, and the implementation architecture itself. Further, we describe the refinement relation between these two, the security properties formalizable at the different layers and their analysis.

2 Excursion: Modeling the Architectures of CVS-server

In this section, we will discuss several alternatives of CVS-server architectures, namely the (traditional) closed CVS-server architecture and our proposed open CVS-server architecture. The closed CVS-server architecture naturally arises, if one wishes to install a CVS repository "out-of-the-box" but wants to prevent that users may directly access and destroy the repository; moreover, if one wants to implement a hierarchical access structure within the elements of the repository. As already mentioned, our "open approach" attempts to overcome some limitations of the "closed approach" at the price of *potentially* higher security risks - whose formal proof of irrelevance is the ultimate goal of this paper. The purpose of this section is to give a wider, architectural perspective of the system than the data-model used and analyzed in the next sections.

In order to describe the *architecture* of a system, it is common practice to use diagrams for this purpose, enriched by outlines of a formal specification in a behavioral specification language. Following the approaches of Garlan [13, 35], architectures are composed by *components* and *connectors*. Components are computational units that interact via connectors with each other; connectors can be remote procedure calls, communication protocols or access to shared variables. Garlan proposes a semantics for these notions in terms of the process-algebra CSP [31]. As a basis for discussions over the architecture of CVS–server, we will roughly follow this approach; in this setting, components are just processes, while connectors are also processes or particular parallel operators of CSP.

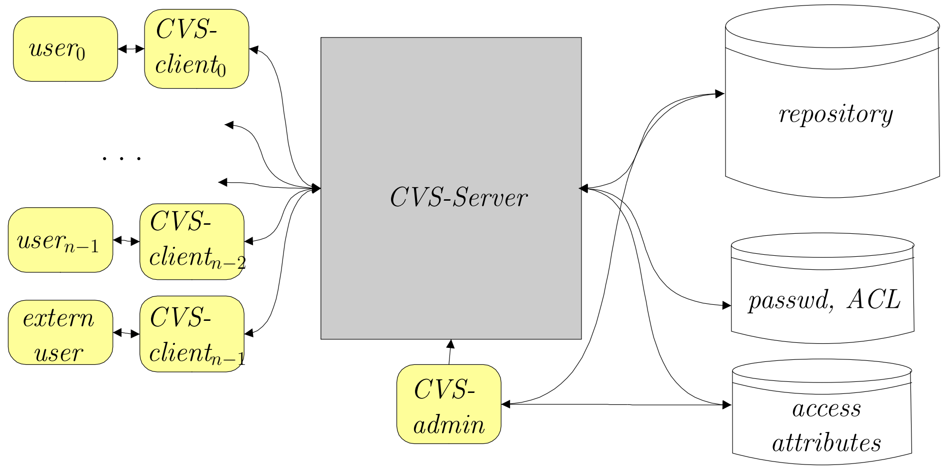
The *closed CVS-server architecture* is easily depicted as in Fig. 2.1. In this architecture, we have:

- two types of components, (one family of client components, one CVS-server).

- client components have two distinct roles (stduser, cvsadmin). The standard user gets access along a predefined hierarchy of access control rights for the commands checkin and checkout. The cvs-administrator may additionally change the state of the repository by changing the access permissions within the repository. The authentication is done in this model by the CVS login operation.

- the underlying assumption that the repository is internal in a particular closed machine with the repository inside.

Figure 2.1 An overview of the standard security architecture.



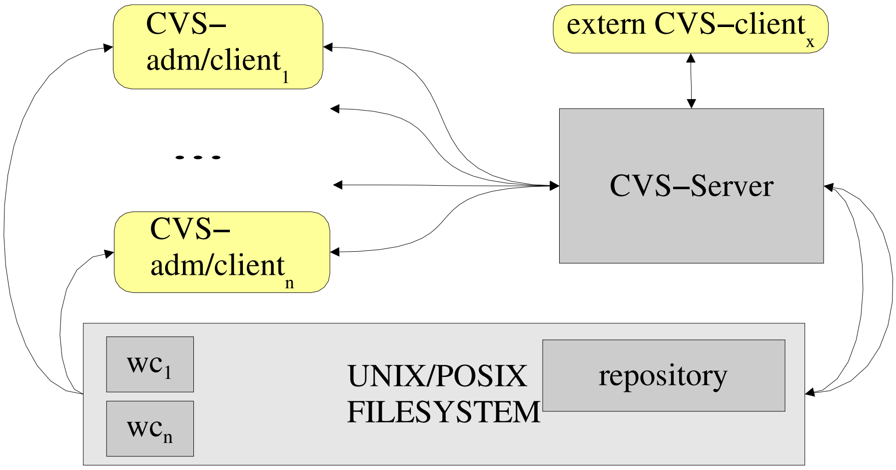
- no direct access to the repository by the user, only indirect access to the repository by the cvs administrator.
- direct access to the repository for the administrator.

In contrast the the setting discussed above, our *open CVS-server architecture* (see Fig. 2.2) is depicted as follows:

- three types components (a family of clients, a server, and the UNIX-filesystem).
- clients may have one of four roles (stduser, cvsadmin × user, sysadmin).
- the repository and the working copies are parts of the same underlying filesystem; this captures exactly the implementation reality.
- users and CVS-server have in principle equal rights to access the filesystem.
- direct access to the repository for the administrator
- the architectural view does not capture the fact that the users should have *different* access to the elements of the repository.

One could add a further type of client into this architecture, namely the "externclient" that has no direct access to the file system; this represents users that connect to the server without having an account in the local network the server operates in. But since this type of client represents a special case of the clients considered here (and can thus not introduce new complications with respect to security), we omit it for simplicity reasons. So far, we have presented architectures as a graph of of components (belonging to certain

Figure 2.2 An overview of the open security architecture.



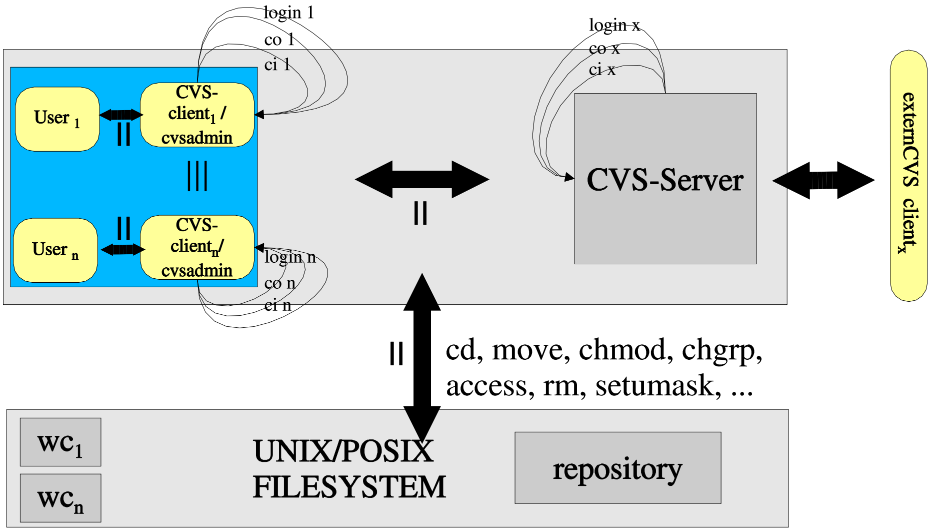
types), that have "roles" and that interact along "connectors" (depicted as arcs in the diagrams, whose exact nature is left open. A more formal description of the model that captures the communication events in more detail may look as shown in Fig. 2.3. Here, we have a collection of user-side CVS-client processes, that may perform cvs_login, cvs_ci (checkin), cvs_co (checkout), `cd`, `read`, `write`, `mv`, `mkdir`, `rmdir`, `rm`, `setumask`, `chgrp` and `chmod` commands (parameterized in this very simple behavioral model just by their user ID *uid*). We assume one particular uid "root". Further, we assume a collection of roles and authentication for them. Now our formal architecture model is defined as a collection of interleaved processes that can be stated as follows:

```
CLIENT(uid) = ⊓ role:ROLE,pwd:PWD •
                    cvs_login!uid!role!pwd → CLIENT(uid) ⊓
                    cvs_ci!uid             → CLIENT(uid) ⊓
                    cvs_co!uid             → CLIENT(uid) ⊓
                    cd!uid                 → CLIENT(uid) ⊓
                    read!uid               → CLIENT(uid) ⊓
                    write!uid              → CLIENT(uid) ⊓
                    mv!uid                 → CLIENT(uid) ⊓
                    mkdir!uid              → CLIENT(uid) ⊓
                    rmdir!uid              → CLIENT(uid) ⊓
                    rm!uid                 → CLIENT(uid) ⊓
                    setumask!uid           → CLIENT(uid) ⊓
                    chgrp!uid              → CLIENT(uid) ⊓
                    chmod!uid              → CLIENT(uid)

CLIENTS = ||| u:UID • login!uid → CLIENT(uid)
```

In the model above, we allow the user to apply for arbitrary roles — later, more refined

Figure 2.3 Communication in the open security architecture.



models of the CVS-server will refuse traces where the clients do not apply appropriate authentication, but here we concentrate just on the possible communications. However, we require that any client process starts with an overall login command that authenticates the client wrt. to the UNIX/Filessystem.

We turn now to the next component, the CVS–server. It is build as a simple component that accepts all logins, checkins and checkouts, does have (internal) communication with the file system, and is put into communication with the `CLIENTS`-component:

```
CVS       = cvs_login?uid     → CVS □
            cvs_ci?uid        → CVS □
            uid               → CVS □
            read!cvsadmin     → CVS □
            write!cvsadmin    → CVS □
            mv!cvsadmin       → CVS □
            mkdir!cvsadmin    → CVS □
            rmdir!cvsadmin    → CVS □
            rm!cvsadmin       → CVS

CLIENTS_CVS = CLIENTS [| {|cvs_login,cvs_ci,cvs_co|} |] CVS
```

We conclude by putting the whole architecture together with the UNIX-file- system, which is essentially a server for the `CLIENTS_CVS` subsystem.

```
FS    =    login?uid     → FS □
           cd?uid        → FS □
           read?uid      → FS □
           write?uid     → FS □
```

```
            mv?uid         → FS □
            mkdir?uid      → FS □
            rmdir?uid      → FS □
            rm?uid         → FS □
            setumask?uid   → FS □
            chgrp?uid      → FS □
            chmod?uid      → FS

SYSTEM = CLIENTS_CVS
            [| {|cd,read,write,mv,mkdir,rmdir,rmdir,rm,
               setumask,chgrp,chmod|}|]
         FS
```

This represents a quite precise view of the possible traces of the system. However, in principle, the following aspects are not covered by the architectural view discussed so far:

- The filesystem is a passive listener; its internal state is not modeled. In particular it does not contain files and their attributes,
- therefore, there are no control of reads and writes: the whole UNIX security service is missing.
- The architecture does not include ACL-lists, we will deliberately not treat this feature of our actual implementation in this paper for simplicity reasons.
- So far, we can not even state security properties ...
- nor the security state of repository (ACL's, passwd, file attributes)
- the roles and the role changes of the user

Since an analysis of the CVS-server security architecture is the main goal in this paper, we will have to model the compound state of `SYSTEM` in more detail. Since the compound state is inherently complex and requires mostly data structures, their composition and an overall transition relation over this, and since the communication aspects of our problem are of minor importance and can be captured on the level of the trace model of CSP (see [31]), we will apply in the next sections another specification formalism that is more suited for this problem domain: namely Z. In particular, tool support for a wealth of infinite data types (lists, tables, sets, ...) and infinite state models is available, while analysis tools for CSP such as FDR are restricted to finitely (and small) states and finite data structures.

The formal specification language Z [36] is based on typed set theory and first-order logic with equality, for which many excellent textbooks are available (cf. [40]). The syntax and the semantics are specified in an ISO-standard [19]; for future standardization efforts of operating system libraries or programming language semantics, Z is therefore a likely candidate. Z provides constructs for structuring and combining data-oriented specifications: schemas model the states of the system (*state schemas*) and operations on states (*operation schemas*), while the *schema calculus* is used to compose these sub-specifications to larger ones.

We introduce into Z and present these constructs using a standard example, Spivey's "birthday book". This simple system stores names and dates of birthdays and provides, for example, an operation to add a new birthday. In Z, abstract types for $NAME$ and $DATE$ can be declared that we use in a schema (consisting of a declaration part and a predicate part) to define the system state *BirthdayBook*. For transitions over the system state, the schema *AddBirthday* is used:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook}}\\ \hline
known : \mathbb{P}\, NAME\\
birthday : NAME \nrightarrow DATE\\ \hline
known = \mathsf{dom}\, birthday\\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}}\\ \hline
\Delta BirthdayBook\\
n? : NAME ; d? : DATE\\ \hline
n? \notin known\\
birthday' = birthday \cup \{n? \mapsto d?\}\\ \hline
\end{array}$$

$\Delta BirthdayBook$ imports the state schema into the operation schema in a "stroked" and a "non-stroked" version: $BirthdayBook'$ and *BirthdayBook*. The resulting variables $birthday'$ and *birthday* are conventionally understood as the states after and before the operation, respectively.

This system is refined to a more concrete one based on a state *BirthdayBook*1 containing two (unbounded) arrays and an operation that implements *AddBirthday* on this state:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook}1}\\ \hline
names : \mathbb{N} \nrightarrow NAME\\
dates : \mathbb{N} \nrightarrow DATE\\
hwm : \mathbb{N}\\ \hline
\forall i, j : 1 \mathinner{\ldotp\ldotp} hwm \bullet i \neq j\\
\qquad \Rightarrow names(i) \neq names(j)\\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}1}\\ \hline
\Delta BirthdayBook1\\
name? : NAME ; date? : DATE\\ \hline
\forall i : 1 \mathinner{\ldotp\ldotp} hwm \bullet name? \neq names(i)\\
hwm' = hwm + 1\\
names' = names \oplus \{hwm' \mapsto name?\}\\
dates' = dates \oplus \{hwm' \mapsto date?\}\\ \hline
\end{array}$$

One can use the schema calculus to combine different operation schemas into one operation. For example, one could strengthen the *AddBirthday* operation with an operation schema *AlreadyKnown* which expresses the fact that the entry that should be added already exists in the birthday book:

$$Add \equiv AddBirthday \lor AlreadyKnown$$

This concludes the birthdaybook-example.

The question arises how architectural — and this means behavioral — descriptions can be formalized in Z, which is traditionally considered as a data-oriented specification formalism. One part of the answer is that the *traditional use* of Z is geared toward data-specifications, the formalism itself is actually powerful enough to capture both facets of a system: Operation schemas can be converted into *relation on states*, over which traditional transitive closure or trace set constructions as in the semantics of CSP can be done. Over these, usual temporal reasoning (is there a state reachable in which a predicate P holds? If one reaches a state in which Q holds, is it possible to reach from there a state in which R holds? etc.) Another part of the answer is concerned with the communication primitives (e.g. `c!a -> P` or `c?x -> P x` in CSP) and their representation in Z in general and with the representation of connectors and their representation in our models in particular. In general, CSP offers the concept of synchronous communication between two processes incorporated in the synchronization operator `P [| C |] Q`: if two processes `P` and `Q` may engage in an event a and perform a system transition, that the combined process `P [| a |] Q` may engage in a and result in the combined successor states of the transition. Such a synchronous communication is a special form of a connector. If we represent in Z the transition relation by an operation schema P and Q, we can achieve the effect of a synchronization as follows, assuming that P sends a to Q:

$$\begin{array}{|l}\multicolumn{1}{l}{P} \\ \hline \Delta State \; ; a : T \\ \hline P_b(x) \\ a = x \\ \hline \end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{Q} \\ \hline \Delta State \; ; a : T \\ \hline Q_b(a) \\ \hline \end{array}$$

With the schema calculus, we can now express the synchronous communication of P to Q via a by a combination of schema conjunction $\wedge$ and the hiding operator corresponding to existential quantification (which is also the usual "wiring"-operator between components in hardware design):

$$PQ \equiv (P \wedge Q) \setminus \{a\}$$

With respect to the logical rules of Z, it is now an easy exercise to prove that this definition is equivalent to:

$$\begin{array}{|l}\multicolumn{1}{l}{PQ} \\ \hline \Delta State \\ \hline P_b(x) \wedge Q_b(x) \\ \hline \end{array}$$

This logical equivalence motivates how we will use Z with respect to architectural connectors in particular throughout our specification: Since we are interested in a "big-step semantics" of our system, i.e. *cvs_add* will be considered as one big transition step over the filesystem and intermediate steps (such as internal communication steps between the client and the server were abstracted away in our model), we will not distinguish the client and server sides in own schemas. Instead of introducing an *cvs_add_client* and *cvs_add_server* and putting them in parallel via synchronization over internal tables to *cvs_add*, we will give the specification of the combined system transistions directly. Alternatively, one could have introduced an own syntax for architectural compositions; however, since our focus is on theorem proving and not language design, we consider this approach as out of the scope of our work.

3 Discussion: Refining Security Architectures

In this chapter, we will discuss the conceptual problems of an architectural refinement and a description of the technology we use to preform our analysis formally.

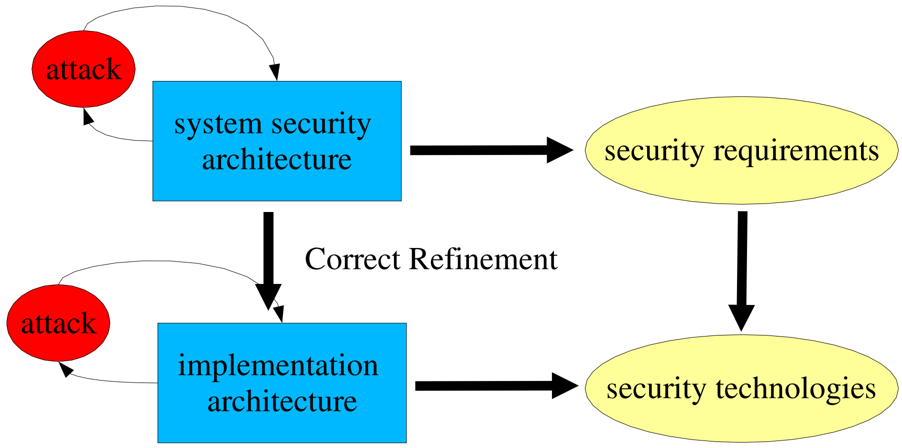
3.1 Concepts

As outlined in the previous section, there are conceptually simpler and more complex architecture models. In a way, one might ask if these models are related and if this relationship can be used for the task of an analysis.

It is useful to distinguish a *system architecture* (which will turn out to be similar to figure 2.1) from a *implementation architecture* (similar to figure 2.2). While the former is a model produced during the system requirements analysis of a software development process, the latter is merely a product of the design phase [6, 24], where the mapping to a concrete target operating system, programming language and programming libraries has to be worked out. In the context of security, this means that conceptually a mapping between these architecture levels is needed (see figure 3.1). We have a system architecture on top, that is designed to fulfill certain (security) requirements (a file can only be written, if a client has an appropriate *role*, for example). This architecture has to be mapped to an implementation architecture with its security technologies or mechanisms (such as read/write/execute permissions for files, for example) and security properties (a file can't be written without write permissions, for example). Now, we require the security technology mapping to be correct, i.e. the technologies and their properties meet indeed the security requirements on the system architecture level. As an *attack* we define a sequence of operations, that attempt to violate a security property; if an architecture "indeed meets its security properties", this means that all attacks have to be unsuccessful.

In the community of formal methods, the relation 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 these. Various refinement notions have been proposed (As for Z, see [40] for example, as for CSP, see [31]). In our setting, we chose to use only a very simple data refinement notion following [36] which essentially requires that any formula of the abstract view is implied by the formulae of the concrete view. Of course, following the approach taken

Figure 3.1 Refining Security Architectures.



in this paper, we do not attempt to formally prove such a relationship. Rather, we will check the consistency of the specifications and their refinement relation by paper and pencil proofs. In our setting, if the refinement relation holds, i.e. all operations on the concrete layer indeed preserve the properties of the abstract layer, it can be concluded for a *correct refinement*, that attacks built by operations of the abstract layer will not be successful, neither on the abstract nor the concrete level.

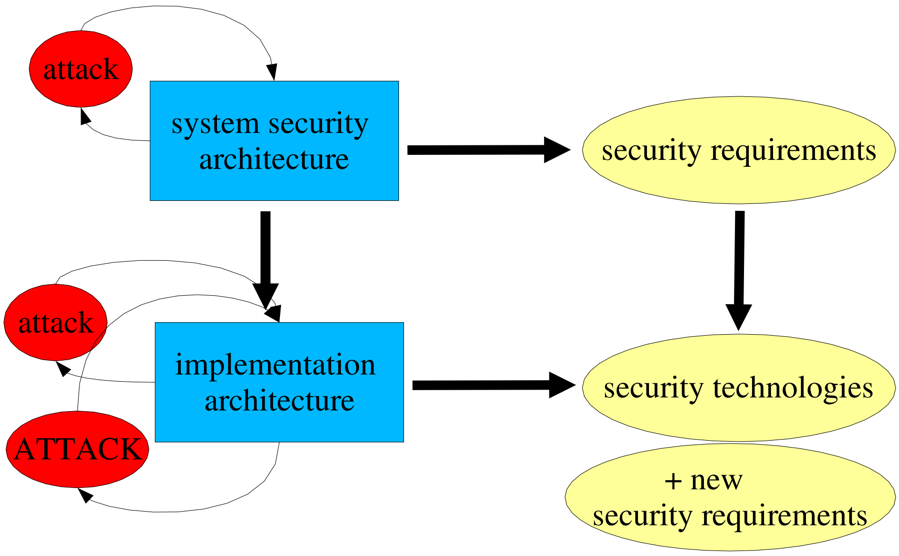
Unfortunately, as shown in figure 3.2, *implementing* one security architecture by another opens the door to *new* types of attacks on the implementation architecture, that can be completely overlooked on the abstract level.

These new types of attacks may be based on internal data structures or internal operations of the implementation layer. The problem is highly relevant for our case, and, as we believe, for many practically relevant applications and their security analysis too. In a way, this reflects common knowledge/experience that implementing an architecture seems to inherently "mess up" a security concept. Maybe that this experience is the deeper reason for the widespread skepticism against formal methods among implementors. However, from the methodological viewpoint this simply means that *attacks against the implementation* must simply be taken more seriously, which implies that models of implementation architectures are deserve more attention as before, where more abstract models have been preferred. But in security, more abstract models are not necessarily better ones.

Coming back to the general refinement scenario, refining an system architecture be a security technology mapping produces new possibilities of attacks, and consequently new security requirements on the implementation level.

As an example, consider the instantiation of the previous scheme with our application

Figure 3.2 An overview of the open security architecture.



scenario in 3.3.

Using our two-level modeling approach makes it possible to do security analysis on both abstraction levels. This enables also a combination-strategy of abstract and more concrete proofs in one setting, allowing to do as much as possible on the abstract level and concentrating the task on the concrete level to the parts that deal with implementation specialties that may open the door for attacks against the implementation.

3.2 Performing Refinement Proofs in HOL/Z 2.0

HOL-Z 2.0 is a tool chain for writing Z specifications, type-checking them, and proving properties about them. In this setting, we can specify our Z specifications in a type setting system, automatically generate proof obligations, import both of them into a theorem prover environment and use the existing proof mechanisms to gain a higher degree of automation. With the proof support for the schema calculus, realistic analysis of specifications, in particular refinement proof, in particular proofs along classical data refinement [36] become feasible.

3.2.1 HOL-Z: The Tool Chain for Literate Specification

HOL-Z is now embedded in a chain of tools, that can either be integrated into XEmacs (the way in which this document was created) or in usual shell scripts, that allow for an easy integration of the specification process into the general software development

Figure 3.3 An overview of the open security architecture.

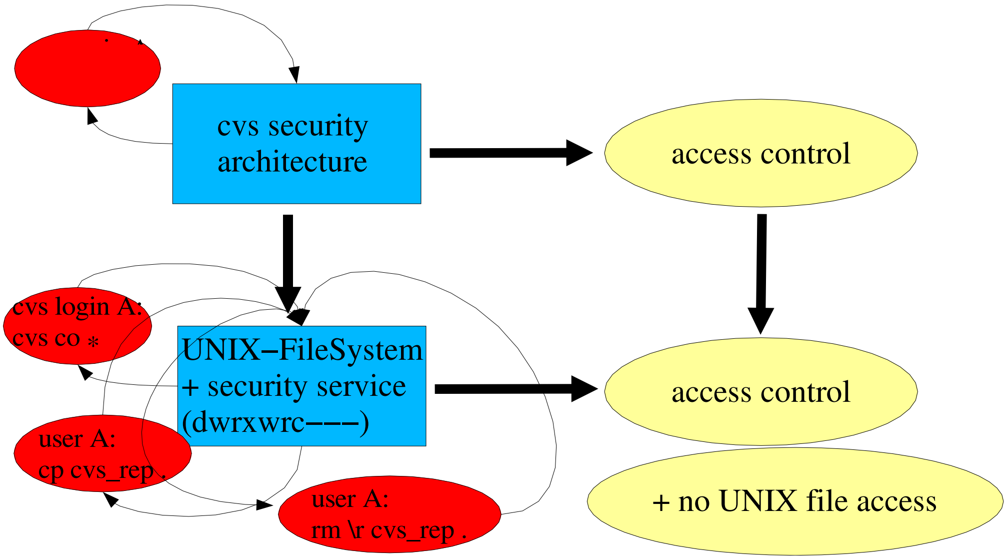
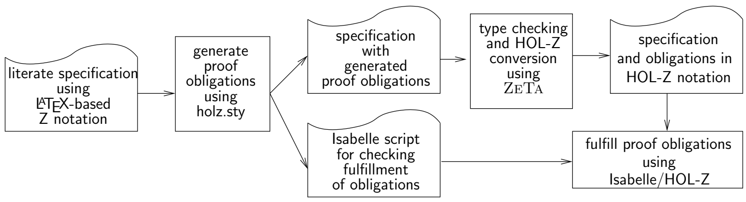


Figure 3.4 A Tool Chain supporting Literate Specification



process. The data flow in our tool chain can be described as follows: At the beginning, a normal LaTeX-based Z specification is created. Running LaTeX leads to the expansion of proof-obligation macros, which also generates an Isabelle-script that checks that the obligations are fulfilled (to be run at a later stage). ZETA takes over, extracts the definitions and axioms from the LaTeX source (including the generated ones) and type checks them or provides animation for some Z schemas. Our plug-in into ZETA converts the specification (sections, declarations, definitions, schemas, ...) into SML-files that can be loaded into Isabelle. In the theory contexts provided by these files, usual Isabelle proof-scripts can be developed. An integration of the process into a version management system allows for semantically checked specifications: for example, only when the proof obligation check scripts run successfully, new versions of the specification document are accepted as final versions.

`holz.sty` — A Macro Package for Generating Proof Obligations

We decided to use LaTeX itself as a flexible mechanism to construct and present proof-obligations inside the specification — this may include consistency conditions, refinement conditions or special safety properties imposed by a special formal method for a certain specification architecture. Our LaTeX-package `holz.sty` provides, among others, commands for generating refinement conditions as described in [36], where also the paradigmatical "BirthdayBook" is presented we use as running example. For *AddBirthday is refined by AddBirthday*1, we instantiate a macro as follows:

```
\zrefinesOp[Astate=BirthdayBook, Cstate=BirthdayBook1,
            Aop=AddBirthday, Cop=AddBirthday1,
            Args={name?: NAME; date?: DATE}, Abs=Abs]{Add}
```

Here, `Astate` contains the schema describing the abstract state and `Cstate` hold the schema describing the concrete state. Based on this input, our LaTeX-package automatically generates the following two proof obligations:

$$\begin{aligned}
Add_1 == {} & \forall BirthdayBook \,;\, BirthdayBook1 \,;\, name? : NAME \,;\, date? : DATE \bullet \\
& (\mathsf{pre}\, AddBirthday \land Abs) \Rightarrow \mathsf{pre}\, AddBirthday1 \\
Add_2 == {} & \forall BirthdayBook \,;\, BirthdayBook1 \,;\, BirthdayBook1' \,;\, name? : NAME \,; \\
& date? : DATE \bullet (\mathsf{pre}\, AddBirthday \land Abs \land AddBirthday1) \\
& \Rightarrow (\exists BirthdayBook' \bullet Abs' \land AddBirthday)
\end{aligned}$$

These proof obligations are type checked using ZeTa and are converted to HOL-Z by our ZeTa-to-HOL-Z converter.

The ZeTa-System

ZeTa [41] is an open environment for the development, analysis and animation of specifications based on Z. Specification documents are represented by *units* in the ZeTa system, that can be annotated with different *content* like LaTeX mark-up, type-checked abstract syntax, etc. The contents of units is computed by adaptors, which can be plugged into the system dynamically.

ZeTa-to-HOL-ZConverter

The converter consists of two parts: an adaptor that is plugged into ZeTaand converts the type-checked abstract syntax of a unit more or less directly into an SML file. On the SML side, this file is read and a theory context is build inside Isabelle/HOL-Z. This involves an own type-checking and an own check of integrity conditions of the specification and some optimizations for partial function application in order to simplify later theorem proving.

3.2.2 Proof Support for Z

Isabelle/HOL-Z revisited

The language Z is centered around a specific structuring mechanism called *schema*. Semantically, schemas are just sets of records (called *bindings* in Z terminology; [19]) of a certain type. Z is based on typed set theory equivalent to HOL set theory. However, a reference to a schema can play different *roles* in a specification: it can serve as *import* in the declaration list in other schemas, or as *predicate* (where all arguments are suppressed syntactically), or as *set* (see [23] for more details).

The approach of HOL-Z is to represent records by products in Isabelle/HOL and to manage their layout in order to support *as import*-references. This is achieved by a parser making implicit bindings in Z expressions explicit and generating coercions of schemas according to their role. For example, we assume throughout this section a schema A of type $[x_1 \mapsto \tau_1, x_2 \mapsto \tau_2]$ and a schema B of type $[x_2 \mapsto \tau_2, x_3 \mapsto \tau_3]$. Then, a schema expression $A \wedge B$ can be represented by

$$\lambda(x_1, x_2, x_3) \bullet A(x_1, x_2) \wedge B(x_2, x_3) \ ,$$

while an expression $A \cup A$ will be represented by $(\mathrm{asSet}\, A) \cup (\mathrm{asSet}\, A)$.

Thus, having "parsed away" the specific binding conventions of Z into standard λ-calculus, Isabelle's proof-engine can handle Z as ordinary HOL-formulas. There is no more "embedding specific" overhead such as predicates stating the well-typedness of certain expressions, etc.

For full-automatic proofs this is fine; however, in practice, realistic case studies require proofs with user interaction. This leads to the requirement that intermediate lemmas can be inserted "in the way of Z", intermediate results are presented "Z-alike" and deduction attempts to mimic the proof style imposed by Z (cf. [40]). As a prerequisite, we defined a special abstraction operator *SB* semantically equivalent to the pair-splitting λ-abstraction from the example above, which is actually encoded by:

$$\mathrm{SB}\ \text{“}x_1\text{”} \leadsto x_1, \text{“}x_2\text{”} \leadsto x_2, \text{“}x_3\text{”} \leadsto x_3 \bullet A(x_1, x_2) \wedge B(x_2, x_3)$$

where each field-name is kept as a (semantically irrelevant) string in the representation. Thus, while the "real binding" is dealt with by Isabelle's internal λ, which is underlying α-conversion, the *presentation* of intermediate results is done on the basis of the original field-names used in the users specification.

New Proof Support in HOL-Z 2.0

Schemas can also be used in quantifications as part of some very Z specific concept, the so-called *schema calculus*, for which we implemented syntax and proof support. For example, $\forall A \bullet B$ is a schema of type $\mathbb{P}([x_3 \mapsto \tau_3])$. In HOL-Z, it is represented by:

$$\mathrm{SB}\ \text{”}x_3\text{”} \leadsto x_3 \bullet \forall(x_1, x_2) : \mathrm{asSet}\, A \bullet B(x_2, x_3)$$

This and similar quantifiers and operators allow for a very compact presentation of typical proof-obligations occurring in refinements in Z. As example, we use an already slightly simplified version of Add_1 already described in [36, pp. 138]. (The full proof had been omitted for space reasons). Instead of referring to constants representing the proof obligations generated by the LaTeX-based front-end, we use the HOL-Z-parser directly:

zgoal thy
"∀ BirthdayBook • ∀ BirthdayBook1 • ∀ name? ∈ Name •∀ date? ∈ Date •
(name? ∉ known ∧ known = {n. ?i∈ #1..hwm. n=names i}
⟹ (∀i ∈ #1..hwm. name? ~= names i))";

which opens an Isabelle proof-state.

In the literature, several calculi for the schema calculus have been presented more or less formally ([19, 16]). From the perspective of HOL-Z it is quite clear what is needed: for any construct of the schema calculus, a special tactic must be provided that works analogously to the usual introduction and elimination rules for standard (bounded) quantifiers and set comprehensions. These tactics have been implemented and combined to new tactics, for example to a tactic that "strips-off" all universal quantifiers (including schema quantifiers) and implications. Thus, the HOL-Z tactic:

```
by(stripS_tac 1);
```

transforms the goal into the following proof state:

1. $\bigwedge$birthday known dates hwm names name? date? i.
   [| BirthdayBook (birthday, known); BirthdayBook1 (dates, hwm, names);
      name? $\in$Name; date? $\in$Date;
      name? $\notin$ known $\wedge$ known = {n. ?i$\in$ #1..hwm. n=names i};
      i $\in$ (#1 .. hwm) |] $\Longrightarrow$name? ~= names i

Note that the quite substantial reconstruction of the underlying binding still leads to a proof state that is similar in style and presentation to [40].

Besides the "schema calculus", Z comes with a large library of set operators specifying relations, functions as relations, sequences and bags; this library — called the *Mathematical Toolkit* of Z — differs in style substantially from the Isabelle/HOL library, albeit based on the same foundations. For HOL-Z 2.0, we improved this library substantially and added many derived rules that make a higher degree of automatic reasoning by Isabelle's standard proof procedures possible. For example, the goal above is simply "blown away" by:

```
auto();
```

which finishes the proof.

4 A Formal Model of the System Security Architecture

In this section, we will describe the more abstract view of our system. In order to avoid unnecessary complexity, we will reduce our architecture to exactly one client communicating with the server. This is realistic since the CVS server processes all requests sequentially and different clients could be modeled by one client logging in and out with different user IDs and roles. We will model the state of a client and the state of the CVS repository and the operations the combined system may perform. These operations include read and write access of clients to the repository and allow for a fairly compact description of the access control policy the CVS-Server may enforce.

This access control policy is well known as *role-based access control* access control@seeRBAC. In [33], a formal framework of role-based access control is described that introduces the RBAC-model for role hierarchies that is closely related to the security policy enforced by our system architecture. However, we differ from RBACin a number of minor issues in order to get a model that can be refined to the implementation architecture. For instance, we will *not* model *sessions* of RBACdirectly. Moreover, since we assume a one-to-one correspondence between roles and permissions (i.e. the *permission assignment* (PA) is a bijective function), we directly define the role hierarchy (RH) on permissions, which is more convenient for technical reasons. Further, since the administrator is part of the model (the authorization for a user may thus be dynamically withdrawn), the pair consisting of a *user ID* and its authentication (a *password*) representing a *permission* has to be kept in the user state and can not be evaluated to a permission for the whole session during login-time; rather, for any access to the repository, the system checks (as the real CVS-Server) if the pair still represents a valid permission at access-time.

Besides the access operations, we introduce a *login* operation that marks the entry of a session (if we neglect the dynamic aspect of authentication). During the login operation, a user authenticates for a role; this operation either results in a change of the permissions of the client state corresponding to that user or in a failure.

4.1 Fundamental Entities of the Model

section *Basics*

Common to both architectural models is an adopted RBACmodel. Therefore, we introduce basic data types for users, permissions (one particularity of our model is the isomorphism of roles and permissions), and passwords for authenticating.

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

The role hierarchy is captured by a reflexive, transitive relation with the administrative permissions (cvs_admin) as greatest and the permissions for public access (cvs_public) as least element. Since roles and permissions are isomorphic in our model, for convenience, we define the order directly on permissions.

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

For authentication and authorization of the client upon him accessing the repository or for logging in, we need a fixed preconceived table that associates permissions to an ID, a client has authenticated for with his password. Additionally, we define a table that associates to CVS user IDs the corresponding passwords. The client state of our model will manage such a table.

$$
\begin{array}{l}
AUTH\_TAB == Cvs\_Uid \times Cvs\_Passwd \nrightarrow Cvs\_Perm \\
PASSWD\_TAB == Cvs\_Uid \leftrightarrow Cvs\_Passwd
\end{array}
$$

4.2 The State of the System Architecture

section *AbstractState* **parents** *Basics*

As a basis to define the system state and its operations in the Z-sections `AbsState` and `AbsOperations` we have to define Z data types that represent files (a mapping from names to data), and data types for modeling access control properties, e.g., necessary permissions to access data.

$$
\begin{array}{l}
[Abs\_Name, Abs\_Data] \\
ABS\_DATATAB == Abs\_Name \nrightarrow Abs\_Data \\
ABS\_UIDTAB == Abs\_Name \nrightarrow Cvs\_Uid \\
ABS\_PERMTAB == Abs\_Name \nrightarrow Cvs\_Perm
\end{array}
$$

CVS server store their authentication table inside the repository — thus, the table can be accessed and modified using the operations of CVS. We will capture this fact in our model; as a prerequisite, we define a name $abs\_cvsauth$ to which data is associated, that is isomorphic to an authentication table. This isomorphism is established by a pair of functions — that can be viewed as *parser* and *pretty_printer* — and the usual axioms.

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

Note that the parser abs_auth_of is a partial function; this models the fact that not all file contents do actually represent input that can be interpreted. Moreover, we did not make any assumptions on authentication tables $AUTH\_TAB$; for example, a requirement like "there is someone that can authenticate as cvs_admin" (i.e. $cvs\_admin \in \mathsf{dom}\, at$ where $at : AUTH\_TAB$) would be helpful but is omitted since the real CVS-Server does not make this assumption. This has far reaching consequences; in particular, it is possible to bring the CVS server into a state where all operations block since all authentications fail, which resembles a *denial of service attack* at the POSIX-layer.

In the remainder of this section, we define the states of the client and the server component. To motivate the details of our specification, we present an excerpt from the informal description of the CVS login command:

> *Contacts a CVS server and confirms authentication information for a particular repository. This command does not affect either the working copy or the repository; it just confirms a password with a repository and stores the password for later use in the .cvspass file in your home directory. Future commands accessing the same repository with the same username will not require you to rerun login, because the client-side CVS will just consult the .cvspass file for the password.* [10]

The state of the client ($ClientState$) captures the concept of a working copy in the variable wc, a set of files, and also introduces wc_uidtab which is used to record for each file in the working copy the user ID that was used to access this file in the repository. The above mentioned file ".cvspass" is modeled by abs_passwd which is a set of pairs of user IDs and passwords the client has logged in with. In addition, we introduce a set of focused files ($wfiles$) which will be the implicit arguments for most CVS operations.

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

It is straightforward to motivate the state $RepositoryState$ of the server component: The central concept of CVS is the repository, a set of files, which is defined as rep. Since the access to each file is checked individually, we also define $rep\_permtab$, a table that associates the required access permissions to each file.

As mentioned above, CVS-Server stores the authentication data inside the repository — thus it can be accessed and modified with CVS operations. To model this, we introduce a name *abs_cvsauth* and auxiliary functions. In the server state, we now require that *abs_cvsauth* is a file in the repository, and that only the CVS administrator has sufficient permissions to access the authentication data, i.e. we set the necessary permission to the greatest element *cvs_admin* of our permission/role hierarchy.

RepositoryState

$rep : ABS\_DATATAB$
$rep\_permtab : ABS\_PERMTAB$

$abs\_cvsauth \in \mathsf{dom}\, rep$
$\mathsf{dom}\, rep = \mathsf{dom}\, rep\_permtab$
$rep\_permtab(abs\_cvsauth) = cvs\_admin$
$rep(abs\_cvsauth) \in \mathsf{dom}\, abs\_auth\_of$

4.3 The Operations of the System Architecture

section *AbsOperations* **parents** *AbstractState*

Now we define the operations of the system that model combined state transitions of the client and the repository. We begin with the login operation of our CVS-Server system architecture. Logging in marks the begin of a session, in which the client authenticates for permissions that control his access to the repository. The operation cd sets the focused files ($wfiles$) for the following operations.

abs_login

$\Delta ClientState$
$\Xi RepositoryState$
$passwd? : Cvs\_Passwd$
$uid? : Cvs\_Uid$

$(uid?, passwd?) \in \mathsf{dom}(authtab\, rep)$
$abs\_passwd' = abs\_passwd$
$\quad \oplus \{uid? \mapsto passwd?\}$
$wc' = wc$
$wc\_uidtab' = wc\_uidtab$
$wfiles' = wfiles$

abs_cd

$\Delta ClientState$
$\Xi RepositoryState$
$wfiles? : \mathbb{P}\, Abs\_Name$

$wfiles' = wfiles?$
$wc' = wc$
$wc\_uidtab' = wc\_uidtab$
$abs\_passwd' = abs\_passwd$

The next three operations allow for a synchronization of the working copy with the operations. Recall that our model deliberately neglects all aspects related to version control and merging of files; our model focuses on the security aspects of the CVS-Server.

The add operation mirrors the fact that the user has created new content (new filenames as well as new associated data). The operation adds this new data to the local working copy and makes sure that the new names are actually new in the repository.

$$
\begin{array}{l}
\textit{abs\_add} \\
\hline
\Delta \mathit{ClientState} \\
\Xi \mathit{RepositoryState} \\
\mathit{newfiles?} : \mathit{ABS\_DATATAB} \\
\hline
\mathsf{dom}\, \mathit{newfiles?} \cap \mathsf{dom}\, \mathit{rep} = \emptyset \\
\mathit{wc}' = \mathit{wc} \oplus \mathit{newfiles?} \\
\mathit{wc\_uidtab}' = \mathit{wc\_uidtab} \oplus \{n : \mathsf{dom}\, \mathit{newfiles?} \bullet n \mapsto (\mu \mathit{id} : \mathsf{dom}\, \mathit{abs\_passwd} | \mathit{true})\} \\
\mathit{wfiles}' = \mathit{wfiles} \\
\mathit{abs\_passwd}' = \mathit{abs\_passwd}
\end{array}
$$

The commit and checkin operations usually take a set if files as arguments (here denoted by $files?$). However, if no arguments are provided then these operations use the set of currently focused files ($wfiles$) as implicit arguments. This is modeled by restricting $files?$ by $files?$[1].

In our model the commit (or checkin) operation assumes that for all focused files (restricted by the input parameter), the user actually has access to these files in the repository, i.e. his actual permission is larger than the required one in the repository[2]. For these files, the content of the working copy is transferred to the repository. Note the use of wc' here to mirror the fact that the data in the working copy may have changed non-deterministically by user interaction and merge operations which are excluded from our model.

$$
\begin{array}{l}
\textit{abs\_ci} \\
\hline
\Delta \mathit{ClientState} \\
\Delta \mathit{RepositoryState} \\
\mathit{files?} : \mathbb{P}\, \mathit{Abs\_Name} \\
\hline
(\mathit{wfiles} \cap \mathit{files?}) \subseteq \mathsf{dom}\, \mathit{wc} \\
\mathit{rep}' = \mathit{rep} \oplus (\{n : \mathit{wfiles} \cap \mathit{files?} | n \notin \mathsf{dom}\, \mathit{rep} \land n \in \mathsf{dom}\, \mathit{wc\_uidtab} \\
\qquad \land (\mathit{wc\_uidtab}(n), \mathit{abs\_passwd}(\mathit{wc\_uidtab}\, n)) \in \mathsf{dom}(\mathit{authtab}(\mathit{rep}))\} \lhd \mathit{wc}') \\
\qquad\qquad \oplus(\{n : \mathit{wfiles} \cap \mathit{files?} | n \in \mathsf{dom}\, \mathit{rep} \land n \in \mathsf{dom}\, \mathit{wc\_uidtab} \\
\qquad\qquad \land (\mathit{wc\_uidtab}(n), \mathit{abs\_passwd}(\mathit{wc\_uidtab}\, n)) \in \mathsf{dom}(\mathit{authtab}\, \mathit{rep}) \\
\qquad\qquad \land (\mathit{rep\_permtab}(n), \mathit{authtab}(\mathit{rep})(\mathit{wc\_uidtab}(n), \\
\qquad\qquad \mathit{abs\_passwd}(\mathit{wc\_uidtab}\, n))) \in \mathit{cvs\_perm\_order}\} \lhd \mathit{wc}') \\
\mathit{rep\_permtab}' = \mathit{rep\_permtab} \oplus \{n : \mathit{wfiles} \cap \mathit{files?} | n \notin \mathsf{dom}\, \mathit{rep} \land n \in \mathsf{dom}\, \mathit{wc\_uidtab} \\
\qquad \land (\mathit{wc\_uidtab}(n), \mathit{abs\_passwd}(\mathit{wc\_uidtab}\, n)) \in \mathsf{dom}(\mathit{authtab}\, \mathit{rep}) \bullet \\
\qquad\qquad n \mapsto \mathit{authtab}(\mathit{rep})(\mathit{wc\_uidtab}(n), \mathit{abs\_passwd}(\mathit{wc\_uidtab}\, n))\} \\
\mathsf{dom}\, \mathit{wc}' = \mathsf{dom}\, \mathit{wc} \\
\mathit{wc\_uidtab}' = \mathit{wc\_uidtab} \land \mathit{abs\_passwd}' = \mathit{abs\_passwd} \land \mathit{wfiles}' = \mathit{wfiles}
\end{array}
$$

[1]Note that in order to operate on the complete set of focused files, $files?$ must be equal to $wfiles$.

[2]We slightly oversimplified the situation for the sake of the presentation: the real implementation actually filters out the non-accessible files, while in our model, the operation blocks.

The update operation updates every file in the working copy by the corresponding file in the repository if the client has sufficient permissions to access the file in the repository, i.e., is in a senior enough role (recall that merging is ignored and covered by non-determinism in the commit operation). Please note that this operation does not block if the client does not have sufficient permissions, but silently ignores those files. Additionally, this operation also updates the permission table in the working copy.

$$
\begin{array}{l}
\textbf{schema } abs\_up \\
\Delta ClientState \\
\Xi RepositoryState \\
files? : \mathbb{P}\, Abs\_Name \\
\hline
wc' = wc \oplus (\{n : wfiles \cap files? | n \in \mathsf{dom}\, rep \land n \in \mathsf{dom}\, wc\_uidtab \\
\qquad \land (wc\_uidtab(n), abs\_passwd(wc\_uidtab\, n)) \in \mathsf{dom}(authtab\, rep) \\
\qquad \land (rep\_permtab(n), authtab(rep)(wc\_uidtab(n), \\
\qquad\qquad abs\_passwd(wc\_uidtab\, n))) \in cvs\_perm\_order\} \lhd rep) \\
wc\_uidtab' = wc\_uidtab \cup \{n : wfiles \cap files? | n \in \mathsf{dom}\, rep \\
\qquad \land n \notin \mathsf{dom}\, wc\_uidtab \bullet n \mapsto (\mu id : \mathsf{dom}\, abs\_passwd | \\
\qquad\qquad authtab(rep)(id, abs\_passwd(id)) = rep\_permtab(n))\} \\
abs\_passwd' = abs\_passwd \land wfiles' = wfiles
\end{array}
$$

5 A Formal Model of the Implementation Security Architecture

5.1 The POSIX Security Architecture

We implement our role-based access control model by embedding it into the Discretionary Access Control (DAC) as provided by the Unix operating system and its hierarchic filesystem layer. This specific DAC implementation was first described in POSIX.1 and adopted by the "Single UNIX Specification Version 2" (Unix98) [38], a standard all modern Unix variants follow. This DAC is also described in the common successor of these two documents, which is "The Single UNIX Specification Version 3" [29]; referenced in the following as SUSV.

In this chapter, we define a formal model of this filesystem specification, and use it as a basis for specifying the implementations of the CVS commands.

5.1.1 Prelude

section *Prelude*

The following structure is a foundational part of our specification. Essentially, type sums are introduced, that are not part of the standard. Type sums can simulate enumerations in Z free type definitions on the fly. In the following, we define type sums via a generic schema in Z[1].

$Prelude\_SUM$ just serves as a tag of the type for the translation.

$[Prelude\_SUM]$

function $30(\_ + \_)$

$$[X, Y]$$
$$\_ + \_ : X \times Y \to \mathbb{P}(Prelude\_SUM \times X \times Y)$$

$$[X, Y]$$
$$Inl : X \to (X + Y)$$
$$Inr : Y \to (X + Y)$$

[1] While generic schemes are not accepted by HOL-Z 2.0, type sums are compiled via another mechanism and are therefore exceptionally accepted.

The definition of injection functions or recursors is omitted; the declarations above serve as pure type interface. A semantic definition of these constructs is straightforward, however.

5.1.2 The Basic Data Structures

section *FileSystem* **parents** *Prelude*, *Basics*

In the following, we declare basic abstract sorts for Unix user IDs, group IDs, data (file contents; left abstract in this model) and filenames. The file hierarchy is left implicit in our model; it will be essentially captured by some kind of "prefix-closedness" on legal paths in the filesystem. Additionally, we define an enumeration *Unit* which will be used to distinguish directories from regular files.

$$[Uid, Gid, Data, Name]$$
$$Path == \mathrm{seq}\, Name$$
$$Unit ::= Nil$$

In our model, we assume a static table *groups* that assigns a set of groups to each user. The axiomatic definition below also states the existence of a special user ID *root* which models the system administrator (usually called *root*). In principle, all security properties can only hold for all users except *root*, because root is allowed to do (almost) everything in a Unix system.

$$groups : Uid \rightarrow \mathbb{P}\, Gid$$
$$root : Uid$$

The function *cutPath* removes a given prefix from a path.

$$cutPath : (Path \times Path) \rightarrow\!\!\!\!\!+ \; Path$$
$$\forall a, b, c : Path \bullet cutPath(a, b) = c \iff a = b \frown c$$

Now we introduce the set of permissions, including the "set-uid bits": Within Unix every file belongs to a unique pair of owner (user) and group. Logically *file access* is divided into three classes:

1. Access by the *user* (owner) of the file,
2. access by a member of the *group* the file is owned by, or
3. access by any *other* user (world).

Further, the standard Unix DAC distinguishes between access for reading (`r`), writing (`w`), and executing (`x`).[2] The execution model of Unix further introduces the set-id (for owner and group) and the sticky bit. Our model uses only the set-gid (set-id for groups) facility on directories, which affects the default group of newly created files within that directory. We exclude the set-id bit, and the sticky bits, whose semantics has changed over time, and therefore can possibly

[2] For directories, this can be interpreted as access for searching in the directory (`r`), creating new files or subdirectories (`w`) and entering (`x`) the directory.

have different semantics within the different Unix-implementations (a detailed discussion of the different implementations is for example given in [12]). Apart from that, the sticky bits are irrelevant for our problem and are not used by our implementation either.

$$\begin{aligned} Perm ::= {} & ru|wu|xu \\ & |rg|wg|xg \\ & |ro|wo|xo \\ & |sg \end{aligned}$$

The filesystem consists of files which are represented by mapping the file content to each path, which is either $Data$ for regular files or $Unit$ for directories[3], and of file attributes (assigning to each file or directory the permissions[4], the user ID of the owner and the group it belongs to). In the following, if we speak only of *files* we mean a regular file or a directory. Note that our concept of file attributes may be extended easily by adding new components to its records.

$$\begin{aligned} & FILESYS\_TAB == Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit) \\ & FILEATTRIBUTES == [perm : \mathbb{P}\, Perm \; ; uid : Uid \; ; gid : Gid] \\ & FILEATTR\_TAB == Path \mathrel{+\!\!\!\!\rightarrow} FILEATTRIBUTES \end{aligned}$$

For testing if a directory contains a specific file we provide the function $\mathsf{is\_in}$. Further we provide functions that test for regular files ($\mathsf{is\_file\_in}$) and for directories ($\mathsf{is\_dir\_in}$).

$$\begin{array}{|l} \_ \mathsf{\ is\_in\ } \_ : Path \leftrightarrow (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \\ \_ \mathsf{\ is\_dir\_in\ } \_ : Path \leftrightarrow (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \\ \_ \mathsf{\ is\_file\_in\ } \_ : Path \leftrightarrow (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \\ \hline \forall fs : (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \; ; f : Path \bullet \\ \qquad (f \mathsf{\ is\_in\ } fs) \iff f \in \mathsf{dom}\, fs \\ \forall fs : (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \; ; d : Path \bullet \\ \qquad (d \mathsf{\ is\_dir\_in\ } fs) \iff (d \mathsf{\ is\_in\ } fs) \land (\exists u : Unit \bullet fs(d) = Inr(u)) \\ \forall fs : (Path \mathrel{+\!\!\!\!\rightarrow} (Data + Unit)) \; ; f : Path \bullet \\ \qquad (f \mathsf{\ is\_file\_in\ } fs) \iff (f \mathsf{\ is\_in\ } fs) \land \neg (f \mathsf{\ is\_dir\_in\ } fs) \end{array}$$

5.1.3 The Filesystem State

At this point we are ready to model the filesystem state, which mainly describes the set of existing files and their attributes. The filesystem has to obey the following two invariants:

1. 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, and

2. file attributes must be defined for every file in the filesystem.

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

[4] Often the terms *attributes* and *permissions* are used interchangeably.

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

First we 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 nitpicking detail of the SUSV access model is formalized, namely (local) file access is strictly tested in the following order and testing ends whenever a test for access fails:

- If the user owns the file, he can only access the file if the access attributes for users ($\{ru, wu, xu\}$) grant access.
- If the user is a member of the group owning the file, he can access the file if the access attributes for the group ($\{rg, wg, xg\}$) grant access.
- In all other cases the access attributes for others ($\{ro, wo, xo\}$) are checked.

This testing strategy leads to the strict precedence of owner permissions over the group or other permissions and to the following detail, which seems to be a little awkward at first sight: Assume the user u who is a member of the group g tries to read a file, owned by him, with permissions $\langle\!| perm == \{rg, ro\}, uid == u, gid == g |\!\rangle$. Is the user u allowed to read the file? Astonishingly, this is not the case because the rights specified for the user u (who is also the owner of the file) precede the rights given for the group or others. Therefore this file can only be read by all users not member of group g and all members of group g except the user u. Using the same "trick" it is possible to specify permissions for a file that is readable for all users except the members of one specific group.

Based on *has_attrib* we introduce *has_r_attrib*, *has_w_attrib* and *has_x_attrib* for testing the read, write and execute attributes.

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

$$
\begin{array}{|l}
has\_r\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_w\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_x\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
\hline
\forall uid : Uid \, ; p : Path \, ; fa : FILEATTR\_TAB \bullet \\
\quad has\_r\_attrib(uid, p, fa) \iff has\_attrib(uid, p, fa, ru, rg, ro) \\
\forall uid : Uid \, ; p : Path \, ; fa : FILEATTR\_TAB \bullet \\
\quad has\_w\_attrib(uid, p, fa) \iff has\_attrib(uid, p, fa, wu, wg, wo) \\
\forall uid : Uid \, ; p : Path \, ; fa : FILEATTR\_TAB \bullet \\
\quad has\_x\_attrib(uid, p, fa) \iff has\_attrib(uid, p, fa, xu, xg, xo)
\end{array}
$$

For accessing a specific file described by an absolute path name, it is not sufficient to test the file attributes. Access to the directory containing that file also has to be granted. In particular, we need to be able to enter each directory (the execution bit has to be set) along the path. Here it makes no difference if the "enter" access is granted based on user, group or other permissions. This is described in the functions has_w_access and has_r_access for testing if a user can write or read a file.

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

5.1.4 Process State

In addition to the filesystem state, we introduce a state schema for user related information, namely the current user and group ID, the user's umask (which is used to set the initial file attributes on newly created files), and the current working directory of the user. The current working directory *wdir* will always be used as an implicit parameter to filesystem and CVS operations.

Table 5.1 Implementation architecture vs. SUSV

Z Specification	Unix specification	purpose
cd	`chdir`	change current working directory
$mkdir$	`mkdir`	create new directory
$mkfile$	`create` (`open`)	create new file
$access$	`read`	read file content
$write$	`write`	write data to file
rm	`unlink`	remove file
$rmdir$	`rmdir`	remove directory
mv	`rename`	rename (move) file or directory
$chown$	`chown`	change group and owner of files
$chmod$	`chmod`	change access permissions of file or directory
$setumask$	`umask`	set the file mode creation mask

ProcessState

$uid : Uid$

$gid : Gid$

$umask : \mathbb{P}(Perm \setminus \{sg\})$

$wdir : Path$

5.1.5 Modeling POSIX/Unix Operations

Now, we introduce the specification of the filesystem operations incorporating the access control mechanisms. We need to specify methods for changing directories, accessing files and changing their access modes. See Tab. 5.1 for an assignment of the SUSV functions [29] to our model.

Although many of these operations have similar requirements, e.g. the current working directory must be a valid,we didn't define these requirements as general state invariants to keep our model general and reusable. Furthermore, according to the SUSV specification, every operation checks at invocation-time if the working directory is a valid directory and fails otherwise.

The operation `cd` changes the user's current working directory. In order to set this, the new path must point to a valid directory, and the user must have sufficient permissions to access it.

cd

$\Xi FileSystem$

$\Delta ProcessState$

$p? : Path$

$\forall p : Path | p \text{ prefix } p? \bullet has\_x\_attrib(uid, p, attributes) \land p \text{ is\_dir\_in } files$

$uid' = uid \land gid' = gid \land umask' = umask \land wdir' = p?$

We introduce filesystem operations for creating directories (`mkdir`) and regular files (`mkfile`). In comparison to the SUSV operations, these functions are slightly simplified, in particular we create the files in the current working directory. Due to the necessary calculation of the initial attributes assigned to the newly created file (depending on the user's umask and set-gid bit), the specification of $mkdir$ and $mkfile$ are already quite complex. As expected, we require that the user must have write permissions on the working directory, and only files with names that don't yet exist in the working directory may be created.

$$
\begin{array}{l}
\textit{mkdir} \\
\hline
\Delta FileSystem \\
\Xi ProcessState \\
u? : Name \\
\hline
wdir \ \mathsf{is\_dir\_in} \ files \\
\neg(wdir \frown \langle u? \rangle \ \mathsf{is\_dir\_in} \ files) \\
has\_w\_access(uid, wdir, attributes) \\
files' = files \cup \{wdir \frown \langle u? \rangle \mapsto Inr(Nil)\} \\
attributes' = (\mathbf{let}\, pms == \{ru, wu, xu, rg, wg, xg, ro, wo, xo\} \setminus umask \bullet \\
\qquad \mathbf{let}\, pms' == \mathbf{if}\ sg \in (attributes(wdir)).perm \\
\qquad\qquad \mathbf{then}\, pms \cup ((attributes(wdir)).perm \cap \{rg, wg, xg\}) \\
\qquad\qquad \mathbf{else}\, pms \bullet \\
\qquad \mathbf{let}\, wdir\_gid == \mathbf{if}\ sg \in (attributes(wdir)).perm \\
\qquad\qquad \mathbf{then} (attributes(wdir)).gid \\
\qquad\qquad \mathbf{else}\, gid \bullet \\
\qquad\quad attributes \oplus \{wdir \frown \langle u? \rangle \mapsto \\
\qquad\qquad \langle\!| perm == pms', uid == uid, gid == wdir\_gid |\!\rangle \}) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{mkfile} \\
\hline
\Delta FileSystem \\
\Xi ProcessState \\
u? : Name \\
d? : Data \\
\hline
wdir \ \mathsf{is\_dir\_in} \ files \\
\neg(wdir \frown \langle u? \rangle \ \mathsf{is\_dir\_in} \ files) \\
has\_w\_access(uid, wdir, attributes) \\
files' = files \oplus \{wdir \frown \langle u? \rangle \mapsto Inl(d?)\} \\
attributes' = (\mathbf{let}\, pms == \{ru, wu, xu, rg, wg, xg, ro, wo, xo\} \setminus umask \bullet \\
\qquad \mathbf{let}\, pms' == \mathbf{if}\ sg \in (attributes(wdir)).perm \\
\qquad\qquad \mathbf{then}\, pms \cup ((attributes(wdir)).perm \cap \{rg, wg, xg\}) \\
\qquad\qquad \mathbf{else}\, pms \bullet \\
\qquad \mathbf{let}\, wdir\_gid == \mathbf{if}\ sg \in (attributes(wdir)).perm \\
\qquad\qquad \mathbf{then} (attributes(wdir)).gid \\
\qquad\qquad \mathbf{else}\, gid \bullet \\
\qquad\quad attributes \oplus \{wdir \frown \langle u? \rangle \mapsto \\
\qquad\qquad \langle\!| perm == pms', uid == uid, gid == wdir\_gid |\!\rangle \}) \\
\hline
\end{array}
$$

Beside the fact that files of different types are created, the only difference lies in the treatment of the "set-gid" attribute (sg): If this attribute is set on the current working directory, it will also be set on newly created subdirectories whereas it is ignored on regular files.

For accessing and manipulating the file content, we provide the operations *access* and *write*, which are only defined for accessing and writing the whole file "at once". Variants of these operations for partial writing and reading (e.g. appending data) can be modeled at a higher abstraction level as a sequence of *access* and *write* operations. As before, we require the accessed file to be a valid file in the current working directory and the user to have access and write permissions on the file, respectively.

access

$$\begin{array}{l}
\Xi FileSystem \\
\Xi ProcessState \\
u? : Name \\
c! : (Data + Unit) \\
\hline
(wdir \frown \langle u? \rangle) \ \mathsf{is_file_in}\ files \\
has_r_access(uid, wdir, attributes) \\
c! = files(wdir \frown \langle u? \rangle)
\end{array}$$

write

$$\begin{array}{l}
\Delta FileSystem \\
\Xi ProcessState \\
u? : Name \\
d? : Data \\
\hline
(wdir \frown \langle u? \rangle) \ \mathsf{is_file_in}\ files \\
has_w_access(uid, wdir \frown \langle u? \rangle, attributes) \\
files' = (\{wdir \frown \langle u? \rangle\} \ndres files) \oplus \{(wdir \frown \langle u? \rangle) \mapsto Inl(d?)\} \\
attributes' = attributes
\end{array}$$

For deleting, we provide the operation `rm` for files, and `rmdir` for directories. Following the SUSV, `rmdir` can only delete empty directories.

rm

$$\begin{array}{l}
\Delta FileSystem \\
\Xi ProcessState \\
u? : Name \\
\hline
(wdir \frown \langle u? \rangle) \ \mathsf{is_file_in}\ files \\
has_w_access(uid, wdir, attributes) \\
has_w_access(uid, wdir \frown \langle u? \rangle, attributes) \\
files' = \{wdir \frown \langle u? \rangle\} \ndres files \\
attributes' = attributes
\end{array}$$

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

Again, we require the file or directory that should be deleted to be valid in the current working directory, and the user to have write permissions on the file or directory and the working directory.

Additionally, we provide an operation *mv* for moving (renaming) files. This specification mimics the often misunderstood "feature" of the SUSV `rename` operation which allows to move a file without requiring any permission on the file itself, but instead depends on the access attributes of the directory containing the file. To be more precise: For moving a file, only write access for the source and the destination directory (including the needed execute attribute along the paths) is needed. As a consequence, the behavior of `rename` can not be modeled as a sequence of `cp`, `create`, `mkdir`, `mkfile`, `rm`, and `rmdir` operations.

This confirms the intuition that directories can be interpreted as regular files whose content is the set of names (or references) of the files it contains. Using this intuition, moving a file is an operation which only accesses the content of the parent directory, hence no access or write privileges are needed on the file itself. We only model *mv* as a simplified version of `rename`, allowing only renaming within the same working directory. This is no restriction, as all other behavior can be "emulated" by calling sequences of *cd*, *access*, *write*, *rmdir* and *unlink* operations. Thus, the only reason for specifying *mv* is the possibility of renaming files within a directory without having any rights on them.

To handle the complexity of this operation, we employ the schema calculus and split the operation into different schemas: One that captures moving files, one that captures moving directories, and one that specifies renaming a directory. Files can either be renamed or an existing file can be overwritten with another file.

$$
\begin{array}{|l}
\hline
\textit{mv\_file} \\
\Delta FileSystem \\
\Xi ProcessState \\
u1?, u2? : Name \\
\hline
wdir \frown \langle u1? \rangle \ \mathsf{is\_file\_in}\ files \\
has\_w\_access(uid, wdir, attributes) \\
\neg(wdir \frown \langle u2? \rangle \ \mathsf{is\_in}\ files) \\
\quad \vee((wdir \frown \langle u2? \rangle \ \mathsf{is\_file\_in}\ files) \wedge has\_w\_access(uid, wdir \frown \langle u1? \rangle, attributes)) \\
files' = (\{wdir \frown \langle u1? \rangle\} \lhd files) \oplus \{(wdir \frown \langle u2? \rangle) \mapsto files(wdir \frown \langle u1? \rangle)\} \\
attributes' = (\{wdir \frown \langle u1? \rangle\} \lhd attributes) \oplus \\
\quad \{(wdir \frown \langle u2? \rangle) \mapsto attributes(wdir \frown \langle u1? \rangle)\} \\
\hline
\end{array}
$$

The schema mv_dir defines the case where an existing file or directory (and subsequently its contents, i.e. files and subdirectories) is moved into an existing directory.

$$
\begin{array}{l}
\textbf{schema } mv\_dir \\
\Delta FileSystem \\
\Xi ProcessState \\
u1?, u2? : Name \\
\hline
wdir \frown \langle u1? \rangle \ \mathsf{is\_in}\ files \\
has\_w\_access(uid, wdir, attributes) \\
wdir \frown \langle u2? \rangle \ \mathsf{is\_dir\_in}\ files \\
has\_w\_access(uid, wdir \frown \langle u1? \rangle, attributes) \\
files' = (\{p : \mathsf{dom}\, files | wdir \frown \langle u1? \rangle \ \mathsf{prefix}\ p\} \mathbin{⩤} files) \\
\qquad \oplus \{p : Path | wdir \frown \langle u1? \rangle \frown p \ \mathsf{is\_in}\ files \bullet \\
\qquad\qquad wdir \frown \langle u2?, u1? \rangle \mapsto files(wdir \frown \langle u1? \rangle \frown p)\} \\
attributes' = (\{p : \mathsf{dom}\, attributes | wdir \frown \langle u1? \rangle \ \mathsf{prefix}\ p\} \mathbin{⩤} attributes) \\
\qquad \oplus \{p : Path | wdir \frown \langle u1? \rangle \frown p \ \mathsf{is\_in}\ files \bullet \\
\qquad\qquad wdir \frown \langle u2?, u1? \rangle \mapsto attributes(wdir \frown \langle u1? \rangle \frown p)\}
\end{array}
$$

Sadly, renaming a directory cannot be captured by the above schema and we have to introduce the following additional schema $rename\_dir$.

$$
\begin{array}{l}
\textbf{schema } rename\_dir \\
\Delta FileSystem \\
\Xi ProcessState \\
u1?, u2? : Name \\
\hline
wdir \frown \langle u1? \rangle \ \mathsf{is\_dir\_in}\ files \\
has\_w\_access(uid, wdir, attributes) \\
\neg(wdir \frown \langle u2? \rangle \ \mathsf{is\_in}\ files) \\
has\_w\_access(uid, wdir \frown \langle u1? \rangle, attributes) \\
files' = (\{p : \mathsf{dom}\, files | wdir \frown \langle u1? \rangle \ \mathsf{prefix}\ p\} \mathbin{⩤} files) \\
\qquad \oplus \{p : Path | wdir \frown \langle u1? \rangle \frown p \ \mathsf{is\_in}\ files \bullet wdir \frown \langle u2? \rangle \mapsto files(wdir \frown \langle u1? \rangle \frown p)\} \\
attributes' = (\{p : \mathsf{dom}\, attributes | wdir \frown \langle u1? \rangle \ \mathsf{prefix}\ p\} \mathbin{⩤} attributes) \\
\qquad \oplus \{p : Path | wdir \frown \langle u1? \rangle \frown p \ \mathsf{is\_in}\ files \bullet \\
\qquad\qquad wdir \frown \langle u2? \rangle \mapsto attributes(wdir \frown \langle u1? \rangle \frown p)\}
\end{array}
$$

Using the schema calculus, we can now join these three operations to build a comprehensive mv operation:

$$mv == mv\_file \lor mv\_dir \lor rename\_dir$$

For the following two operations (*chown* and *chmod*), we require that the input file exists in the filesystem and that the user is either the administrator or the owner of the file.

The operation "change owner" (*chown*) allows to set the owner and group of files and directories. To set the group attribute of a file, the user must be a member of that new group. Then the attributes are set according to the input parameters; the permissions remain unchanged except for one minor issue: the set-id flags are omitted.

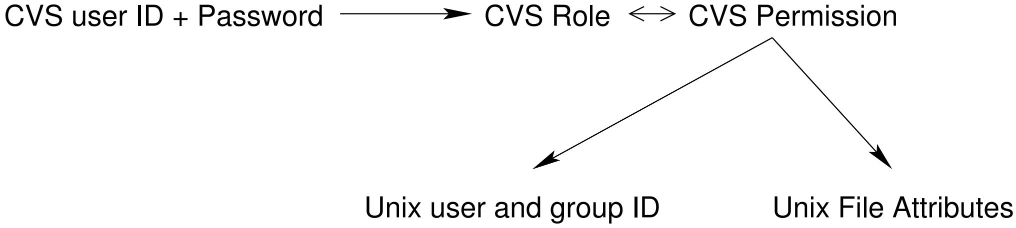
$$
\begin{array}{l}
\textbf{chown} \\
\hline
\Delta FileSystem \\
\Xi ProcessState \\
p? : Path \\
gr? : Gid \\
ow? : Uid \\
\hline
p? \in \mathsf{dom}(files) \\
(root = uid) \lor \\
\qquad (root \neq uid \land ow? = uid \land ow? = ((attributes\, p?).uid) \land gr? \in groups(uid)) \\
files' = files \\
attributes' = attributes \oplus \{p? \mapsto \langle\!| perm == (attributes\, p?).perm \setminus \{sg\}, \\
\qquad\qquad\qquad\qquad\qquad uid == ow?, \\
\qquad\qquad\qquad\qquad\qquad gid == gr? |\!\rangle\} \\
\hline
\end{array}
$$

The operation "change mode" (*chmod*) allows to set the file access attributes for a file.

$$
\begin{array}{l}
\textbf{chmod} \\
\hline
\Delta FileSystem \\
\Xi ProcessState \\
ps? : \mathbb{P}\, Perm \\
p? : Path \\
\hline
p? \in \mathsf{dom}\, files \\
(root = uid) \lor (root \neq uid \land uid = (attributes\, p?).uid) \\
files' = files \\
attributes' = attributes \oplus \{p? \mapsto \langle\!| perm == ps?, \\
\qquad\qquad\qquad\qquad\qquad uid == (attributes\, p?).uid, \\
\qquad\qquad\qquad\qquad\qquad gid == (attributes\, p?).gid |\!\rangle\} \\
\hline
\end{array}
$$

The operation *setumask* allows setting the value of the *umask*, which influences the file attributes of newly created files. It is fairly simple – apart from the fact that one cannot use it to set the set-uid and the set-gid bit, it straightforwardly updates the *umask* in the *ProcessState*.

Figure 5.1 Relation between different IDs



$$
\begin{array}{|l}
\hline
\textit{setumask} \\
\Xi FileSystem \\
\Delta ProcessState \\
mask? : \mathbb{P}(Perm \setminus \{sg\}) \\
\hline
uid' = uid \land gid' = gid \land umask' = mask? \land wdir' = wdir \\
\hline
\end{array}
$$

5.2 The CVS-Server Security Architecture

section $CVSServer$ **parents** $Prelude, FileSystem, Basics$

Our CVS-Server provides two security mechanisms for access control of files managed by the data base of the CVS server (called repository), an ACL-based (access control list) mechanism and a general hierarchical role concept.

However, in the following, we will not model the ACL-based mechanism that allows for an individual, "point-wise" access control of files in the CVS system. This mechanism is an add-on to the hierarchical concept described in this section and has quite complementary pragmatics. We base our model on a bijection between roles and permissions and use the terms "role" and "permission" interchangeably as described in Section 4. In our formal model we only define a type for permissions (Cvs_Perm) but in the textual descriptions we might use the term "roles" when appropriate.

5.2.1 Mapping CVS onto Unix

Technically, our CVS-Server role model will be mapped onto a Unix file system. In the following, we collect several requirements of this mapping. Essentially, we describe a mapping of the CVS roles (actually CVS permissions) on Unix user and group IDs and their permissions (file attributes).

For every CVS operation, the server determines the CVS permissions according to the user's CVS user ID and password. These permissions are then mapped onto Unix user and group IDs, and these are compared to the file attributes of the files and directories which are the arguments of the operations. It is important to notice that CVS IDs (Cvs_Uid) are independent of Unix IDs (Uid) and that the Unix IDs which are used by CVS are disjoint from "normal" Unix user IDs, i.e. it is impossible to login to the Unix system with such a special Unix ID. Figure 5.1 should help clarify this cumbersome concept. From these distinctness constraints it follows that the

Unix system administrator and the CVS administrator are different (have different user IDs). Of course, our security mechanism is bypassed if a user gets `root` permissions on the Unix system.

These distinctness constraints will show up as critical preconditions of the security architecture's safety properties. Moreover, we require that the group table (administrated by the system administrator and nobody else) is compatible with *cvs_perm_order*, as defined in Section 4.1.

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

The latter requirement means that for all CVS-related user IDs u (and hence the ones in $\mathsf{ran}\, cvsperm2uid$), the function *groups* is expected to map u to the set of group IDs that correspond to the CVS permissions that are equivalent to roles below the user's current role. As a consequence, $cvsperm2uid(cvs\_public)$ is in $groups(u)$ for all u, $cvsperm2gid(cvs\_admin)$ is contained in $groups(u)$ for no u different from *cvs_admin*, and we have

$$groups(cvsperm2uid(cvs\_admin)) = \mathsf{ran}\, cvsperm2gid,$$

i.e. the CVS administrator belongs to all CVS related groups. (For an example of an ordering and a setup of groups satisfying all these constraints, see Appendix A).

Now we turn to the model of the repository (the internal data base of the CVS server). A distinguished feature of our approach is that the repository is just a part of the ordinary filesystem. Hence, users might have access to that part of the filesystem representing the repository via standard Unix commands like `cd` or `mv`, bypassing the access control of CVS or — even worse — the integrity of the repository. By accessing the internal files that contain the representation of the mapping from CVS user IDs to CVS permissions or to the necessary authentication information, users could also corrupt the authentication mechanism of the CVS server.

5.2.2 The CVS Repository

The CVS repository is a subtree of the normal filesystem. The root of this tree is denoted by the absolute path *cvs_rep* (called `$CVS_ROOT` in the implementation). All paths inside the repository are relative to *cvs_rep*. 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 $CVSROOT$. Additionally, we define functions that convert between the conceptual authentication table (defined in section 4.1) and the actual data associated to files containing an authentication table, in particular *cvsauth*.

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

5.2.3 Modeling the CVS Filesystem State

A major design decision for our specification is to enrich the $FileSystem$ state specified in the previous section by new state components relevant to CVS, or more precisely, the combined client/server process of CVS. First, there is the concept of a working copy — a copy of (a local part of) the repository, for which the user needs write permissions (w.r.t. Unix permissions) and at least read permissions for the repository (w.r.t. the CVS permissions). A working copy can be represented by a table associating CVS attributes to paths referring to the file system; in our model, CVS attributes are characterized by the CVS user ID f_uid, representing the "effective role" or equivalently in our model the "effective permission" a file possesses, and the path rep where the file is located in the repository.[5]

$$
\begin{array}{l}
CVS\_ATTRIBUTES == [rep : Path\ ;\ f\_uid : Cvs\_Uid] \\
CVS\_ATTR\_TAB == Path \rightarrowtail CVS\_ATTRIBUTES
\end{array}
$$

The function $repOf$ is simply a projection function needed to overcome some shortcomings in the current HOL-Z implementation.

$$
\begin{array}{|l}
repOf : CVS\_ATTRIBUTES \rightarrow Path \\
\hline
\forall as : CVS\_ATTRIBUTES \bullet repOf(as) = as.rep
\end{array}
$$

This way, some non-trivial invariants can be stated particularly nicely; namely that

1. working copies are just subtrees in the file system, and that
2. the CVS administrative file containing the authentication information ($cvsauth$) exists in the filesystem.

Moreover, we introduce a state variable cvs_passwd into the $Cvs\_FileSystem$ state, denoting a table that associates the CVS user IDs and their passwords the client logged in with.

Summarizing, we have the following central invariant properties of the CVS extended filesystem ($Cvs\_FileSystem$):

- Working copies must be subsystems of the file system.

[5] In the CVS implementation, this working copy related information is stored in `CVS` directories and comprises lists of version numbers, dates, branches, names of controlled files etc. Thus "original path" rep is kept in the file "`Repository`", the CVS user ID is part of the information kept in "`Root`".

- There must be a `cvsauth` file in the file system, that is a file in the administrative area $cvs\_rep \frown CVSROOT$ (from prefix-closedness it follows, that they exist).
- $cvs\_rep \frown CVSROOT$ must be a directory.
- The root directory of the repository *cvs_rep* must disallow any access for *others*. In particular it must have the permissions $\{ru, wu, xu, xg, sg\}$; the user ID must be *cvs_admin*, and the group ID *cvs_public*. For the implementation, it seems to be preferable to postulate these rights for the parent directory of *cvs_rep*. In this setting the CVS administrator is free to set the access rights of *cvs_rep* to suit his needs. These settings insure that
  - only a CVS administrator is able to create new "top level" files and directories, and
  - only members of the CVS group hierarchy are able to change into the repository; in particular this prevents *any* attack on the Unix file system level by a user not belonging to the CVS group hierarchy.
- General requirements on attributes *within* the repository are:
  - The owners of files must be Unix user IDs that are disjoint from "regular" Unix user IDs, and the group IDs must be legal w.r.t. the CVS role hierarchy. This guarantees that any Unix user with a user ID and group ID that is not a special Unix ID in the restricted CVS range has only the rights described by the file attributes for others. Thus, our initial invariant for the *cvs_rep* directory implies that such a user cannot do anything, using only Unix operations, within the repository.
  - All read, write and execute permissions are identical for users and groups. Together with our group setup this ensures that the initial CVS role (represented by the Unix file owner) and all roles with higher precedence have the same access rights on that file.
- General requirements on attributes within the administrative area of the repository are:
  - The top of the administrative area must belong to the group *cvs_admin* and owned by the user *cvs_admin* with permissions $\{ru, wu, xu, rg, wg, xg, sg, xo\}$. This way the CVS administrator has write access to the administrative area.
  - The owner user ID of all files within the administrative area must correspond to the CVS permission (role) *cvs_admin*.
  - The owner of all files must be either *cvs_admin* or *cvs_public*. We have to allow this because some files (namely `history` and `modules`) have to be readable, or even writable (e.g. `history`), by any CVS user.
  - The file $cvs\_rep \frown CVSROOT \frown cvsauth$ has to be handled sensitively because it contains the most security related data: The CVS-Server user administration. Therefore this directory is owned by the user *cvs_admin* with the restrictive attributes $\{ru, wu, xu\}$. This configuration guarantees that *no one* except the CVS administrator is able to access files in this directory.

- All files must have read permissions for user and group but no write permission at all. This invariant holds for all files in the repository which are subject to version control by CVS.[6]

- General requirements on attributes *outside* the administrative area of the repository are:

  - On all directories the permissions must be set to $\{ru, wu, xu, rg, wg, xg, sg, xo\}$. Because our model allows subdirectories to have more rights than their parent directories have (e.g. directories that allow only access for *csv_admin* can have subdirectories that allow write access for *cvs_public*), we have to allow anybody to change into directories (thus the *ox* attribute). Further, the *sg* attribute ensures that newly created subdirectories inherit the access rights from their parent directories by default.

  - All files must have read permissions for user and group, but no write permission. This is a property of the real CVS implementation for preventing accidental deletion within the repository. This mechanism is in principle superseded by our general setup.

These requirements are captured in the following predicates. General requirements inside the repository are defined by the predicate *attributes_in_rep*, requirements on attributes inside the administrative area are defined by the predicate *attributes_in_root*, and requirements on attributes outside the administrative area are captured by the predicate *attributes_outside_root*.

$$
\begin{array}{l}
attributes\_in\_rep\_ : \mathbb{P}\, FileSystem \\
\hline
\forall fs : FileSystem \bullet attributes\_in\_rep(fs) \iff \\
\quad (\forall p : \mathsf{dom}\, fs.files | (cvs\_rep \;\mathsf{prefix}\; p) \bullet \\
\quad\quad (((fs.attributes\, p).uid) \in \mathsf{ran}\, cvsperm2uid \land \\
\quad\quad ((fs.attributes\, p).gid) \in groups((fs.attributes\, p).uid) \land \\
\quad\quad (ru \in ((fs.attributes\, p).perm) \iff rg \in (fs.attributes\, p).perm) \land \\
\quad\quad (wu \in ((fs.attributes\, p).perm) \iff wg \in (fs.attributes\, p).perm) \land \\
\quad\quad (xu \in ((fs.attributes\, p).perm) \iff xu \in (fs.attributes\, p).perm)))
\end{array}
$$

[6]Exceptions are `$CVS_ROOT/CVSROOT/val-tags` and `$CVS_ROOT/CVSROOT/history` which are subject to revision control, but stored within the repository.

$$
\begin{array}{l}
attributes\_in\_root\_ : \mathbb{P}\, FileSystem \\
attributes\_outside\_root\_ : \mathbb{P}\, FileSystem \\
\hline
\forall fs : FileSystem \bullet attributes\_in\_root(fs) \iff \\
(\quad \forall p : \mathsf{dom}\, fs.attributes | (cvs\_rep \frown \langle CVSROOT \rangle \;\mathsf{prefix}\; p) \bullet \\
\qquad (((fs.attributes\, p).uid) = cvsperm2uid(cvs\_admin) \land \\
\qquad (((fs.attributes\, p).gid) = cvsperm2gid(cvs\_admin) \lor \\
\qquad\quad ((fs.attributes\, p).gid) = cvsperm2gid(cvs\_public)) \land \\
\qquad ((p \;\mathsf{is\_dir\_in}\; fs.files \land ((fs.attributes\, p).perm) = \\
\qquad\quad \{ru, wu, xu, xg, rg, wg, sg\}) \lor \\
\qquad\quad (\neg (p \;\mathsf{is\_dir\_in}\; fs.files) \land \{ru, rg\} \subseteq ((fs.attributes\, p).perm) \\
\qquad\quad \land ((fs.attributes\, p).perm) \subseteq \{xu, xg, xo, ru, rg, ro, sg\})))) \\
\forall fs : FileSystem \bullet attributes\_outside\_root(fs) \iff \\
(\quad \forall p : \mathsf{dom}\, fs.attributes\; ;\; pms : \mathbb{P}\, Perm | (cvs\_rep \;\mathsf{prefix}\; p) \land \\
\qquad \neg (cvs\_rep \frown \langle CVSROOT \rangle \;\mathsf{prefix}\; p) \bullet \\
\qquad\quad (((p \;\mathsf{is\_dir\_in}\; fs.files \land ((fs.attributes\, p).perm) = \\
\qquad\qquad \{ru, wu, xu, xg, rg, wg, sg, xo\}) \\
\qquad\qquad \lor ((p \;\mathsf{is\_dir\_in}\; fs.files) \land \{ru, rg\} \subseteq ((fs.attributes\, p).perm) \\
\qquad\qquad \land ((fs.attributes\, p).perm) \subseteq \{xu, xg, xo, ru, rg, ro, sg\})))
\end{array}
$$

These predicates are now used to defined the state invariants of the CVS file system state $CvsFileSystem$:

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

5.2.4 Auxiliary Functions on the CVS Filesystem State

Before we describe in the next section, the operations of the CVS-Server including its internal security architecture (login, add, checkout, update, and commit), we need, as a prerequisite, to model the access to the CVS authentication table that is part of the $cvs\_rep \frown CVSROOT$

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 standard user of the system.

$$
\begin{array}{|l}
get\_auth\_tab : FILESYS\_TAB \rightarrow AUTH\_TAB \\
\hline
\forall fs : FILESYS\_TAB \bullet \\
\quad get\_auth\_tab(fs) = (\mathbf{let}\, p == cvs\_rep \frown \langle CVSROOT, cvsauth \rangle \bullet \\
\qquad\qquad\qquad \mathbf{let}\, m == (\mu x : \mathsf{dom}\, auth\_of | Inl(x) = fs(p)) \bullet \\
\qquad\qquad\qquad\quad auth\_of(m))
\end{array}
$$

5.2.5 The Operations on CVS-Server

Now we have an established basis for defining the operations on the combined Unix and CVS environment. The first operation treats the login w.r.t. the CVS server and essentially updates the password table *cvs_passwd*, provided that for this combination the server authentication does not fail.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\text{---}\ cvs\_login\ \text{---}} \\
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
cvs\_uid? : Cvs\_Uid \\
cvs\_pwd? : Cvs\_Passwd \\
\hline
(cvs\_uid?, cvs\_pwd?) \in \mathsf{dom}(get\_auth\_tab\, files) \\
cvs\_passwd' = cvs\_passwd \oplus \{cvs\_uid? \mapsto cvs\_pwd?\} \\
wcs\_attributes' = wcs\_attributes \\
\theta FileSystem = \theta(FileSystem)' \\
\hline
\end{array}
$$

The add operation adds new files or directories in the working copy into the repository. add owns a mechanism for deducing the permissions of an item to be added from its context in the working copy (i.e. its father directory). For the permissions a valid authorization is required. We restrict our concrete model of the CVS add operation to argument paths of length one — this restriction is used for simplification reasons without loss of generality and loss of power of our security results. Further preconditions are:

- The *working directory* must be *valid*, i.e. point to a legal directory in the filesystem (this mirrors the reality in Unix: it is, in principle, possible that another client process removes the working directory — which leads usually to error messages).
- We require that the file to be added is fresh, i.e. does not yet exist in the repository.
- Since CVS implements the attribute table in a collection of files in the CVS subdirectories in the working copies, we require that the working directory is readable and the item to be added is writable provided it is a directory (this mirrors the reality in CVS but does not play any role in our model).

In order to structure the operation specification, we will apply a standard technique: we introduce an operation schema with normal behavior and another one capturing the exceptional behavior. We employ the schema calculus to combine (in this case disjoin) the two schemas and form the operation schema *cvs_add*. This technique will also be applied to the other operations of our model.

cvs_add_normal

$$
\begin{array}{l}
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
wdir \ \mathsf{is\_dir\_in}\ files \\
has\_r\_access(uid, wdir, attributes) \\
(wdir \frown p? \ \mathsf{is\_dir\_in}\ files) \Rightarrow \\
\quad has\_w\_access(uid, wdir \frown p?, attributes) \\
\#p? = 1 \\
cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p? \notin \mathsf{dom}\, files \\
has\_w\_access(\mathbf{let}\, fu == (wcs\_attributes\, wdir).f\_uid \bullet \\
\quad \mathbf{let}\, perm == get\_auth\_tab(files)(fu, cvs\_passwdfu) \bullet \\
\qquad cvsperm2uidperm, \\
\qquad\quad cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p?, attributes) \\
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd
\end{array}
$$

cvs_add_error

$$
\begin{array}{l}
\Xi Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
\neg wdir \ \mathsf{is\_dir\_in}\ files \\
\quad \vee \neg has\_r\_access(uid, wdir, attributes) \\
\quad \vee \neg ((wdir \frown p? \ \mathsf{is\_dir\_in}\ files) \Rightarrow \\
\qquad has\_w\_access(uid, wdir \frown p?, attributes)) \\
\quad \vee \neg \#p? = 1 \\
\quad \vee \neg cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p? \notin \mathsf{dom}\, files \\
\quad \vee \neg has\_w\_access(\mathbf{let}\, fu == (wcs\_attributes\, wdir).f\_uid \bullet \\
\qquad \mathbf{let}\, perm == get\_auth\_tab(files)(fu, cvs\_passwdfu) \bullet \\
\qquad\quad cvsperm2uidperm, \\
\qquad\qquad cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p?, attributes)
\end{array}
$$

$$cvs\_add == cvs\_add\_normal \vee cvs\_add\_error$$

The following operation is the most crucial one: the checkout from the repository. It requires that the given path $p?$ is actually defined in the repository and that the (CVS) client has write permission on the working directory. Then, all files below $p?$ for which the client (*cvs_uid*) has access, will be copied relative to the working directory, where the copy gets the Unix permissions according to the client's *umask* and the *wcs_attributes* in *Cvs_FileSystem*.

As a prerequisite, we define a function *rep_access* that computes the paths into the repository to which the client has read-access according to his CVS role (which is determined from the CVS user ID and password).[7]

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

Additionally, we define a function *choose* which deduces the CVS user ID for a path which was necessary to check out this path, or is necessary for a consecutive check in of that path.

$$
\begin{array}{l}
choose : (Cvs\_FileSystem \times Path) \rightarrow Cvs\_Uid \\
\hline
\forall fs : Cvs\_FileSystem \,;\, q : Path \bullet choose(fs, q) = (\mu id : \mathsf{dom}\, fs.cvs\_passwd | \\
\quad (fs.attributes(cvs\_rep \mathbin{^\frown} q)).uid = \\
\qquad cvsperm2uid(get\_auth\_tab(fs.files)(id, fs.cvs\_passwd(id))) \\
\quad \land(fs.attributes(cvs\_rep \mathbin{^\frown} q)).gid = \\
\qquad cvsperm2gid(get\_auth\_tab(fs.files)(id, fs.cvs\_passwd(id))))
\end{array}
$$

Now we can define the checkout operation. Additionally, we require that the directory we wan to check out must exist in the top-level directory of the repository and that we have not checked it out yet.

[7] We do not model here the fact that the working copy may already exist and that CVS roles in it may be available; here, we model that in such cases an existing working copy is simply overwritten, and we (safely) assume that this is the implemented behavior.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_co}}\\
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
cvs\_rep \frown p? \in \mathsf{dom}\, files \\
\forall q : rep\_access(\theta Cvs\_FileSystem)(p?) \bullet wdir \frown q \notin \mathsf{dom}\, files \\
has\_w\_access(uid, wdir, attributes) \\
files' = files \oplus \{q : rep\_access(\theta Cvs\_FileSystem)(p?) \bullet \\
\qquad wdir \frown q \mapsto files(cvs\_rep \frown q)\} \\
attributes' = attributes \oplus \{q : rep\_access(\theta Cvs\_FileSystem)(p?) \bullet \\
\qquad wdir \frown q \mapsto \langle\!|perm == \emptyset, uid == uid, gid == gid|\!\rangle\} \\
wcs\_attributes' = wcs\_attributes \oplus \{q : rep\_access(\theta Cvs\_FileSystem)(p?) \bullet \\
\qquad wdir \frown q \mapsto \langle\!|rep == q, f\_uid == choose(\theta Cvs\_FileSystem, q)|\!\rangle\} \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

The following operations use the current working directory as a default parameter, e.g. the checkin operation tries to check in every file and recursively every subdirectory in the working directory ($wdir$) unless we restrict it with the input parameter $p?$ to some subdirectory. These operations do not fail if the user only has sufficient permissions for some files. Instead, it ignores all other files and directories and only performs a partial update or checkin.

The *cvs_update* operation schema, which similar to *cvs_checkout* except that the files to be updated can already exist in the working copy. As for the add operation, we use the schema calculus to model normal behavior and exceptional behavior, and then to conjoin these schemas to form the *cvs_up* operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvsUp}}\\
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
cvs\_rep \frown repOf(wcs\_attributes\, wdir) \frown p? \in \mathsf{dom}\, files \\
has\_w\_access(uid, wdir \frown p?, attributes) \\
files' = files \oplus \{q : rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?) \bullet \\
\qquad wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep) \mapsto files(cvs\_rep \frown q)\} \\
attributes' = attributes \oplus \\
\qquad \{q : rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?) \bullet \\
\qquad\qquad wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep) \\
\qquad\qquad\qquad \mapsto \langle\!|perm == \emptyset, uid == uid, gid == gid|\!\rangle\} \\
wcs\_attributes' = wcs\_attributes \cup \\
\qquad \{q : rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?) | \\
\qquad\qquad wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep) \notin \mathsf{dom}\, wcs\_attributes \bullet \\
\qquad\qquad\qquad wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep) \\
\qquad\qquad\qquad\qquad \mapsto \langle\!|rep == q, f\_uid == choose(\theta Cvs\_FileSystem, q)|\!\rangle\} \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

The schema $cvsUpNoAct$ defines the condition that the input file name does not exist in the repository and nothing happens.

$$
\begin{array}{|l}
\hline
\textit{cvsUpNoAct} \\
\Xi Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
cvs\_rep \frown repOf(wcs\_attributes\ wdir) \frown p? \notin \mathsf{dom}\ files \\
\hline
\end{array}
$$

The schema $cvsUpErr$ defines the behavior when the client does not have sufficient write access in the working copy. The operation outputs an error message and does nothing else.

$$
\begin{array}{|l}
\hline
\textit{cvsUpErr} \\
\Xi Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
error! : \mathsf{denotation} \\
\hline
\neg(has\_w\_access(uid, wdir \frown p?, attributes)) \\
error! = \texttt{"No write permission!"} \\
\hline
\end{array}
$$

$$cvs\_update == cvsUpNoAct \lor cvsUp \lor cvsUpErr$$

The checkin command is modeled by the schema cvs_ci. Here we require that the client has read access for the file or directory in the current working directory and sufficient permissions to modify the repository.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_ci}} \\
\hline
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
p? : Path \\
\hline
has\_r\_access(uid, wdir \frown p?, attributes) \\
wdir \in \mathrm{dom}\, wcs\_attributes \\
files' = files \oplus \{q : rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?) \mid \\
\qquad has\_r\_access(uid, wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep), attributes) \bullet \\
\qquad\qquad cvs\_rep \frown q \mapsto files(wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep))\} \\
attributes' = \\
\qquad attributes \oplus \{q : rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?) \mid \\
\qquad has\_r\_access(uid, wdir \frown cutPath(q, (wcs\_attributes\, wdir).rep), attributes) \bullet \\
\qquad\qquad cvs\_rep \frown q \mapsto \langle\!| perm == \{ru, rg\}, \\
\qquad\qquad\qquad\qquad uid == cvsperm2uid(get\_auth\_tab(files)((wcs\_attributes\, q).f\_uid, \\
\qquad\qquad\qquad\qquad\qquad\qquad cvs\_passwd((wcs\_attributes\, q).f\_uid))), \\
\qquad\qquad\qquad\qquad gid == cvsperm2gid(get\_auth\_tab(files)((wcs\_attributes\, q).f\_uid, \\
\qquad\qquad\qquad\qquad\qquad\qquad cvs\_passwd((wcs\_attributes\, q).f\_uid))) |\!\rangle\} \\
wcs\_attributes' = wcs\_attributes \land cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

Operations lifted from the UNIX level

The following schemas are used to lift the Unix filesystem operations (which were defined in terms of the $FileSystem$ state) to the $Cvs\_FileSystem$ state. This is only done to get a uniform specification. We can then reason about traces of mixed Unix and CVS operations over the same state. All schemas follow the same model: they introduce $Cvs\_FileSystem$ and the actual Unix operation and keep CVS-specific components of $Cvs\_FileSystem$ constant.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_cd}} \\
\hline
\Delta Cvs\_FileSystem \\
cd \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_mkdir}} \\
\hline
\Delta Cvs\_FileSystem \\
mkdir \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_mkfile}} \\
\hline
\Delta Cvs\_FileSystem \\
mkfile \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_access}} \\
\hline
\Delta Cvs\_FileSystem \\
access \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_write}} \\
\hline
\Delta Cvs\_FileSystem \\
write \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_rm}} \\
\hline
\Delta Cvs\_FileSystem \\
rm \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_rmdir}} \\
\hline
\Delta Cvs\_FileSystem \\
rmdir \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_mv}} \\
\hline
\Delta Cvs\_FileSystem \\
mv \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_chown}} \\
\hline
\Delta Cvs\_FileSystem \\
chown \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_chmod}} \\
\hline
\Delta Cvs\_FileSystem \\
chmod \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{cvs\_setumask}} \\
\hline
\Delta Cvs\_FileSystem \\
setumask \\
\hline
wcs\_attributes' = wcs\_attributes \\
cvs\_passwd' = cvs\_passwd \\
\hline
\end{array}
$$

6 Verifying the Consistency and the Refinement

6.1 Consistency Conditions

Two types of consistency conditions that we expect in general in a specification, can be listed:

1. Logical consistency. This comprises:
    a) The *conservativity* of all axiomatic definitions, and
    b) the *definedness* of all relational applications $f(x)$.
2. Methodological consistency. This comprises the *deadlock-$freeness$* of all operations.

Conservativity can be assured if it is syntactically checked that all axiomatic definitions belong to one of the following *conservative axiom schemes* described by [14], which also form the basis of the Isabelle/HOL libraries. Here, we briefly review the five most essential schemes:

1. The *constant definition*: An equational, non-recursive axiom introducing a fresh constant as an "abbreviation" of already defined concepts.
2. The *constant specification*: An arbitrary predicate $P(c)$ over a fresh constant c; as proof obligation, it remains to show that there is a witness of the predicate: $\exists x.P(x)$.
3. The *type definition*: A fresh type constructor is defined via an isomorphism to a set of values defined before.
4. The *well-founded recursion* scheme: A recursive axiom for a fresh constant that requires that it is applied to "less" arguments in all occurrences on the right hand side, where "less" is understood in terms of a preconceived well-founded ordering.
5. The *primitive recursion* scheme: This is a special case of the previous definition scheme.

Note that the latter two schemes were mapped to constant definitions in Isabelle/HOL; the necessary proof work is performed by the system behind the scene. Unfortunately, currently HOL-Z does not automatically check the side-conditions of conservativity according to these schemes (as, in contrast, Isabelle/HOL does). Hence, throughout this paper, these checks have been established by reviews — a mechanical check for these conditions, although highly desirable since specifications are constantly changed during development, has not been implemented so far.

The second type of logical inconsistency condition is not crucial from a HOL point of view; relational application is defined by the Hilbert operator:

```
f(x) == @ y. (x,y) : f
```

such that from the HOL point-of-view, these expressions are well-defined – but arbitrary. With respect to Z semantics, this kind of expression is *undefined*. This means that Z is semantically more restrictive, and these situations should be avoided. Thus, in the case of HOL-Z, this boils down to the proof that for all relational applications $f(x)$, x actually belongs to the declared domain of f.

Methodological consistency is unproblematic from the logical point of view — methodologically inconsistent operations are just non-left-total relations —, but from the pragmatical point of view, specifications of operations that represent a deadlock are simply "wrong" in the sense that they do not model real operational behavior. Therefore, checks of this kind help to find errors in the specification. Here, "deadlock-freeness" of an operation means that for any *legal* input $x?$ and a given *state*, there must be at least one possible output $y!$ (if output parameters were defined) and a successor state $state'$. In other words, whenever an operation is given legal input in a consistent state, a transition to a consistent successor state is assured. This condition can be compactly represented by the schema operator *preop* for any given schema *op*, which is equivalent to $\exists y! state'.op$. The question remains to settle what "legal" input $x?$ is. As such, we consider input that fulfills all constraints in the body of a schema, more precisely, all these conjoints in the body, that contain only occurrences of input variables (i.e. variables with a ?-suffix) or state variables (i.e. variables that do not have a stroke-suffix or an exclamation mark suffix). We call the conjunction of all conjoints fulfilling this condition the *syntactic precondition syn_pre* of an operation schema *op*. On this basis, we can define deadlock-freeness schematically as:

$$df_{op} \equiv syn\_pre_{op} \Rightarrow preop$$

In a backward proof, a theorem of deadlock-freeness can be used to reduce the proof-task *preop* just to the syntactic precondition. It turns out that this step is crucial for proving the first refinement condition (see Section 6.2). Therefore, theorems of deadlock-freeness

6.1.1 The Conservativity of the System Architecture

First, recall that all schema definitions and all schema logical definitions are non-recursive definitions as "sets of records", thus constant definitions. Therefore, only axiomatic definitions remain to consider. Manually, it is easy to check that only non-polymorphic constant definitions are used throughout all axiomatic definitions. There are two exceptions:

1. The specification of *cvs_perm_order* in Section 4.1, which is in fact a constant specification: recall that *cvs_perm_order* must be a partial order (reflexive, transitive, antisymmetric) with *cvs_public* as least and *cvs_admin* as greatest elements. Such an ordering exists; witness for it is just the trivial ordering $\{(cvs\_public, cvs\_admin)\}$.
2. The operation *cutPath* is a relict due to laziness: it can be reformulated as constant definition easily via the Hilbert-Operator.

6.1.2 The Definedness of the System Architecture

In the following, we will list the definedness conditions. The proofs (mostly done by H. Hiss in his "Studienarbeit" are contained in special `.ML`-files contained in the HOL-Z distribution (see `examples/zeta/cvs-server/holz`). The proofs are in their vast majority just calls of Isabelle's `auto`-tactics.

The following definedness consistency proof-obligations obey the following naming conventions:

1. consistency conditions on schemas are named after the schema they belong to.
2. consistency conditions arising in axiomatic definitions are named after either
   - after the constant/function/relation name which contains the critical application
   - if there is none (e.g., the condition is caused by the predicate part of axdef), the first name of a Name/function/relation names defined in the same axdef.
3. In front of each group of consistency conditions, we write [Schema] or [Axiom], depending on what the group concerns. So it should be easy to review all schema or axiom -specific consistency conditions.

The conditions are generated by the hLaTeX-style `holz/src/latex/holz.dtx`.
section *SysArchConsistency* **parents** *AbsOperations*, *FileSystem*, *CVSServer*

$abs\_cvsauth\_cc1$

$$abs\_cvsauth\_cc1\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (abs\_cvsauth\_cc1(n) \iff (\forall x : \mathsf{dom}\, abs\_auth\_of \bullet x \in \mathsf{dom}\, abs\_auth\_of))$$

$abs\_cvsauth\_cc2$

$$abs\_cvsauth\_cc2\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (abs\_cvsauth\_cc2(n) \iff (\mathsf{ran}\, abs\_auth\_of \subseteq \mathsf{dom}\, abs\_data\_of))$$

$abs\_cvsauth\_cc3$

$$abs\_cvsauth\_cc3\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (abs\_cvsauth\_cc3(n) \iff (\forall x : AUTH\_TAB \bullet x \in \mathsf{dom}\, abs\_data\_of))$$

$abs\_cvsauth\_cc4$

$$abs\_cvsauth\_cc4\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (abs\_cvsauth\_cc4(n) \iff (\mathsf{ran}\, abs\_data\_of \subseteq \mathsf{dom}\, abs\_auth\_of))$$

$abs\_cvsauth\_cc5$

$$\begin{array}{|l}
abs_cvsauth_cc5_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (abs_cvsauth_cc5(n) \iff \\
\quad (\forall x : ABS_DATATAB \bullet \\
\qquad x \in \mathrm{dom}\, authtab))
\end{array}$$

$abs\_cvsauth\_cc6$

$$\begin{array}{|l}
abs_cvsauth_cc6_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (abs_cvsauth_cc6(n) \iff \\
\quad (\forall x : ABS_DATATAB \bullet (abs_cvsauth \in \mathrm{dom}\, x)))
\end{array}$$

$abs\_cvsauth\_cc7$

$$\begin{array}{|l}
abs_cvsauth_cc7_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (abs_cvsauth_cc7(n) \iff \\
\quad (\forall r : ABS_DATATAB \bullet \\
\qquad (\mathrm{ran}\, r \subseteq \mathrm{dom}\, abs_auth_of)))
\end{array}$$

abs_login_cc

$$\begin{array}{l}
abs_login_cc1 == \forall abs_login \bullet \\
[\Delta ClientState \,;\, \Xi RepositoryState \,;\, uid? : Cvs_Uid \\
;\, passwd? : Cvs_Passwd | rep \in \mathrm{dom}\, authtab]
\end{array}$$

abs_ci_cc

$$\begin{array}{l}
abs_ci_cc1 == \forall abs_ci \bullet \\
[\Xi ClientState \,;\, \Delta RepositoryState \,;\, files? \\
: \mathbb{P}\, Abs_Name | \forall n : wfiles \bullet n \in \mathrm{dom}\, wc_uidtab]
\end{array}$$

abs_ci_cc

$$\begin{array}{l}
abs_ci_cc2 == \forall abs_ci \bullet \\
[\Xi ClientState \,;\, \Delta RepositoryState \\
;\, files? : \mathbb{P}\, Abs_Name | rep \in \mathrm{dom}\, authtab]
\end{array}$$

abs_ci_cc

$abs\_ci\_cc3 == \forall abs\_ci \bullet$
$[\Xi ClientState ; \Delta RepositoryState ; files?$
$: \mathbb{P}\, Abs\_Name | \forall n : wfiles \bullet n \in \mathsf{dom}\, rep\_permtab]$

abs_ci_cc

$abs\_ci\_cc4 == \forall abs\_ci \bullet$
$[\Xi ClientState ; \Delta RepositoryState ; files? : \mathbb{P}\, Abs\_Name | \forall f$
$: (\mathsf{ran}\, authtab) \bullet \forall x : (\mathsf{ran}\, wc\_uidtab) \bullet x \in \mathsf{dom}\, f]$

abs_up_cc

$abs\_up\_cc1 == \forall abs\_up \bullet$
$[\Xi ClientState ; \Delta RepositoryState ; files?$
$: \mathbb{P}\, Abs\_Name | \forall n : wfiles \bullet n \in \mathsf{dom}\, wc\_uidtab]$

abs_up_cc

$abs\_up\_cc2 == \forall abs\_up \bullet$
$[\Xi ClientState ; \Delta RepositoryState$
$; files? : \mathbb{P}\, Abs\_Name | rep \in \mathsf{dom}\, authtab]$

abs_up_cc

$abs\_up\_cc3 == \forall abs\_up \bullet$
$[\Xi ClientState ; \Delta RepositoryState ; files?$
$: \mathbb{P}\, Abs\_Name | \forall n : wfiles \bullet n \in \mathsf{dom}\, rep\_permtab]$

abs_up_cc

$abs\_up\_cc4 == \forall abs\_up \bullet$
$[\Xi ClientState ; \Delta RepositoryState ; files? : \mathbb{P}\, Abs\_Name | \forall f$
$: (\mathsf{ran}\, authtab) \bullet \forall x : (\mathsf{ran}\, wc\_uidtab) \bullet x \in \mathsf{dom}\, f]$

is_in_cc1

$$\begin{array}{|l}
is_in_cc1_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (is_in_cc1(n) \iff \\
\quad (\forall fs : (Path \pfun (Data + Unit)) \bullet \forall d : Path \bullet (d \mathsf{\ is_in\ } fs) \Rightarrow (d \in \mathsf{dom}\, fs)))
\end{array}$$

is_in_cc2

$$\begin{array}{|l}
is_in_cc2_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (is_in_cc2(n) \iff \\
\quad (\forall u : Unit \bullet u \in \mathsf{dom}\, Inr[Data, Unit]))
\end{array}$$

$FileSystem\_cc$

$$FileSystem\_cc1 == \forall FileSystem \bullet$$
$$[files : FILESYS\_TAB\ ;\ attributes$$
$$: FILEATTR\_TAB | \forall p : \mathsf{dom}\, files \bullet p \in \mathsf{dom}\, front]$$

has_attrib_cc1

$$\begin{array}{|l}
has_attrib_cc1_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (has_attrib_cc1(n) \iff \\
\quad (\forall uid : Uid \bullet uid \in \mathsf{dom}\, groups))
\end{array}$$

has_attrib_cc2

$$\begin{array}{|l}
has_attrib_cc2_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (has_attrib_cc2(n) \iff \\
\quad (\forall fa : FILEATTR_TAB \bullet \forall p : Path \bullet p \in \mathsf{dom}\, fa))
\end{array}$$

But the following condition will for sure be watched again, as the front operator is not defined on empty sequences, and p may be the empty sequence.

$has\_w\_access\_cc1$

$$\begin{array}{|l}
has_w_access_cc1_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (has_w_access_cc1(n) \iff \\
\quad (\forall p : Path \bullet p \in \mathsf{dom}\, front))
\end{array}$$

$mkdir\_cc$

$mkdir\_cc1 == \forall mkdir \bullet$
$[\Delta FileSystem \,;\, \Xi ProcessState \,;\, u? : Name | Nil \in \mathsf{dom}\, Inr[Data, Unit]]$

$mkdir\_cc$

$mkdir\_cc2 == \forall mkdir \bullet$
$[\Delta FileSystem \,;\, \Xi ProcessState \,;\, u? : Name | wdir \in \mathsf{dom}\, attributes]$

$mkfile\_cc$

$mkfile\_cc1 == \forall mkfile \bullet$
$[\Delta FileSystem \,;\, \Xi ProcessState \,;\, u? : Name$
$;\, d? : Data | d? \in \mathsf{dom}\, Inl[Data, Unit]]$

$mkfile\_cc$

$mkfile\_cc2 == \forall mkfile \bullet$
$[\Delta FileSystem \,;\, \Xi ProcessState \,;\, u?$
$: Name \,;\, d? : Data | wdir \in \mathsf{dom}\, attributes]$

$access\_cc$

$access\_cc1 == \forall access \bullet$
$[\Xi FileSystem \,;\, \Xi ProcessState \,;\, u? : Name$
$;\, c! : (Data + Unit) | (wdir \frown \langle u? \rangle) \in \mathsf{dom}\, files]$

$write\_cc$

$write\_cc1 == \forall write \bullet$
$[\Delta FileSystem \,;\, \Xi ProcessState \,;\, u? : Name$
$;\, d? : Data | d? \in \mathsf{dom}\, Inl[Data, Unit]]$

mv_cc

$$mv\_cc1 == \forall mv \bullet$$
$$[\Delta FileSystem ; \Xi ProcessState ; u1? : Name$$
$$; u2? : Name | (wdir \frown \langle u1? \rangle) \in \mathsf{dom}\, files]$$

mv_cc

$$mv\_cc2 == \forall mv \bullet$$
$$[\Delta FileSystem ; \Xi ProcessState ; u1? : Name$$
$$; u2? : Name | (wdir \frown \langle u1? \rangle) \in \mathsf{dom}\, attributes]$$

$chown\_cc$

$$chown\_cc1 == \forall chown \bullet$$
$$[\Delta FileSystem ; \Xi ProcessState ; p? : Path$$
$$; gr? : Gid ; ow? : Uid | p? \in \mathsf{dom}\, attributes]$$

$chown\_cc$

$$chown\_cc2 == \forall chown \bullet$$
$$[\Delta FileSystem ; \Xi ProcessState ; p? : Path$$
$$; gr? : Gid ; ow? : Uid | uid \in \mathsf{dom}\, groups]$$

$chmod\_cc$

$$chmod\_cc1 == \forall chmod \bullet$$
$$[\Delta FileSystem ; \Xi ProcessState ; ps?$$
$$: \mathbb{P}\, Perm ; p? : Path | p? \in \mathsf{dom}\, attributes]$$

$cvsperm2gid\_cc1$

$$cvsperm2gid\_cc1\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (cvsperm2gid\_cc1(n) \iff (users \subseteq \mathsf{dom}\, groups))$$

$cvsperm2uid\_cc1$

$$
\begin{array}{|l}
cvsperm2uid\_cc1\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvsperm2uid\_cc1(n) \iff \\
\quad (Cvs\_Perm \subseteq \mathsf{dom}\, cvsperm2uid))
\end{array}
$$

$cvsperm2uid\_cc2$

$$
\begin{array}{|l}
cvsperm2uid\_cc2\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvsperm2uid\_cc2(n) \iff \\
\quad (Cvs\_Perm \subseteq \mathsf{dom}\, cvsperm2gid))
\end{array}
$$

cvs_rep_cc1

$$
\begin{array}{|l}
cvs\_rep\_cc1\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvs\_rep\_cc1(n) \iff \\
\quad (Data \subseteq \mathsf{dom}\, auth\_of))
\end{array}
$$

cvs_rep_cc2

$$
\begin{array}{|l}
cvs\_rep\_cc2\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvs\_rep\_cc2(n) \iff \\
\quad (\mathsf{ran}\, auth\_of \subseteq \mathsf{dom}\, data\_of))
\end{array}
$$

cvs_rep_cc3

$$
\begin{array}{|l}
cvs\_rep\_cc3\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvs\_rep\_cc3(n) \iff \\
\quad (AUTH\_TAB \subseteq \mathsf{dom}\, data\_of))
\end{array}
$$

cvs_rep_cc4

$$
\begin{array}{|l}
cvs\_rep\_cc4\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (cvs\_rep\_cc4(n) \iff \\
\quad (\mathsf{ran}\, data\_of \subseteq \mathsf{dom}\, auth\_of))
\end{array}
$$

$Cvs\_FileSystem\_cc$

$$Cvs\_FileSystem\_cc1 == \forall Cvs\_FileSystem \bullet$$
$$[FileSystem ; wcs\_attributes : CVS\_ATTR\_TAB ; cvs\_uid$$
$$: Cvs\_Uid ; passwd : Cvs\_Passwd | cvs\_rep \in \mathsf{dom}\, attributes]$$

$Cvs\_FileSystem\_cc$

$$Cvs\_FileSystem\_cc2 == \forall Cvs\_FileSystem \bullet$$
$$[FileSystem ; wcs\_attributes : CVS\_ATTR\_TAB ; cvs\_uid$$
$$: Cvs\_Uid ; passwd : Cvs\_Passwd | cvs\_admin \in \mathsf{dom}\, cvsperm2uid]$$

$Cvs\_FileSystem\_cc$

$$Cvs\_FileSystem\_cc3 == \forall Cvs\_FileSystem \bullet$$
$$[FileSystem ; wcs\_attributes : CVS\_ATTR\_TAB ; cvs\_uid$$
$$: Cvs\_Uid ; passwd : Cvs\_Passwd | cvs\_public \in \mathsf{dom}\, cvsperm2gid]$$

$Cvs\_FileSystem\_cc$

$$Cvs\_FileSystem\_cc4 == \forall Cvs\_FileSystem \bullet$$
$$[FileSystem ; wcs\_attributes : CVS\_ATTR\_TAB ; cvs\_uid$$
$$: Cvs\_Uid ; passwd : Cvs\_Passwd | \mathsf{dom}\, files \subseteq \mathsf{dom}\, attributes]$$

$Cvs\_FileSystem\_cc$

$$Cvs\_FileSystem\_cc5 == \forall Cvs\_FileSystem \bullet$$
$$[FileSystem ; wcs\_attributes : CVS\_ATTR\_TAB ; cvs\_uid$$
$$: Cvs\_Uid ; passwd : Cvs\_Passwd | Uid \subseteq \mathsf{dom}\, groups]$$

$get\_auth\_tab\_cc1$

$$get\_auth\_tab\_cc1\_ : \mathbb{P}\,\mathbb{N}$$
$$\forall n : \mathbb{N} \bullet (get\_auth\_tab\_cc1(n) \iff$$
$$(Data \subseteq \mathsf{dom}\, Inl[Data, \mathbb{P}\, Name]))$$

$get\_auth\_tab\_cc2$

$$
\begin{array}{|l}
get\_auth\_tab\_cc2\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (get\_auth\_tab\_cc2(n) \iff \\
\quad (\forall c : FILESYS\_TAB \bullet cvs\_rep \frown \langle CVSROOT, cvsauth \rangle \in \mathsf{dom}\, c))
\end{array}
$$

$get\_auth\_tab\_cc3$

$$
\begin{array}{|l}
get\_auth\_tab\_cc3\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (get\_auth\_tab\_cc3(n) \iff \\
\quad (Data \subseteq \mathsf{dom}\, auth\_of))
\end{array}
$$

cvs_login_cc

$$
\begin{array}{l}
cvs\_login\_cc1 == \forall cvs\_login \bullet \\
[\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, cvs\_uid? : Cvs\_Uid \\
;\, cvs\_pwd? : Cvs\_Passwd | files \in \mathsf{dom}\, get\_auth\_tab]
\end{array}
$$

rep_access_cc1

$$
\begin{array}{|l}
rep\_access\_cc1\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (rep\_access\_cc1(n) \iff \\
\quad (\forall x : Cvs\_FileSystem \bullet x \in \mathsf{dom}\, rep\_access))
\end{array}
$$

rep_access_cc2

$$
\begin{array}{|l}
rep\_access\_cc2\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (rep\_access\_cc2(n) \iff \\
\quad (\forall f : \mathsf{ran}\, rep\_access \bullet Path \subseteq \mathsf{dom}\, f))
\end{array}
$$

rep_access_cc3

$$
\begin{array}{|l}
rep\_access\_cc3\_ : \mathbb{P}\,\mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet (rep\_access\_cc3(n) \iff \\
\quad (\forall cfs : Cvs\_FileSystem \bullet (cfs.cvs\_uid, cfs.passwd) \in \mathsf{dom}(get\_auth\_tab(cfs.files))))
\end{array}
$$

rep_access_cc4

$$\begin{array}{|l} rep\_access\_cc4\_ : \mathbb{P}\,\mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet (rep\_access\_cc4(n) \iff \\ \quad (FILESYS\_TAB \subseteq \mathsf{dom}\, get\_auth\_tab)) \end{array}$$

cvs_co_cc

$$cvs\_co\_cc1 == \forall cvs\_co \bullet \\ [\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, p? \\ : Path | \theta Cvs\_FileSystem \in \mathsf{dom}\, rep\_access]$$

cvs_co_cc

$$cvs\_co\_cc2 == \forall cvs\_co \bullet \\ [\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, p? \\ : Path | \operatorname{seq} Name \subseteq \mathsf{dom}(rep\_access(\theta Cvs\_FileSystem))]$$

cvs_update_cc

$$cvs\_update\_cc1 == \forall cvs\_update \bullet \\ [\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, p? \\ : Path | \theta Cvs\_FileSystem \in \mathsf{dom}\, rep\_access]$$

cvs_update_cc

$$cvs\_update\_cc2 == \forall cvs\_update \bullet \\ [\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, p? \\ : Path | (\operatorname{seq}[Name]) \subseteq \mathsf{dom}(rep\_access(\theta Cvs\_FileSystem))]$$

cvs_update_cc

$$cvs\_update\_cc3 == \forall cvs\_update \bullet \\ [\Delta Cvs\_FileSystem \,;\, \Xi ProcessState \,;\, p? : Path | \forall q \\ : (rep\_access(\theta Cvs\_FileSystem)((wcs\_attributes\, wdir).rep \frown p?)) \\ \bullet (wdir \frown p? \frown q) \in \mathsf{dom}\, files]$$

cvs_ci_cc

$cvs\_ci\_cc1 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState$
$; p? : Path | wdir \in \mathsf{dom}\, wcs\_attributes]$

cvs_ci_cc

$cvs\_ci\_cc2 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState ; p?$
$: Path | \theta Cvs\_FileSystem \in \mathsf{dom}\, rep\_access]$

cvs_ci_cc

$cvs\_ci\_cc3 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState ; p?$
$: Path | \forall f : \mathsf{ran}\, rep\_access \bullet Path \subseteq \mathsf{dom}\, f]$

cvs_ci_cc

$cvs\_ci\_cc4 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState ; p? : Path | \forall q$
$: (rep\_access(\theta Cvs\_FileSystem)((wcs\_attributeswdir).rep))$
$\bullet (wdir \frown q) \in \mathsf{dom}\, files]$

cvs_ci_cc

$cvs\_ci\_cc5 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState ; p? : Path | files \in \mathsf{dom}\, get\_auth\_tab]$

cvs_ci_cc

$cvs\_ci\_cc6 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem ; \Xi ProcessState ; p?$
$: Path | (cvs\_uid, passwd) \in \mathsf{dom}(get\_auth\_tab(files))]$

cvs_ci_cc

$cvs\_ci\_cc7 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem\ ; \Xi ProcessState\ ; p?$
$: Path | Cvs\_Perm \subseteq \mathsf{dom}\, cvsperm2uid]$

cvs_ci_cc

$cvs\_ci\_cc8 == \forall cvs\_ci \bullet$
$[\Delta Cvs\_FileSystem\ ; \Xi ProcessState\ ; p?$
$: Path | Cvs\_Perm \subseteq \mathsf{dom}\, cvsperm2gid]$

6.1.3 The Deadlock-freeness of the Operation Schemas

Since the construction of these proof obligations (as described in the introductory section of this chapter) is canonical, we will simply list these operations below. Some proofs of them are contained in the in the HOL-Z distribution (see `examples/zeta/cvs-server/holz`). The proofs are highly non-trivial, since they usually require to show that the state invariants (i.e. of *FileSystem*,*ClientState* and *RepositoryState*) remain valid under transition.

abs_login_df

$abs\_login\_df == [ClientState\ ; RepositoryState\ ; uid?$
$: Cvs\_Uid\ ; passwd? : Cvs\_Passwd |] \Rightarrow \mathsf{pre}\, abs\_login$

abs_cd_df

$abs\_cd\_df == [ClientState\ ; RepositoryState$
$; wfiles? : \mathbb{P}\, Abs\_Name |] \Rightarrow \mathsf{pre}\, abs\_cd$

abs_add_df

$abs\_add\_df == [ClientState\ ; RepositoryState\ ; newfiles?$
$: ABS\_DATATAB | \mathsf{dom}\, newfiles? \cap \mathsf{dom}\, rep = \emptyset] \Rightarrow \mathsf{pre}\, abs\_add$

abs_ci_df

$abs\_ci\_df == [ClientState\ ; RepositoryState |] \Rightarrow \mathsf{pre}\, abs\_ci$

abs_up_df

$$abs\_up\_df == [ClientState \; ; RepositoryState \; ; wfiles : \mathbb{P}\, Abs\_Name |] \Rightarrow \mathsf{pre}\, abs\_up$$

These conditions correspond to the schema operations of the Prelude section.

cd_df

$$cd\_df == [FileSystem \; ; ProcessState \; ; u? : Path | (u? \in \mathsf{dom}\, files) \wedge (\forall p : Path | p \;\mathsf{prefix}\; u? \bullet has\_x\_attrib(uid, p, attributes)) \wedge (\forall p : Path | p \;\mathsf{prefix}\; u? \bullet p \;\mathsf{is\_dir\_in}\; files)] \Rightarrow \mathsf{pre}\, cd$$

mkdir_df

$$mkdir\_df == [FileSystem \; ; ProcessState \; ; u? : Name | (wdir \;\mathsf{is\_dir\_in}\; files) \wedge (\neg (wdir \frown \langle u? \rangle \;\mathsf{is\_dir\_in}\; (files))) \wedge has\_w\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, mkdir$$

mkfile_df

$$mkfile\_df == [FileSystem \; ; ProcessState \; ; u? : Name \; ; d? : Data | (wdir \;\mathsf{is\_dir\_in}\; files) \wedge (\neg (wdir \frown \langle u? \rangle \;\mathsf{is\_dir\_in}\; (files))) \wedge has\_w\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, mkfile$$

access_df

$$access\_df == [FileSystem \; ; ProcessState \; ; u? : Name | (wdir \frown \langle u? \rangle \;\mathsf{is\_dir\_in}\; files) \wedge has\_r\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, access$$

write_df

$$write\_df == [FileSystem \; ; ProcessState \; ; u? : Name \; ; d? : Data | (wdir \frown \langle u? \rangle \;\mathsf{is\_file\_in}\; files) \wedge has\_w\_access(uid, wdir \frown \langle u? \rangle, attributes)] \Rightarrow \mathsf{pre}\, write$$

rm_df

$$rm\_df == [FileSystem \; ; ProcessState \; ; u? : Name | (wdir \frown \langle u? \rangle \; \mathsf{is\_file\_in} \; files) \land has\_w\_access(uid, wdir, attributes) \land has\_w\_access(uid, wdir \frown \langle u? \rangle, attributes)] \Rightarrow \mathsf{pre}\, rm$$

rmdir_df

$$rmdir\_df == [FileSystem \; ; ProcessState \; ; u? : Name | (wdir \frown \langle u? \rangle \; \mathsf{is\_dir\_in} \; files) \land (\{x : Name | wdir \frown \langle u? \rangle \frown \langle x \rangle \; \mathsf{is\_dir\_in} \; (files)\} = \emptyset) \land has\_w\_access(uid, wdir, attributes) \land has\_w\_access(uid, wdir \frown \langle u? \rangle, attributes)] \Rightarrow \mathsf{pre}\, rmdir$$

mv_df

$$mv\_df == [FileSystem \; ; ProcessState \; ; u1? : Name \; ; u2? : Name | (wdir \frown \langle u1? \rangle \; \mathsf{is\_dir\_in} \; files) \land (\neg (wdir \frown \langle u2? \rangle \; \mathsf{is\_dir\_in} \; (files))) \land has\_w\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, mv$$

chown_df

$$chown\_df == [FileSystem \; ; ProcessState \; ; p? : Path \; ; gr? : Gid \; ; ow? : Uid | p? \in \mathsf{dom}(files) \land ((root = uid) \lor ((root \neq uid \land ow? = uid \land ow? = ((attributes\, p?).uid) \land gr? \in groups(uid))))] \Rightarrow \mathsf{pre}\, chown$$

chmod_df

$$chmod\_df == [FileSystem \; ; ProcessState \; ; ps? : \mathbb{P}\, Perm \; ; p? : Path | p? \in \mathsf{dom}(files) \land ((root = uid) \lor (root \neq uid \land uid = ((attributes\, p?).uid)))] \Rightarrow \mathsf{pre}\, chmod$$

setumask_df

$$setumask\_df == [FileSystem \; ; ProcessState \; ; u? : \mathbb{P}(Perm \setminus \{sg\}) |] \Rightarrow \mathsf{pre}\, setumask$$

These conditions correspond to the schema operations of the CVSServer section.

cvs_login_df

$$cvs\_login\_df == [Cvs\_FileSystem \ ; ProcessState \ ; cvs\_uid? : Cvs\_Uid \ ; cvs\_pwd? : Cvs\_Passwd | (cvs\_uid?, cvs\_pwd?) \in \mathsf{dom}(get\_auth\_tab\, files)] \Rightarrow \mathsf{pre}\, cvs\_login$$

cvs_co_df

$$cvs\_co\_df == [Cvs\_FileSystem \ ; ProcessState \ ; p? : Path | cvs\_rep \frown p? \in \mathsf{dom}\, files \land (\forall q : rep\_access(\theta Cvs\_FileSystem)(p?) \bullet wdir \frown q \notin \mathsf{dom}\, files) \land has\_w\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, cvs\_co$$

cvs_update_df

$$cvs\_update\_df == [Cvs\_FileSystem \ ; ProcessState \ ; p? : Path | cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p? \in \mathsf{dom}\, files \land has\_w\_access(uid, wdir, attributes)] \Rightarrow \mathsf{pre}\, cvs\_update$$

cvs_ci_df

$$cvs\_ci\_df == [Cvs\_FileSystem \ ; ProcessState \ ; p? : Path | has\_r\_access(uid, wdir, attributes) \land wdir \in \mathsf{dom}\, wcs\_attributes] \Rightarrow \mathsf{pre}\, cvs\_ci$$

cvs_add_df

$$cvs\_add\_df == [Cvs\_FileSystem \ ; ProcessState \ ; p? : Path | has\_r\_access(uid, wdir \frown p?, attributes) \land cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p? \notin \mathsf{dom}\, files \land has\_w\_access(uid, cvs\_rep \frown (wcs\_attributes\, wdir).rep \frown p?, attributes)] \Rightarrow \mathsf{pre}\, cvs\_add$$

cvs_cd_df

$$cvs\_cd\_df == [Cvs\_FileSystem \ ; cd|] \Rightarrow \mathsf{pre}\, cvs\_cd$$

6.2 Verifying the Refinement

section $Refinement$ **parents** $AbsOperations, CVSServer$

In this section, we focus on the refinement of the data types of our abstract system architecture to data structures of our concrete implementation architecture. In particular, we will map the CVS repository and working copy to a Unix file system and permissions to file attributes.

In order to prove that the concrete data structures correctly implements the abstract data types, we have to define an abstraction schema R which relates the components of the abstract state schemas to the components of the concrete implementation state schemas and thereby explains which concrete states represent which abstract states.

The definition of this relating schema needs auxiliary functions $Rdata$ and $Rname2path$ that map abstract data to concrete data, i.e. files, and the abstract names of files to paths in the Unix file system. Since we are not concerned about the content of the files (except the administration files of CVS), a very rough specification of these functions suffices.

$$
\begin{array}{|l}
Rdata : Abs\_Data \rightarrowtail\!\!\!\!\rightarrow Data \\
Rname2path : Abs\_Name \rightarrowtail\!\!\!\!\rightarrow Path \\
\hline
Rname2path(abs\_cvsauth) = cvs\_rep \frown \langle CVSROOT, cvsauth \rangle
\end{array}
$$

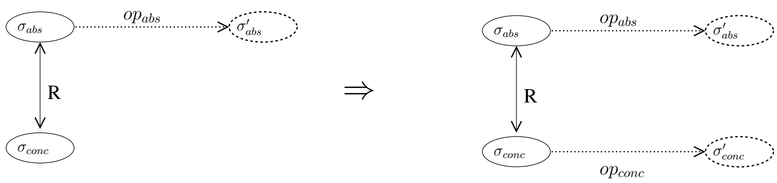
Now we can proceed to define the abstraction schema R. Importing the abstract and concrete state schemas introduces all components that have to be set into relation. Note that some components (cvs_uid or $passwd$ for example) of the abstract state also appear in the concrete state. These components have the same meaning in both states and do not need to be related in this schema. The set $wfiles$ of names of the abstract state, which is used to control the access operations, must be related to the concrete working directory $wdir$. Furthermore, we require that the files in the working copy wc and in the repository rep of the abstract state have corresponding files in the appropriate directories of the concrete file system $files$. We also require that the permission tables in the abstract and in the concrete state are the same and that the roles and passwords that are assigned to each file in the working copy of the abstract state have corresponding attributes in the concrete state.

$$
\begin{array}{l}
\textbf{schema } R \\
ClientState \\
RepositoryState \\
ProcessState \\
Cvs\_FileSystem \\
\hline
abs\_passwd = cvs\_passwd \\
\forall n : wfiles \bullet wdir \ \mathsf{prefix}\ Rname2path(n) \\
Rname2path(\!| \mathsf{dom}\, wc |\!) = \mathsf{dom}\, wcs\_attributes \\
authtab(rep) = get\_auth\_tab(files) \\
\forall n : \mathsf{dom}\, rep \bullet \exists f : \mathsf{dom}\, files \bullet cvs\_rep \ \mathsf{prefix}\ f \\
\qquad \wedge Rname2path(n) = f \wedge Inl(Rdata(rep\, n)) = files(f) \\
\forall n : \mathsf{dom}\, wc \bullet wc\_uidtab(n) = ((wcs\_attributes(Rname2path\, n)).f\_uid) \\
\forall n : \mathsf{dom}\, wc \bullet \exists f : \mathsf{dom}\, files \bullet \neg(cvs\_rep \ \mathsf{prefix}\ f) \\
\forall f : \mathsf{dom}\, rep \bullet \exists ps : \mathbb{P}\, Perm | \{ru, rg\} \subseteq ps \bullet \\
\qquad attributes(Rname2path\, f) = \langle\!| perm == ps, \\
\qquad\qquad\qquad\qquad\qquad\qquad uid == cvsperm2uid(rep\_permtab\, f), \\
\qquad\qquad\qquad\qquad\qquad\qquad gid == cvsperm2gid(rep\_permtab\, f) |\!\rangle
\end{array}
$$

Closely following the refinement notion of Spivey [36], we must prove two refinement conditions for each operation on the abstract state σ_{abs} and its corresponding operation on the concrete state σ_{conc}. However, the original setting assumes that the abstract operation op_{abs} and the corresponding concrete operation op_{conc} are defined over the same input and output parameters, which does not always hold for our operations. Therefore, we extend the original refinement notion by (sometimes) strengthening the premises of the refinement conditions.

Refinement condition (I1) A concrete operation op_{conc} can make a transition whenever its corresponding abstract operation op_{abs} can make a transition (i.e. a successor state σ'_{conc} exists). The situation is depicted in Figure 6.1.

Figure 6.1 Refinement Condition (I1).

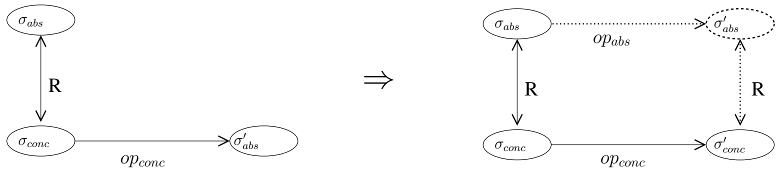


In other words, condition (I1) ensures that a concrete operation terminates whenever its corresponding abstract operation is guaranteed to terminate. For input parameters $in?$, this is formalized as:

$$\forall \sigma_{abs} ; \sigma_{conc} ; in? : T_1 \bullet \mathsf{pre}\, op_{abs} \wedge R \Rightarrow \mathsf{pre}\, op_{conc}$$

Refinement condition (I2) For any possible state σ'_{conc} reachable by the concrete operation, there must be a R-related state σ'_{abs} that can be reached by the abstract operation, provided that the initial states σ_{abs} and σ_{conc} had been R-related. The situation is depicted in Figure 6.2.

Figure 6.2 Refinement Condition (I2).



This condition ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate. For input parameters $in?$ and output parameters $out!$, this is represented formally:

$$\forall \sigma_{abs} \; ; \sigma_{conc} \; ; \sigma'_{conc} \; ; in? : T_1 \; ; out! : T_2 \bullet$$
$$\quad \mathsf{pre}\, op_{abs} \wedge R \wedge op_{conc} \Rightarrow (\exists \sigma'_{abs} \bullet R' \wedge op_{abs})$$

Additionally, there is the requirement that the initial states are compatible:

Refinement condition (II) Each possible initial state of the concrete type must represent a possible initial state of the abstract type.

$$\forall \sigma_{conc} \bullet Init_{conc} \Rightarrow (\exists \sigma_{abs} \bullet Init_{abs} \wedge R)$$

In the following subsections, we will list the proof obligations for the refinement in full detail. The proofs can be found in the in the HOL-Z distribution (see `examples/zeta/cvs-server/holz`). The proofs are highly non-trivial.

6.2.1 The login operation

As a first simple example, we show the two conditions (I1) and (I2) for the CVS login command, which is used to initially authenticate a client to the CVS server. For the definition of the operation schemas, we refer to the *abs_login* schema in the abstract section, and the *cvs_login* schema in the implementation section.

We need the additional preconditions that the requested role and the supplied passwords are the same on the abstract and concrete level. Here, we intentionally did not give the input parameters the same names since this leads to problems in the generated refinement proof-obligations; the standard refinement notion implicitly requires abstract and concrete states to have distinct bindings. Since we cannot use arbitrary formulas in schema calculus expressions, we must wrap a schema *Asm_login* around these preconditions:

$$\begin{array}{|l}
\hline
\textit{Asm_login} \\
\textit{passwd?}, \textit{cvs_pwd?} : \textit{Cvs_Passwd} \\
\textit{uid?}, \textit{cvs_uid?} : \textit{Cvs_Uid} \\
\hline
\textit{passwd?} = \textit{cvs_pwd?} \land \textit{uid?} = \textit{cvs_uid?} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{pre_of_cvs_login} \\
\textit{files} : \textit{FILESYS_TAB} \\
\textit{cvs_uid?} : \textit{Cvs_Uid} \\
\textit{cvs_pwd?} : \textit{Cvs_Passwd} \\
\hline
(\textit{cvs_uid?}, \textit{cvs_pwd?}) \in \mathsf{dom}(\textit{get_auth_tab}\ \textit{files}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{pre_of_abs_login} \\
\textit{uid?} : \textit{Cvs_Uid} \\
\textit{passwd?} : \textit{Cvs_Passwd} \\
\textit{rep} : \textit{ABS_DATATAB} \\
\hline
(\textit{uid?}, \textit{passwd?}) \in \mathsf{dom}(\textit{authtab}\ \textit{rep}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{PreLogin} : \mathbb{N} \\
\hline
\forall \textit{Cvs_FileSystem} ; \textit{ProcessState} ; \textit{Cvs_FileSystem}' ; \textit{ProcessState}' ; \\
\quad \textit{cvs_uid?} : \textit{Cvs_Uid} ; \textit{cvs_pwd?} : \textit{Cvs_Passwd} \bullet \\
\quad\quad \textit{pre_of_cvs_login} \Rightarrow \mathsf{pre}\ \textit{cvs_login} \\
\forall \textit{ClientState} ; \textit{RepositoryState} ; \textit{ClientState}' ; \textit{RepositoryState}' ; \\
\quad \textit{uid?} : \textit{Cvs_Uid} ; \textit{passwd?} : \textit{Cvs_Passwd} \bullet \textit{pre_of_abs_login} \Rightarrow \mathsf{pre}\ \textit{abs_login}
\end{array}$$

$login_1$

$$\begin{array}{l}
login_1 == \forall \textit{ClientState} ; \textit{RepositoryState} ; \textit{ProcessState} \\
; \textit{Cvs_FileSystem} ; \textit{passwd?} : \textit{Cvs_Passwd} ; \textit{cvs_pwd?} \\
: \textit{Cvs_Passwd} ; \textit{uid?} : \textit{Cvs_Uid} ; \textit{cvs_uid?} : \textit{Cvs_Uid} \bullet \\
\textit{Asm_login} \land \mathsf{pre}\ \textit{abs_login} \land R \Rightarrow \mathsf{pre}\ \textit{cvs_login}
\end{array}$$

$login_2$

$$\begin{array}{l}
login_2 == \forall \textit{ClientState} ; \textit{RepositoryState} ; \textit{ProcessState} \\
; \textit{Cvs_FileSystem} ; \textit{ProcessState}' ; \textit{Cvs_FileSystem}' \\
; \textit{passwd?} : \textit{Cvs_Passwd} ; \textit{cvs_pwd?} : \textit{Cvs_Passwd} ; \textit{uid?} \\
: \textit{Cvs_Uid} ; \textit{cvs_uid?} : \textit{Cvs_Uid} ;\bullet\ \textit{Asm_login} \land \mathsf{pre}\ \textit{abs_login} \land R \\
\quad \land \textit{cvs_login} \Rightarrow (\exists \textit{ClientState}' ; \textit{RepositoryState}' \bullet R' \land \textit{abs_login})
\end{array}$$

6.2.2 The add operation

$$\begin{array}{|l}\hline \textit{Asm\_add} \\ p? : Path \\ newfiles? : ABS\_DATATAB \\ wdir : Path \\ \hline \forall n : \mathrm{dom}\, newfiles? \bullet wdir \frown p? \ \mathsf{prefix}\ Rname2path(n) \\ \hline \end{array}$$

add_1

$$\begin{array}{l} add_1 == \forall ClientState ; RepositoryState ; ProcessState \\ ; Cvs\_FileSystem ; newfiles? : ABS\_DATATAB ; p? : Path \bullet \\ Asm\_add \land \mathsf{pre}\, abs\_add \land R \Rightarrow \mathsf{pre}\, cvs\_add \end{array}$$

add_2

$$\begin{array}{l} add_2 == \forall ClientState ; RepositoryState ; ProcessState \\ ; Cvs\_FileSystem ; ProcessState' ; Cvs\_FileSystem' ; newfiles? \\ : ABS\_DATATAB ; p? : Path ;\bullet\ Asm\_add \land \mathsf{pre}\, abs\_add \land R \\ \quad \land\, cvs\_add \Rightarrow (\exists ClientState' ; RepositoryState' \bullet R' \land abs\_add) \end{array}$$

6.2.3 The update operation

$$\begin{array}{|l}\hline \textit{Asm\_update} \\ p? : Path \\ files? : \mathbb{P}\, Abs\_Name \\ wdir : Path \\ \hline \forall n : files? \bullet wdir \frown p? \ \mathsf{prefix}\ Rname2path(n) \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{pre\_of\_cvsUp} \\ p? : Path \\ files : FILESYS\_TAB \\ wcs\_attributes : CVS\_ATTR\_TAB \\ wdir : Path \\ uid : Uid \\ attributes : FILEATTR\_TAB \\ \hline cvs\_rep \frown repOf(wcs\_attributes\ wdir) \frown p? \in \mathrm{dom}\, files \\ has\_w\_access(uid, wdir \frown p?, attributes) \\ \hline \end{array}$$

$$
\begin{array}{|l}
\hline
\quad pre\_of\_cvsUpErr \\
p? : Path \\
wdir : Path \\
uid : Uid \\
attributes : FILEATTR\_TAB \\
\hline
\neg(has\_w\_access(uid, wdir \frown p?, attributes)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\quad pre\_of\_cvsUpNoAct \\
p? : Path \\
files : FILESYS\_TAB \\
wcs\_attributes : CVS\_ATTR\_TAB \\
wdir : Path \\
uid : Uid \\
attributes : FILEATTR\_TAB \\
\hline
cvs\_rep \frown repOf(wcs\_attributes\ wdir) \frown p? \notin \mathrm{dom}\ files \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
PreUpdate : \mathbb{N} \\
\hline
\forall Cvs\_FileSystem ; ProcessState ; Cvs\_FileSystem' ; ProcessState' ; \\
\quad p? : Path ; files : FILESYS\_TAB ; wcs\_attributes : CVS\_ATTR\_TAB ; \\
\quad wdir : Path ; uid : Uid ; attributes : FILEATTR\_TAB ; error! : \mathsf{denotation} \bullet \\
\qquad pre\_of\_cvsUp \Rightarrow \mathsf{pre}\ cvsUp \\
\forall Cvs\_FileSystem ; ProcessState ; Cvs\_FileSystem' ; ProcessState' ; \\
\quad p? : Path ; files : FILESYS\_TAB ; wcs\_attributes : CVS\_ATTR\_TAB ; \\
\quad wdir : Path ; uid : Uid ; attributes : FILEATTR\_TAB ; error! : \mathsf{denotation} \bullet \\
\qquad pre\_of\_cvsUpNoAct \Rightarrow \mathsf{pre}\ cvsUpNoAct \\
\forall Cvs\_FileSystem ; ProcessState ; Cvs\_FileSystem' ; ProcessState' ; \\
\quad p? : Path ; files : FILESYS\_TAB ; wcs\_attributes : CVS\_ATTR\_TAB ; \\
\quad wdir : Path ; uid : Uid ; attributes : FILEATTR\_TAB ; error! : \mathsf{denotation} \bullet \\
\qquad pre\_of\_cvsUpErr \Rightarrow \mathsf{pre}\ cvsUpErr
\end{array}
$$

$update_1$

$$update_1 == \forall ClientState ; RepositoryState ; ProcessState ; Cvs\_FileSystem ; p? : Path ; files? : \mathbb{P}\, Abs\_Name \bullet Asm\_update \land \mathsf{pre}\, abs\_up \land R \Rightarrow \mathsf{pre}\, cvs\_update$$

$update_2$

$$update_2 == \forall ClientState ; RepositoryState ; ProcessState ; Cvs\_FileSystem ; ProcessState' ; Cvs\_FileSystem' ; p? : Path ; files? : \mathbb{P}\, Abs\_Name ;\bullet\ Asm\_update \land \mathsf{pre}\, abs\_up \land R \land cvs\_update \Rightarrow (\exists ClientState' ; RepositoryState' \bullet R' \land abs\_up)$$

6.2.4 The commit operation

The schema *Asm_ci* defines the assumption that the current user has read access to the working directory. This must be stated in a separate assumption because the abstract state has no notion of read/write/execute rights on files.

$$\begin{array}{|l}
\multicolumn{1}{l}{Asm_ci} \\ \hline
uid : Uid \\
wdir : Path \\
attributes : FILEATTR_TAB \\
p? : Path \\
files? : \mathbb{P}\, Abs_Name \\ \hline
has_r_access(uid, wdir \frown p?, attributes) \\
\forall n : files? \bullet wdir \frown p?\ \mathsf{prefix}\ Rname2path(n) \\ \hline
\end{array}$$

With the assumptions in *Asm_ci* we can now formulate the refinement proof obligations for the commit operations:

ci_1

$$ci_1 == \forall ClientState ; RepositoryState ; ProcessState ; Cvs\_FileSystem ; p? : Path ; files? : \mathbb{P}\, Abs\_Name \bullet Asm\_ci \land \mathsf{pre}\, abs\_ci \land R \Rightarrow \mathsf{pre}\, cvs\_ci$$

ci_2

$$ci_2 == \forall ClientState ; RepositoryState ; ProcessState ; Cvs\_FileSystem ; ProcessState' ; Cvs\_FileSystem' ; p? : Path ; files? : \mathbb{P}\, Abs\_Name ;\bullet\ Asm\_ci \land \mathsf{pre}\, abs\_ci \land R \land cvs\_ci \Rightarrow (\exists ClientState' ; RepositoryState' \bullet R' \land abs\_ci)$$

6.2.5 The cd operation

To relate the parameters of the abstract and the concrete *cd* operation, we define the schema *Asm_cd*. We require that all abstract files, which are to be focused ($wfiles?$) must be related to a concrete file within the current working directory *wdir* (appended by some path $p?$) or one of its subdirectories.

Setting the focus to a set of files in the abstract model corresponds to setting the working directory (which also represents a set of files, namely all files within it or within one of its subdirectories).

$$
\begin{array}{|l}
\hline
\textit{Asm\_cd} \\
wfiles? : \mathbb{P}\, Abs\_Name \\
p?, wdir : Path \\
\hline
\forall n : wfiles? \bullet wdir \frown p? \;\mathsf{prefix}\; Rname2path(n) \\
\hline
\end{array}
$$

For this operation it suffices to only take $FileSystem$ into account, not $Cvs\_Filesystem$.

cd_1

$$
\begin{aligned}
&cd_1 == \forall ClientState \,;\, RepositoryState \,;\, ProcessState \\
&;\, FileSystem \,;\, p? : Path \,;\, wfiles? : \mathbb{P}\, Abs\_Name \bullet \\
&Asm\_cd \land \mathsf{pre}\, abs\_cd \land R \Rightarrow \mathsf{pre}\, cd
\end{aligned}
$$

cd_2

$$
\begin{aligned}
&cd_2 == \forall ClientState \,;\, RepositoryState \,;\, ProcessState \\
&;\, FileSystem \,;\, ProcessState' \,;\, FileSystem' \,;\, p? : Path \\
&;\, wfiles? : \mathbb{P}\, Abs\_Name \,;\bullet\, Asm\_cd \land \mathsf{pre}\, abs\_cd \land R \\
&\quad \land cd \Rightarrow (\exists ClientState' \,;\, RepositoryState' \bullet R' \land abs\_cd)
\end{aligned}
$$

7 Formal Analysis

In this chapter, we formally investigate functional and security properties of both abstraction layers of the CVS-Server. By their nature, these are *behavioral properties*, i.e. it is necessary to consider the set of possible *sequences* of operations the system may engage in — the *traces* — and state requirements on the possible states the system reaches after these traces. Hence, the specification of the safety properties in the system architecture and the implementation architecture motivate two Z sections that contain classical *behavioral* specifications.

As functional properties, we describe requirements of what the system should do: Give a client access to the objects in the repository it has permission to. In contrast, security properties in our setting state that the client *must not* access data that he has no permission to, whatever combination of cvs-commands he applies.

Methodically, we need an interface between the data-oriented part of the specification in previous chapters and the behavioral part as presented in the following chapter. The trick is done by converting the operation schemas of both system layers into relations over the underlying state. Traces are represented as transitive closures over the union of these relations; in principle, all sorts of modal operators can be used to specify properties over them.

7.1 Properties of the System Architecture

7.1.1 Security Properties

section *SysArchSec* **parents** *AbsOperations*

We assume that a user "knows" a set of pairs of IDs and passwords, and that the user "invents" only files from a given set of pairs from names to data in the add-operation.

$$
\begin{array}{|l}
Aknows : \mathbb{P}(Cvs\_Uid \times Cvs\_Passwd) \\
Ainvents : \mathbb{P}(Abs\_Name \times Abs\_Data)
\end{array}
$$

In the following, we assume that the IDs and passwords that the client uses to log in are restricted to IDs and passwords the user "knows" and that the name and data that a client adds are restricted to names and data that the user "invents".

$$
\begin{array}{l}
abs\_loginR == abs\_login \land [cvs\_uid? : Cvs\_Uid\ ;\ passwd? : Cvs\_Passwd | \\
\qquad\qquad\qquad\qquad (cvs\_uid?, passwd?) \in Aknows] \\
abs\_addR == abs\_add \land [newfiles? : ABS\_DATATAB | newfiles? \subseteq Ainvents]
\end{array}
$$

The schemas are embedded in a relation, and the "sequence of operations" is captured by a transitive closure over the union of the operation relations.

$$AService == abs\_loginR \lor abs\_addR \lor abs\_ci \lor abs\_up \lor abs\_cd$$
$$AbsState == ClientState \land RepositoryState$$
$$AtransR == \{AService \bullet (\theta AbsState, \theta AbsState')\}^*$$

Now we can postulate the following list of security properties of our architecture:

1. Any sequence of CVS operations starting from an empty working copy does not lead to a working copy with data from the repository that the user has no permission to access in the repository (except if he was able to "invent" it).

$$\begin{array}{l} InitAbsState1 == AbsState \land [wc : ABS\_DATATAB | wc = \emptyset] \\ Areachable1 == AtransR(\!| InitAbsState1 |\!) \\ ASec01 == \forall Areachable1 \bullet AbsState \land [wc : ABS\_DATATAB \ ; \\ \quad rep : ABS\_DATATAB \ ; rep\_permtab : ABS\_PERMTAB | \\ \quad\quad \forall n : \mathsf{dom}\, wc \bullet (n, wc(n)) \in Ainvents \lor \\ \quad\quad\quad ((wc(n) = rep(n)) \land (\exists m : Aknows \bullet \\ \quad\quad\quad\quad (rep\_permtab(n), authtab(rep)(m)) \in cvs\_perm\_order))] \end{array}$$

We could simplify this property and only look at the *cvs_up* operation. This is sensible because this is the only abstract operation that can move data from the repository to the working copy. The property then reads as follows: *After an update, all files in the working copy must have either already been there before the update, or must come from the repository and the user has permission to access them.*

$$\begin{array}{|l} \hline APro p1 \\ wc, wc', rep : ABS\_DATATAB \\ rep\_permtab : ABS\_PERMTAB \\ \hline \forall f : \mathsf{dom}\, wc' \bullet (f, wc'(f)) \notin wc \Rightarrow \\ \quad (f \in \mathsf{dom}\, rep \land wc'(f) = rep(f) \land (\exists m : Aknows \bullet \\ \quad\quad (rep\_permtab(f), authtab(rep)(m)) \in cvs\_perm\_order)) \\ \hline \end{array}$$

$$\begin{array}{l} ASec1 == \forall ClientState \ ; ClientState' \ ; RepositoryState \ ; RepositoryState' \bullet \\ \quad abs\_up \Rightarrow AProp1 \end{array}$$

2. No sequence of CVS operations leads to a repository with "invented" data, except the user "knows" the ID and password that corresponds to the data.

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{--- } AProp2 \text{ ---}}\\
rep : ABS_DATATAB \\
rep_permtab : ABS_PERMTAB \\
\hline
\forall f : \mathsf{dom}\, rep \bullet (f, rep(f)) \in Ainvents \Rightarrow (\exists m : Aknows \bullet \\
\qquad (rep_permtab(f), authtab(rep)(m)) \in cvs_perm_order) \\
\hline
\end{array}$$

$$Areachable2 == AtransR(\!| AbsState |\!)$$
$$ASec2 == \forall Areachable2 \bullet AbsState \wedge AProp2$$

3. Moreover, there is the obvious safety-property that the user can access all data he has the permission to. With access meaning reading (i.e. cvs update) the repository and writing (i.e. cvs commit) the repository, and with ignoring merges, we can formulate this property as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{--- } AlwaysUpdateProp \text{ ---}}\\
wfiles : \mathbb{P}\, Abs_Name \\
wc' : ABS_DATATAB \\
rep : ABS_DATATAB \\
rep_permtab : ABS_PERMTAB \\
abs_uid : Cvs_Uid \\
abs_pwd : Cvs_Passwd \\
\hline
\forall f : wfiles | f \in \mathsf{dom}\, rep \wedge f \in \mathsf{dom}\, wc' \wedge \\
\qquad (rep_permtab(f), authtab(rep)(abs_uid, abs_pwd)) \in cvs_perm_order \bullet \\
\qquad\qquad wc'(f) = rep(f) \\
\hline
\end{array}$$

$$\begin{aligned}
&AlwaysUpdate == \forall ClientState\ ;\ ClientState'\ ;\ RepositoryState\ ; \\
&\qquad RepositoryState' \bullet abs_up \Rightarrow AlwaysUpdateProp
\end{aligned}$$

The commit operation is analogous.

A proof of safety properties can be based on an induction over the transitive closure; in the base-case, a case distinction has to be made: either the user “knows” the authentication for the *cvs_admin* (then he can access everything anyway) or not (then the authentication table remains constant and the user will always have the same access rights.

Due to the refinement, these safety properties carry over to the implementation level. To be more precise, for all compatible states (i.e. the *Cvs_Client* is not *root* has normal access to his home account and his working copy in it), any combination of cvs commands on the implementation level will not violate the security properties. However, this proof breaks down when the cvs commands can be interleaved with Unix-commands. For this purpose, an own induction over refined states has to be done, which is subject of the next section.

7.2 Properties of the Implementation Architecture

With respect to "knows" and "invents", we make the same assumptions as in section 7.1.1. Moreover, we have to assume specialties due to the implementation architecture level: for instance, the user must have write access to his own working copy, the working directory must point into the working copy, etc.

7.2.1 Security Properties

section $ImplArchSec$ **parents** $CVSServer$

We approach the formalization of the security properties in the same way as for the abstract case.

$$\begin{array}{|l} Cknows : \mathbb{P}(Cvs\_Uid \times Cvs\_Passwd) \\ Cinvents : FILESYS\_TAB \end{array}$$

$$\begin{array}{l} CService == cvs\_cd \lor cvs\_mkdir \lor cvs\_mkfile \lor cvs\_access \lor cvs\_write \lor cvs\_rm \\ \qquad \lor cvs\_rmdir \lor cvs\_mv \lor cvs\_chown \lor cvs\_chmod \lor cvs\_setumask \\ \qquad \lor cvs\_login \lor cvs\_co \lor cvs\_update \lor cvs\_ci \lor cvs\_add \\ ConcState == ProcessState \land Cvs\_FileSystem \\ CtransR == \{CService \bullet (\theta ConcState, \theta ConcState')\}^* \end{array}$$

Then we postulate the following list of security properties of our implementation architecture:

1. No sequence of CVS operations *and* Unix-operations starting from an empty working copy leads to a working copy with data from the repository that the user has no permission to access in the repository (except if he was able to "invent" it).

$$\begin{array}{l} InitConcState1 == ConcState \land [wcs\_attributes : CVS\_ATTR\_TAB | \\ \qquad wcs\_attributes = \emptyset] \\ Creachable1 == CtransR(\!| InitConcState1 |\!) \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CProp1} \\ \hline wcs\_attributes : CVS\_ATTR\_TAB \\ files : FILESYS\_TAB \\ attributes : FILEATTR\_TAB \\ \hline \forall n : \mathrm{dom}\, wcs\_attributes \bullet (n, files(n)) \in Cinvents \lor \\ \qquad ((files(n) = files((wcs\_attributes\, n).rep)) \land (\exists m : Cknows \bullet \\ \qquad\qquad has\_r\_access(cvsperm2uid(get\_auth\_tab(files)(m)), \\ \qquad\qquad\qquad cvs\_rep \frown (wcs\_attributes\, n).rep, attributes))) \\ \hline \end{array}$$

$CSec01 == \forall Creachable1 \bullet ConcState \land CProp1$

2. Any sequence of CVS-operations *and* Unix-operations does not lead to a repository with "invented" data, except the user "knows" the id and password that corresponds to it.

$Creachable2 == CtransR(\lparen ConcState \rparen)$

$$\begin{array}{|l}
\multicolumn{1}{l}{CProp2} \\
\hline
wcs_attributes : CVS_ATTR_TAB \\
files : FILESYS_TAB \\
attributes : FILEATTR_TAB \\
\hline
\forall n : \mathsf{dom}\, files | cvs_rep \;\mathsf{prefix}\; n \bullet (n, files(n)) \in Cinvents \Rightarrow \\
\quad (\exists m : Cknows \bullet has_w_access(cvsperm2uid(get_auth_tab(files)(m)), \\
\qquad cvs_rep \frown (wcs_attributes\, n).rep, attributes)) \\
\hline
\end{array}$$

$CSec2 == \forall Creachable2 \bullet ConcState \land CProp2$

3. No user can directly access or manipulate (chmod) the repository, except he is the system administrator "root" (this is essentially achieved by mapping CVS user IDs to Unix user IDs and group IDs that are disjoint from all user IDs of "real" users).

   Here, we have the choice to formulate this property as a general state invariant over the system state or over all possible traces of Unix-operations.

   Formulating the property as a general state invariant:

$$\begin{array}{|l}
\multicolumn{1}{l}{CProp3} \\
\hline
uid : Uid \\
cvs_rep : Path \\
attributes : FILEATTR_TAB \\
\hline
uid \notin \{cvsperm2uid(cvs_admin), root\} \\
\neg has_w_access(uid, cvs_rep, attributes) \\
\hline
\end{array}$$

$CSec3 == \forall ConcState \bullet ConcState \land CProp3$

4. The CVS administrator can withdraw the permissions for a user (after a suitable update of the authentication table); i.e. after a withdraw, the user does not get data into his working copy except by "inventing" it (e.g. the user cannot get data from the repository into his working copy).

We define a relation $CWithdrawId$ that defines the withdrawal of a user id and a relation $ChangeWC$ which defines the possible actions of a user to alter his working copy.

$$
\begin{array}{|l}
CWithdrawId : ConcState \leftrightarrow ConcState \\
ChangeWC : ConcState \leftrightarrow ConcState \\
\hline
\forall a, b : ConcState \bullet (a, b) \in CWithdrawId \iff \\
\quad b.files = a.files \oplus \{cvs\_rep \frown \langle CVSROOT, cvsauth \rangle \mapsto \\
\qquad Inl(data\_of(a.cvs\_passwd \lhd get\_auth\_tab(a.files)))\} \\
\forall a, b : ConcState \bullet (a, b) \in ChangeWC \iff (\forall n : \mathsf{dom}\, b.wcs\_attributes \bullet \\
\quad (n, b.files(n)) \notin Cinvents \\
\qquad \land (n \notin \mathsf{dom}\, a.wcs\_attributes \lor a.files(n) \neq b.files(n)))
\end{array}
$$

$$
\begin{array}{l}
CSec4 == [a, b : ConcState | (a, b) \in CtransR \land (a, b) \in CWithdrawId] \\
\quad \Rightarrow [b, c : ConcState | (b, c) \in CtransR \land (b, c) \notin ChangeWC]
\end{array}
$$

Proofs have the same structure as in section 7.1.1, but are far more complex since concepts such as paths, the distinction between files and directories, and their permissions (in particular when being created) are involved.

7.3 Setting both Security Models into Perspective

section $CombinedSec$ **parents** $SysArchSec, ImplArchSec$

This section serves to discuss the relation between the system level security and the implementation level security.

Some security properties cannot be expressed on both abstraction levels. For example, property 4 of the implementation model cannot be expressed on the abstract level.

On the implementation level, we could require the following property: No user can access the working copy of other users — except if the others set their umask too liberally or the user is the sysadmin “root”. This property cannot be expressed in our single user model!

The following Z-section is only a virtual section that depends on all other sections, and is therefore used to load the complete specification:
section all **parents** $Refinement, CombinedSec, SysArchConsistency, ImplArchConsistency$

8 Conclusion

We presented a case-study where an access control security problem for a realistic system has been made amenable to formal, machine-based analysis. As a hidden theme, we also presented a *method* for analyzing security in off-the-shelve operating system environments: First, specify the security architecture (as a framework for formal security properties), second, specify the implementation architecture (validated by inspecting informal specifications or code and by testing), third, set up the security technology mapping as a refinement problem, forth, establish a formal security analysis over the transitive closure over the system transition relation, and fifth, prove refinements and security properties by mechanized, gap-less proofs. This method is applicable for a wider range of security problems and may be relevant, e.g., in mission critical e-commerce applications or in e-government applications, where high-level certifications are mandatory.

More precisely, we have mapped an abstract model of the CVS-Server (with the access-control model RBAC inside) to a more concrete implementation model based on the concrete security mechanisms as offered by the traditional UNIX/POSIX security architecture. Moreover, we also provided an implementation in form of a configuration of the standard off-the-shelf implementation of CVS [8]. Thus, we showed an application of formal methods for the field of security implementations, both from the technical and the conceptual side. In the sequel, we will discuss these aspects in more detail.

8.1 The Technical Side: A Secure CVS Configuration

Our formal model (and its analysis) built the foundation of "our" CVS server setup in our group at the University of Freiburg. At present, it is applied for a repository containing 2.5 Gigabyte of data for over 80 users categorized in five roles.

The presented setup addresses our main requirements: it does not monopolize an additional machine with the CVS service, which would result in special administration and installation of it, and it enforces a highly desirable access control model. Moreover, any machine in our network can be assigned to our CVS service, without any loss in security. Our CVS-server implementation requires only the standard access control policy provided by the Unix standard, i.e. there is no need for e.g. access control lists (ACL) as provided by some Unix variants. Further, our implementation guarantees the encrypted transmission of the passwords and the versioned data (not discussed in this report).

The main disadvantage for the users is the need for a special (patched) CVS client, which also leads to some additional administrative work, because we have to distribute these clients in source code and as binary for different operation systems. Further, the security policy is 'hard-wired' into our setup, e.g. support for write exclusion within one CVS role is realized via a different mechanism.

8.1.1 Related Work

Other approaches for securing the standard CVS client/server model (using the `pserver`-protocol) can roughly be divided into four classes:

1. Securing authentication and network traffic by providing SSL support, e.g. via tunneling [5] or wrapping the standard CVS server [39],
2. Protecting the local network by running the standard CVS server in a sandbox (chrooted) environment [17, 18],
3. Re-engineering the implementation of the `pserver`-protocol [1],
4. Setting up large scale closed servers providing CVS functionality, together with a project administration [11, 28].

These solutions mainly attempt to fix the known problems with the standard CVS implementation by either providing encryption or minimizing the harms of a intruder. Whereas this is clearly one part of our problem, we also wanted to provide an access control model, which lead to our choice of [39] as basis for our work, which solves the vital problem of authentication and encryption of data transmission. Our DAC based RBAC setup is an add-on on top of that, such that our setup can be used together with most of the other approaches listed above.

8.1.2 Future Work

In our opinion, the greatest improvement of our CVS-server setup would be the support of more flexible access control policies. Our implementation supports already write exclusion within a CVS permission via a script[1] implementing ACLs (access control lists). Whereas such a write exclusion support clearly opens the door for generic access policies its success is clearly limited by the lack of generic hooks for implementing security checks (see for example [2] for such an interface). Therefore any effort implementing a generic access control into CVS (without rewriting the CVS system) would either lead to a complex and intransparent configuration or it would heavily depend on the underlying operation system (e.g. ACLs of the underlying filesystem), or both.

Taking all the well known hassles (e.g. moving of files, ...) of CVS into account, we see a great future for projects creating a *successor* of CVS. Candidates for a replacement — we only mention Subversion [4] and OpenCM [3] here — are taking long strides toward production use. Subversion can be considered as the natural choice, since it aims to solve the problems of CVS on the version control side and supports a fine grain access control model using WebDAV [] in the future. On the other side, OpenCM was designed with a refined access control mechanism beyond RBAC from the beginning, but is weaker as a version control system.

Concluding, when setting up a *new* version control system in the near future, we would probably use one the CVS successors. We would favor Subversion over OpenCM because of its better support in the free software community and our relatively modest demands with respect to access control.

[1] This script, called `cvs_acls`, is part of the standard CVS distribution and can be found in the `contrib` directory.

8.2 The Conceptual Side: Formal Methods for Security Technology

It has been widely recognized that security properties can not be easily refined — actually, finding refinement notions that preserve security properties are a hot research topic. However, standard refinement proof technology has still its value here since it checks that abstract security requirements (which can be seen as *security against unintentional misuse*) are indeed achieved by a mapping to concrete security technology, and that implicit assumptions on this implementation have been made explicit. With respect to security against *intentional exploits of security leaks*, we believe that specialized refinement notions will be limited to restricted aspects of a system. For this problem, in most cases the answer will be to analyze the security on the implementation level, possibly by reusing results from the abstract level. From the methodological viewpoint this simply means that for an analysis of *intentional exploits of security leaks* implementation architectures must simply be taken more seriously, which implies that more detailed and implementation oriented models deserve more attention as before, where more abstract models have been preferred. But in security analysis, more abstract models are not necessarily better ones.

It is characteristic for our approach to consider security properties like access control as ordinary functional properties; as a consequence, the classical distinction between security and safety is very often difficult to establish within our model. When refining a security architecture down to an implementation architecture, it is merely tradition to consider one form of exceptional behavior (e.g. precondition that *has_w_access* holds for a user and a file) as security and another form (e.g. a path has the right format and enables indeed to access a file in the filesystem) as safety.

8.2.1 Related Work

1. The common reference for stating that security properties (i.e. information flow) are incompatible with refinement is [20], where also a first method for stepwise development is proposed. More recent work for information flow is [25] (where also a nice overview can be found). Other recent work on security-preserving refinements are [21] (secrecy) and [37] (confidentiality). To all these approaches, the critique above applies: these notion are focused on one particular security notion (so the question on combination arises), the transition to "realistic" implementations is not handled, such that these techniques remain limited to special aspects of a system.

2. Sandhu already described in [32] a method for embedding Role-Based Access Control with the Discretionary Access Control provided by standard Unix systems. Our implementation used this construction for providing the static role.

3. To our knowledge, there is amazingly few work on formalizing the UNIX/POSIX filesystem in the literature. Some approaches focus on the functional aspects [15, 27]. The model in the Isabelle/HOL distribution by Markus Wenzel copes with security aspects and has inevitably many similarities with ours; however, it does not handle the execution flag, groups, not to mention the set-bits which play an important role in the role propagation when creating subdirectories.

8.2.2 Future Work

The formal proofs we did so far represent in our opinion a “proof of technology” for this type of reasoning, but we do not claim that they represent a thorough and complete analysis of the (real) CVS-server. So far, most consistency, but only selected refinement and security properties have been proven.

However, from our experience gained by our proofs, we believe that specialized proof technologies for certain types of proofs (refinement, security-properties) in our methodological frame and concrete proof-technology can be improved and will lead — together with more and extended standard models for operating system interfaces — will dramatically increase the effectivity and feasibility of our approach.

Appendix

A An Example CVS server setup

In this chapter, we give a short description of "our" CVS server setup (see Fig. A.1 for an overview) for the software engineering group at the University Freiburg. We only describe the specific setup of the used tools, for a comprehensive understanding one should also read the manuals [39, 7, 26, 12] of the used tools.

A.1 Motivation

A.1.1 Our Requirements

We needed a reliable CVS-server implementation, that fulfills our main needs, concerning

ease of administration: Administration of the CVS-server should not rely on system administrator (`root`) privileges. Ideally, it should be possible to administrate (add new users, revoke user rights, change module permissions, ...) the CVS-server as a "normal" Unix user.

system security: The CVS-Server should not introduce new security problems into the existing system configuration. Especially passwords should not be transmitted unencrypted.

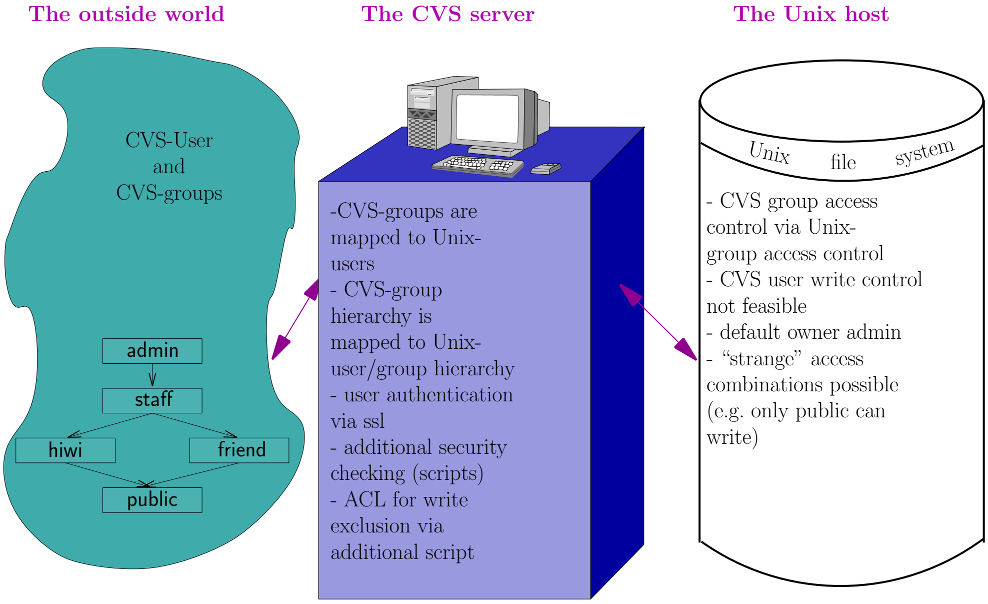
A.1.2 Existing Solutions

Other approaches for securing the standard CVS client/server model (using the `pserver`-protocol) can roughly be divided into four classes:

1. Securing authentication and network traffic by providing SSL support, e.g. via tunneling [5] or wrapping the standard CVS server [39],
2. Protecting the local network by running the standard CVS server in a sandbox (chrooted) environment [17, 18],
3. Re-engineering the implementation of the `pserver`-protocol [1],
4. Setting up large scale closed servers providing CVS functionality, together with a project administration [11, 28].

These solutions mainly attempt to fix the known problems with the standard CVS implementation by either providing encryption or minimizing the harms of a intruder. Whereas this is clearly one part of our problem, we also wanted to provide an access control model, which lead to our choice of [39] as basis for our work, which solves the vital problem of authentication and encryption of data transmission. Our DAC based RBAC setup is an add-on on top of that, such that our setup can be used together with most of the other approaches listed above.

Figure A.1 The general idea/overview for the mapping.



A.2 The System Configuration

A.2.1 The Unix Groups

Our scenario requires five different CVS roles with the group hierarchy shown in Fig.A.2:

admin: The CVS administrator.

staff: The majority of users works within this role, it is suited for members of the software engineering group Freiburg.

friend: This role is suited for joint work with other research groups. It has a lower priority than the group `staff`.

hiwi: This role is suited for student assistants which are supervised by a member of the software engineering group. It also has a lower priority than the group `staff` and is placed on the same level as the group `friend`.

public: This is the lowest group in our hierarchy, suited for public access.

For mapping these roles (and implementing a hierarchic RBAC on top of them) onto the Unix DAC, we need five users and five groups on the Unix host, where the group membership relation of the users is shown in Tab. A.1 on the facing page (see Tab. A.2 on the next page for the corresponding entries in the Unix system `/etc/groups` file). These implementation of an role-based access control model on top of a filesystem providing only DAC follows the ideas described in [32].

Figure A.2 Our example CVS roles hierarchy

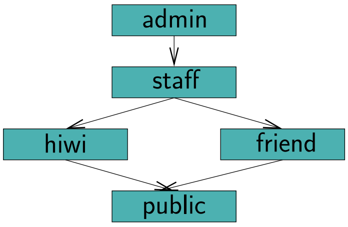


Table A.1 The Unix group hierarchy (membership relation)

user	**groups the user is member of**
admin	admin, staff, hiwi, friend, public
staff	staff, hiwi, friend, public
hiwi	hiwi, public
friend	friend, public
public	public

Table A.2 Our `/etc/groups` file

```
admin: admin
staff: staff, admin
hiwi: hiwi, staff, admin
friend: friend, staff, admin
public: public, friend, hiwi, staff, admin
```

A.2.2 The Inetd Configuration

In our environment, the `cvsauth` daemon is started via `inetd` on a server machine running Solaris 8. In this case

- the file `/etc/inetd.conf` should contain (in one line!) this line:

```
cvsauth stream  tcp    nowait  root
      /export/local/unix/cvs/CVS-1.11.auth/sbin/SunOS-sparc/cvsauth cvsauth -l
```

- the file `/etc/services` should contain these lines:

```
cvsauth        2405/tcp        # CVS-Authentication-Service
cvsauth        2405/udp        # CVS-Authentication-Service
```

A.3 The cvsauth Configuration

For the support of our RBAC model we do a little trick in the cvsauth configuration. For every Unix-user (of our CVS group hierarchy) we provide a 'cvsauth repository configuration' in the `cvsauth` configuration (see Tab. A.3). They only differ in the `WriterUID` and the list of (external) CVS users (with password). Note, that all configuration section share the same cvs repository root, namely `/export/deposit/repository`.

A.4 The Sudo Configuration

Unfortunately, changing access rights for files or directories in the repository is not possible via cvs itself. One has do this via the normal Unix commands `chown` and `chgrp`. For simplifying the administrative work, we use a Sudo [26] setup (see Tab. A.4 for an excerpt of the used `/etc/sudoers`) allowing the cvs-administrators (in our example, these are the Unix users `dagobert`, `donald`, and `daisy`) the rescaling of access rights.

A.5 Example Administrative Usage

A.5.1 Creating new users

Adding a new cvs account requires mainly two steps, generating a hash value of the password and adding the new account information to the cvsauth configuration. Assume, we want to add the user `joe` (in the group `softechhiwi`) with password `joespwd`:

1. First, we generate the desired hash of the password:

```
tcsh% ./cvsauth -e joespwd
encrypt password:joespwd
result is START --->UQ06r7PMnXm5A<--- END
```

2. Now we add Joes information in the `softechhiwi` section of the cvsauth configuration:

Table A.3 A sample cvsauth-config

```
[softechadm]
Path=/export/deposit/repository
WriterUID=cvsadmin
User=daisyadm:sorq05l7SIzg2:W
User=donaldadm:SMppyM3hspXII:W

[softechstaff]
Path=/export/deposit/repository
WriterUID=cvssstaff
User=bert:mUO7E/IWhi17o:W
User=ernie:3fCoyjYrgCPUk:W

[softechhiwi]
Path=/export/deposit/repository
WriterUID=cvssthiwis
User=trick:USVUD73ucjO2U:W
User=tick:oRYdLGfNJ5e2s:W

[softechfriends]
Path=/export/deposit/repository
WriterUID=cvsstfriends
User=track:5uAho6c6qjOlk:W
User=gustav:yFh34gaCS82TM:W
```

Table A.4 Sample Sudo configuration

```
User_Alias CVS=dagobert,donald,daisy

Cmnd_Alias CVSFILE1=/usr/bin/chown cvs* /export/deposit/repository/*
Cmnd_Alias CVSFILE3=/usr/bin/chown -?* cvs* /export/deposit/repository/*
Cmnd_Alias CVSFILE2=/usr/bin/chgrp cvs* /export/deposit/repository/*
Cmnd_Alias CVSFILE4=/usr/bin/chgrp -?* cvs* /export/deposit/repository/*


Cmnd_Alias SHELL1=/usr/bin/sh *,/usr/local/bin/bash *,/usr/bin/bash *
Cmnd_Alias SHELL2=/usr/local/bin/tcsh *,/usr/bin/tcsh *

CVS rivejern=(cvsadmin) SHELL1,(cvsadmin) SHELL2
CVS rivejern=(root) CVSFILE1, (root) CVSFILE2, (root) CVSFILE3, (root) CVSFILE4

SOFTECH  rivejern=(cvssstaff) SHELL1,(cvssstaff) SHELL2
```

```
[softechhiwi]
Path=/export/deposit/repository
WriterUID=cvssthiwis
User=trick:USVUD73ucjO2U:W
User=tick:oRYdLGfNJ5e2s:W
User=joe:UQ06r7PMnXm5A:W
```

A.5.2 Changing Access Rights

Assume we have the directory `src/sable` in our repository and we would like, that all members of group `softechhiwi` can access this directory. Therefore this directory must be owned by the Unix group `cvssthiwis` which also must have write, read, and execute rights (the Unix User can also be `cvssthiwis`, or any 'higher' element of our hierarchy). We can achieve this by issuing the following commands (based on our sudo configuration):

```
sudo -u root chown -R cvsadmin:cvssthiwis /export/deposit/repository/src/sable
```

In a similar way, one would assure that the `gw`, `gr`, `gx` bits of the directory are set.

A.6 Extensions

For a more fine-granular write exclusion, we use the `cvs_acls.pl` from David G. Grubbs script, which is implemented as commitinfo-hook. This script is part of the standard CVS-distribution (in the `contrib` directory). This is slightly orthogonal to our remaining architecture. We tolerate this in our situation, because our needs are mostly fulfilled with role based security. Thus we rarely need write-exclusion within a group (which in any case, extends classical RBAC)

A.7 Our Experiences

The presented formalization of a CVS security architecture was developed in parallel to the underlying implementation, which serves approximately 2.5 Gigabyte of data for over 80 users categorized in five roles. We are using this setup since over a year on a daily basis, without problems.

Bibliography

[1] cvs-nserver, 2002. `http://cvs-nserver.sourceforge.net/`.

[2] Linux security module, 2002. `http://lsm.immunix.org/`.

[3] OpenCM, 2002. `http://www.opencm.org`.

[4] Subversion, 2002. `http://subversion.tigris.org`.

[5] Anton Berezin. Chrooted tunneled read–write CVS server: How-To, June 2001. `http://www.prima.eu.org/tobez/cvs-howto.html`.

[6] A.-P. Bröhl and W. Dröschel (Hrsg.). *Das V-Modell — Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Software — Anwendungssysteme — Informationssysteme. Oldenbourg, München, second edition, 1995.

[7] Per Cederqvist et al. *Version Management with CVS*, 2000. `http://cvshome.org/docs/`.

[8] Concurrent version system, April 2001. `http://www.cvshome.org`.

[9] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999. `ftp://ftp.internic.net/rfc/rfc2246.txt`. Status: proposed standard, supersedes SSL version 3.0 available at `http://www.netscape.com/eng/ssl3/`.

[10] Karl Franz Fogel. *Open source development with CVS*. The Coriolis Group, 14455 N. Hayden Road, Suite 220 Scottsdale, AZ 85260, USA, 1999. ISBN 1-576-10490-7. `http://cvsbook.red-bean.com`.

[11] Savannah, June 2001. `savannah.gnu.org`.

[12] Æleen Frisch. *Essential System Administration*. A Nutshell handbook. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, second edition, December 1995. ISBN 1-56592-127-5. `http://www.oreilly.com/catalog/esa2`.

[13] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, Singapore, 1993.

[14] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993, 472 pages.

[15] Maritta Heisel. Specification of the Unix file system: A comparative case study. *Lecture Notes in Computer Science*, 936:475–488, 1995. ISSN 0302-9743. `http://ivs.cs.uni-magdeburg.de/~heisel/publications.html`.

[16] Martin C. Henson and Steve Reeves. A logic for the schema calculus. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 172–191. Springer-Verlag, 1998.

[17] Joey Hess. Anonymous CVS access via ssh, June 2001. `http://www.kitenet.net/programs/sshcvs/`.

[18] Chrooted SSH CVS server How-to, June 2001. `http://www.idealx.org/prj/idx-chrooted-ssh-cvs/dist/chrooted-ssh-cvs-server.html`.

[19] Z formal specification notation — syntax, type system and semantics. 2002. ISO/IEC 13568:2002, International Standard.

[20] Jeremy Jacob. On the derivation of secure components. In *IEEE Symposion on Security and Privacy*, pages 242–247. Oakland, CA., 1989.

[21] Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (FME)*, volume LNCS 2021. 12-16 March 2001.

[22] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of TPHOLs'96*, LNCS 1125, pages 283–298. Springer, Berlin, 1996.

[23] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

[24] Philippe Kruchten. *The Rational Unified Process, An Introduction.* The Addison-Wesley Object Technology Series. Addison-Wesley, December 1998. ISBN 0201604590.

[25] Heiko Mantel. Preserving Information Flow Properties under Refinement. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 78–91. IEEE Computer Society, Oakland, CA, USA, May 14–16 2001.

[26] Todd Miller. Sudo, July 2002. `http://www.courtesan.com/sudo/`.

[27] Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*, SE-10(2):128–142, March 1984.

[28] Sourceforge, June 2001. `http://www.sourceforge.net`.

[29] The single UNIX specification version 3, 2002. `http://www.opengroup.org/onlinepubs/007904975/toc.htm`. This standard superseeds the "Single UNIX Specification Version 2" (Unix 98) and the "IEEE Std 1003.1-2001" (POSIX.1).

[30] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828. Springer-Verlag Inc., New York, NY, USA, 1994. ISBN 3-540-58244-4, 0-387-58244-4, xvii + 321 pages.

[31] A.W. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, Englewood Cliffs, NJ, 1998.

[32] R. Sandhu and G-J. Ahn. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 486–502. 1998. `http://ite.gmu.edu/list/confrnc/nissc/pdf_ver/n98unix.pdf`.

[33] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996. ISSN 0018-9162. `http://ite.gmu.edu/list/journals/computer/pdf_ver/i94rbac(org).pdf`.

[34] T. Santen. On the semantic relation of Z and HOL. *Lecture Notes in Computer Science*, 1493:96–115, 1998. ISSN 0302-9743. `http://swt.cs.tu-berlin.de/~santen/pub/zum98.html`.

[35] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[36] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, second edition, 1992. ISBN 013-978529-9. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.

[37] Andreas Pfitzmann Thomas Santen, Maritta Heisel. Confidentiality-preserving refinement is compositional - sometimes. In *Proc. 7th European Symposium on Research in Computer Security (ESORICS)*, volume LNCS 2502, pages 194–211. 2002.

[38] The Single UNIX Specification, Version 2: Unix 98, 1997. `http://www.unix-systems.org/`.

[39] Martin Vogt. cvsauth, April 2001. `http://cvsauth.sourceforge.net`.

[40] Jim Woodock and Jim Davies. *Using Z*. Prentice Hall, 1996. ISBN 0-13-948472-8. `http://softeng.comlab.ox.ac.uk/usingz/`.

[41] The ZETA system. `http://uebb.cs.tu-berlin.de/zeta/`.

Index