Technical Report of the Virtual Unix Lab Project

Implementing exercise result verification for the Virtual Unix Lab

Hubert Feyrer <hubert@feyrer.de>

Computer Science Department of the University of Applied Sciences Regensburg and

Information Science Department of the University of Regensburg

December 4, 2007

Abstract

This paper discusses implementation of exercise result verification in the Virtual Unix Lab. In the first step, instructions and checks to verify the results are not coupled, which is changed in the second step. For each step, an overview of the underlying system design is given, followed by implementation details of the system as it was realized. For reference, exercise texts, details of the processor of the resulting domain specific language, check scripts, as well as details of the underlying database structure used in the Virtual Unix Lab are included.

Keywords: Applications in subject areas, Virtual Unix Lab, vulab, Verification Unit Domain Specific Language, VUDSL, system administration, implementation, Regensburg

Contents

1 Introduction

Besides setting machines up and making them accessible for exercises, a major task of the Virtual Unix Lab is verifying and evaluating the results of the exercises at the end. This paper discusses implementation details of the verification of results of exercises that is performed in the Virtual Unix Lab by using a Domain Specific Language (DSL).

Basics on DSLs are assumed[1], and while a coarse overview of the system's design is given here, this is covered in more detail elsewhere[2]. The focus here is to illustrate details of the implementation.

2 Step I: Instructions and Checks not coupled

This section describes the first design and implementation step of the result verification architecture implemented in the Virtual Unix Lab. A description of the design with the key components and how they are integrated is given, followed by selected implementation details which lead to a discussion of possible improvements for step II.

2.1 Design and interaction overview

The Virtual Unix Lab result verification architecture consists of a number of key components that reflect the exercise and which interact in certain ways. This section introduces these components and their interaction, more details can be found in [Feyrer, 2008].

The components of step I of the Virtual Unix Lab exercise result verification architecture are:

Exercise text: In the first version of the Virtual Unix Lab result verification architecture, the exercise text was stored in plain HTML text, and displayed at exercise time. The HTML text was embedded into a larger document that gave the usual web layout, a display of the time remaining and a button to indicate that all tasks were completed and the exercise was finished early.

See figure 5 for an example of the plain ASCII and rendered exercise text, and appendix A for texts of the exercises presented to the student.

[1] [Spinellis, 2001]
[2] [Feyrer, 2008]

Check-scripts: Run either on of the lab machines or an "outside" machine to check if an aspect of the exercise was performed successfully or not. The whole exercise consists of a number of checks to verify various aspects.

Following Cocke and Schwartz' "stereotype" paradigm, check scripts are abstractions to map complex verification operations expressed in an arbitrary language (usually as a perl or shell script) into abstract primitives that perform their pre-defined task, and report success or failure upon completion[1].

Database with web-interface to define checks: Check scripts are stored on the Virtual Unix Lab master machine, which installs the lab machines, runs the check scripts for result verification and also runs the web frontend for course management. All data on exercises is stored in various tables of a database, and the `uebungs_checks`" table describes the connection between an exercise and a check, using the following information (see also appendix G.6):

- A unique identifier for the exercise, e.g. "nis", "nfs", ...
- Filename of the check-script, e.g. "check-domainname-set")
- Which machine to run the check-script on, e.g. "vulab1", "vulab2" or "localhost" for the Virtual Unix Lab master machine.
- A description of what the check-script does, to be printed when giving the user feedback about the exercise's result, e.g. "Was domainname(1) set properly?"

Besides the data on which check to run (and on what lab machine), there was another table ("`'ergebnis_checks'`) in the database that describes the checks' results. Basically the check associated with the exercise and a boolean "success" value is stored for evaluation and feedback purpose.

Sections 2.1 and appendix A.1 as well as the "preparation" shown in figure 1 and in the "verification" phases shown in figure 3 contain more details on the database and the web frontend.

Result verification engine: This is the part where all the components are tied together – exercises, their checks as stored in the database, evaluation of the checks and storing the results. All this is done by the script `uebung_auswerten` which is described in more detail in the "verification" part of section 2.1.

Integration of those components is shown in figures 1, 2, and 3. The following steps in the exercise life cycle exist, which use the named components:

Preparation: Preparation of an exercise in the first implementation consisted of several parts:

- Define the general parameters as of the exercise using the web interface shown in figure 6.

[1] [Cocke and Schwartz, 1970] pp. 6

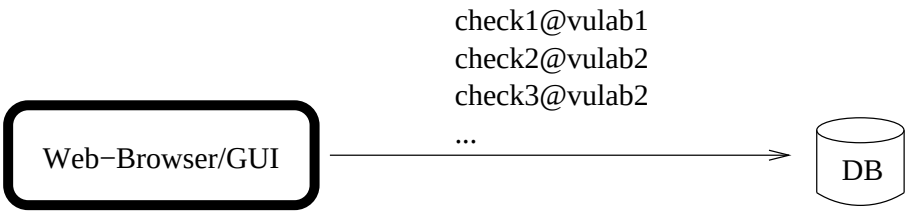


Figure 1: Step I: Preparation

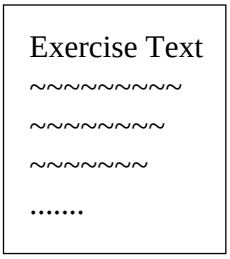


Figure 2: Step I: Exercise

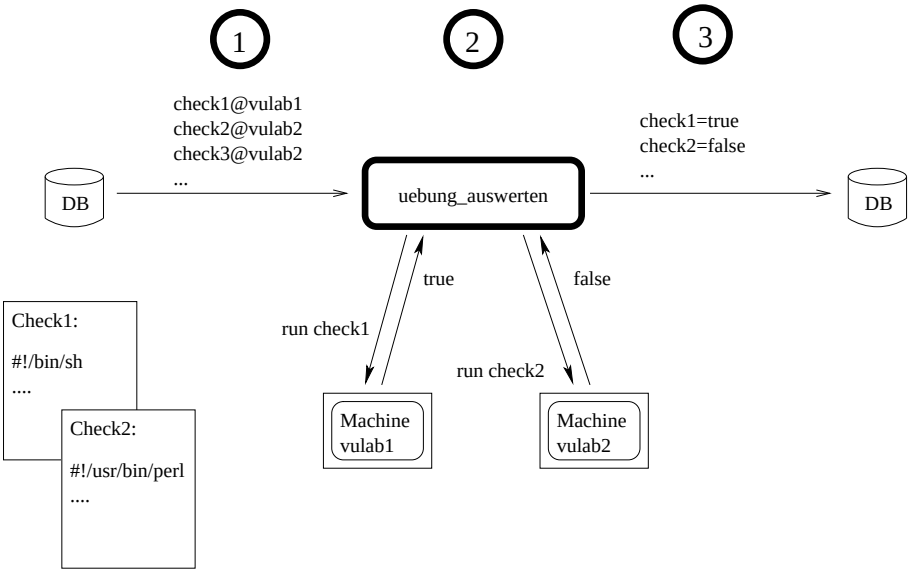


Figure 3: Step I: Verification

- Write the exercise text as a HTML file.
- Determining which checks are needed to verify if the exercise was completed successfully.
- Associate the checks with the exercise in the database, using the web frontend.
  Although data entry via the web frontend was not very difficult, using the web frontend to enter 30-40 checks was tedious. This point was addressed in the second implementation, see 3.1.

Exercise: After the exercise was written and stored on the Virtual Unix Lab master machine, nothing needed to be done at runtime. The PHP script that displayed the exercise text read the exercise text file, added HTML header and footer, and displayed it to the client.

Verification: When the exercise time is over, either because the user clicked on the "Done!"-button or because time ran out, the verification process is started. The three steps involved in the verification process are illustrated in figures 1, 2, and 3:

1. Determine which checks to run on which lab machine.
2. Get the lab machine to run the check script, and collect the result.
3. Store the check's result in the database.

Analysis: No analysis of results stored in the database was realized for the first implementation of the Virtual Unix Lab, neither for users to query their individual results nor for teachers to get an overview of overall performance for each test.

2.2 Implementation details

After describing the design of the Virtual Unix Lab result verification architecture in the previous section, this section lists some of details of the first implementation.

Database & tables: The database holding all information is probably the first detail to discuss, as it is involved in all interactions of the Virtual Unix Lab.

The PostgreSQL[1] database system was chosen as the database engine as it worked on the main machine running NetBSD/sparc without problems, in contrast to MySQL, which did not even compile. The database contained a number of tables that were used to tie things together[2]. Notable tables are:

buchungen: This table contains all the exercises to prepare, including date and time, the user the exercise is prepared for, and which exercise.

[1] [Momjian, 2000]
[2] [Zimmermann, 2003] pp. 113

See G.7.1 for the definition of this table and G.7.2 for a list of all booked exercises at a certain point of time.

uebungen: This table contains all the data on an exercise - how much preparation time the exercise needs, duration, and the filename of the exercise text. See G.4.1 for the definition of this table, a list of all available exercises is available in G.4.2.

uebungs_checks: This table lists the checks that need to be run for a certain exercise, including the exercise id as described in the `uebungen` table, the check script name to be run and the machine on which the scripts should be run. See G.6.1 for the definition of this table, G.6.2 shows checks that are needed for the 'netbsd' exercise.

ergebnis_checks: After a check script is ran and the result (success or non-success) is available, this table stored the result of the exercise and check script for later feedback. See G.8.1 for the definition of this table. G.8.2 shows a list of all the results from booked exercise #129, first as stored in the `ergebnis_checks` table and second listed with descriptions from the `uebungs_checks` table as used in giving feedback on exercises later.

Appendix G contains the full list of definitions of all tables used in the Virtual Unix Lab. The named tables are used in many parts of the Virtual Unix Lab, including result verification, as described below.

Determining exercise data: When a user logs into the Virtual Unix Lab website, a check is made if there is an exercise prepared for him. This is done in `vulab/code/public_html/user/home/index.php`:

```
$sql="SELECT buchungen.buchungs_id AS buchungs_id, ".
     "       benutzer.login AS login, ".
     "       buchungen.uebung_id AS uebung_id, ".
     "       buchungen.startzeit AS startzeit, ".
     "       buchungen.freigegeben AS freigegeben, ".
     "       buchungen.datum AS datum, ".
     "       uebungen.dauer AS dauer ".
     "FROM buchungen, benutzer, uebungen ".
     "WHERE buchungen.user_id = benutzer.user_id ".
     "      AND uebungen.uebung_id = buchungen.uebung_id ".
     "      AND benutzer.login='". $HTTP_SESSION_VARS['login'] ."'";
...

$result = executeStatement($connection, $sql, $msg);
...

$now = time();
$start = mktime(
                substr($array['startzeit'],0,2), #h
                substr($array['startzeit'],3,2), #m
                substr($array['startzeit'],6,2), #s
                substr($array['datum'], 5,2),    #m
```

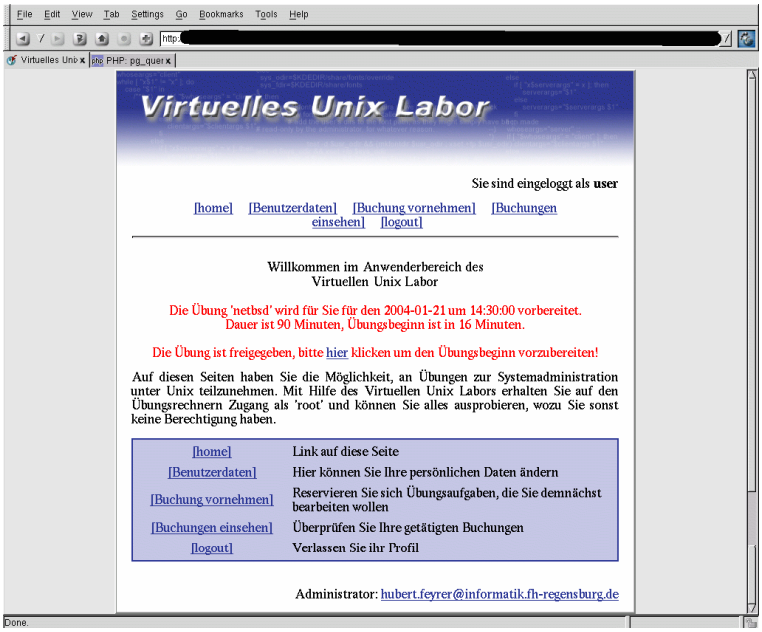


Figure 4: Logging in with an exercise prepared and ready to take

```
                substr($array['datum'], 8,2),    #d
                substr($array['datum'], 0,4));   #y
$dauers = 60*60 * substr($array['dauer'],0,2) +  #h
          60    * substr($array['dauer'],3,2) +  #m
          1     * substr($array['dauer'],6,2);   #s
$diff = (int)(($start-$now) / 60);
$dauerm = (int)($dauers / 60);
```

If the user comes in too early or too late, nothing is displayed about a booked exercise:

```
if ($now < $start - 2*60*60) continue;            # too early
if ($now > $start + $dauers) continue;            # too late
if ($array['freigegeben'] == 'nicht-mehr')        # over
        continue;
```

If the user logs on in time for the exercise, text is printed that invites the user to start the exercise, and provides a hypertext link to start the exercise as displayed in figure 4, passing the ID for the booked exercise ("`buchungs_id`") on with the following PHP code:

```
echo "Die Übung ist freigegeben, ";
echo "bitte <a href=\"${projectPath}index_session.php?menu=user&".
     "content=ueben/fwconfig".
        "&login=".$array['login'].
        "&buchungs_id=".$array['buchungs_id'].
        "&start=${start}".
```

```
        "&dauers=${dauers}".
        "&PHPSESSID=${PHPSESSID}".
        "\">hier</a> klicken ";
echo "um den Übungsbeginn vorzubereiten!\n ";
```

The `buchungs_id` is later used to identify the exact exercise that the user has booked, and associate exercise data and results with it.

Exercise text: When taking an exercise, its ID is passed in from the login webpage and used to determining the filename of the exercise text from the `uebungen` table. The filename is given in the "`text`" field of the `uebungen` table, and the exercise's text is then read from the `vulab/code/public_html/texte` directory using the PHP `require_once` statement.

From `vulab/code/public_html/user/ueben/start.php`:

```
$sql = "SELECT uebungen.uebung_id AS uebung_id, ".
       "       uebungen.bezeichnung AS bezeichnung, ".
       "       uebungen.text AS text, ".
       "       uebungen.mehr_info AS mehr_info ".
       "FROM buchungen, uebungen ".
       "WHERE buchungen.buchungs_id = '$buchungs_id' ".
       "      AND buchungen.uebung_id = uebungen.uebung_id";
...

$fn="$textPath/$text";
...

require_once $absoluteProjectPath . "$fn";
```

Besides printing of the exercise text, the `start.php` script prepends HTML headers and footers, including a countdown that shows how much time is left of the exercise, and a "Fertig!" (Done) button which can be used to signal that the exercise is finished and the result should be verified.

An example exercise text in plain ASCII and rendered in a web browser as displayed at exercise time is displayed in figure 5.

Check-scripts: Besides the exercise text, the check scripts are the second part that needs to be created for a specific exercise. In the first implementation step of the Virtual Unix Lab, check scripts were written in both Perl and Bourne shell. Example shell scripts can be found in appendix F.1 for various checks of the 'netbsd' exercise, e.g.:

- Check if a specific user exists, using the finger(1) command to interface to the system's user database (F.1.1)
- Checks if a specific user exists in the NetBSD password database, `/etc/master.passwd`. (F.1.2)
- Check if both the bash and tcsh packages are installed. (F.1.3)
- Check if a certain user's password is set to a specific string. Perl is used for the implementation to access the DES encryption routines, crypt(). (F.1.4)

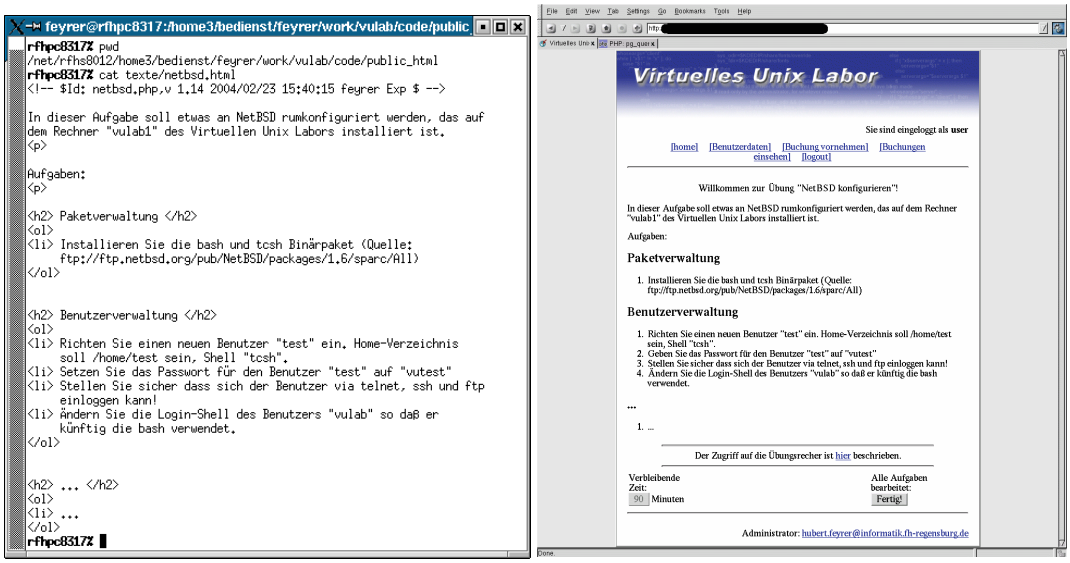


Figure 5: Exercise text with no associated checks in plain ASCII and rendered in web browser

- Check if the login shell of user 'vulab' is set to bash. (F.1.5)

The check scripts were mostly written in Bourne Shell (/bin/sh) where possible for rapid development, perl was mostly used to interface system routines that were not available for shell programming.

Web-frontend to database: The web frontend to associate check scripts with a certain exercise was written in PHP, like the rest of the web interface of the Virtual Unix Lab. It takes input from the administrator designing the exercise, and stores it into the PostgreSQL database using the "pgsql" PHP-module.

Screenshots of the web interface for defining an exercise can be seen in figure 6, displaying the three screens for defining part of the exercise:

1. Input of general information (fig. 6, a)): textual description of the exercise, preparation and duration of the exercises, time to reserve for running checks and other possible postprocessing, possibly restricting the exercise to a single user (more on this later), filename of the exercise's text file and a text file for more information on the exercise (unused).
2. Which harddisk-image to deploy to which machine (fig. 6, b)): the second screen defines how many and which of the lab machines will participate in the exercise, and which harddisk image to deploy to them. The harddisk image contains the full content of the harddisk used on the lab machine during the exercise. Installations of Solaris 9/sparc and NetBSD 1.6.2/i386 were available for exercises, adding e.g. Linux would have been just a matter of preparing a harddisk im-

age for the lab machine, and adjusting the exercise.
If an exercise just requires one machine, only that one will be installed for the exercise the other lab machine(s) won't be touched.

3. What checks to run at the end of the exercise, and on which machines (fig. 6, c)): This dialogue presents all available check scripts (as found in the `/vulab` directory on the master machine, which also runs the web frontend), and allows selecting which script to run on which of either the lab machines, or the master machine ("localhost") to run the script for checking if part of the exercise was completed successfully or not. For later feedback to the exercising user, a description can also be given that will be printed together with an indicator of success or failure of the tested item in question.

Result verification engine: The script `uebung_auswerten` does the main work of the result verification described under "Verification" in section 2.1. It is written in perl to have both access to the PostgreSQL database and to the system interfaces for running the check scripts.

There are two ways for the `uebung_auswerten` script to get started:

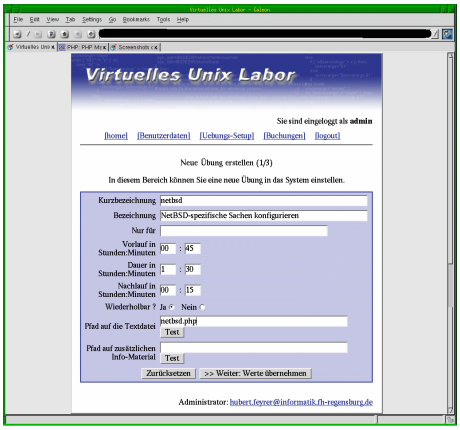
1. The obvious one is when the user starts the exercise via the web interface and clicks on the "Fertig!" (Done) button. The PHP script handling the button press from the web frontend (`vulab/code/public_html/-user/ueben/ende.php`) calls it to get the result verification done while a message is displayed to the user.
2. When a user books an exercise, the exercise is prepared. If the user doesn't show up, the exercise must still be terminated properly. When the user doesn't show up, the web frontend can't be used to do this. Instead, after the first at(1)-job which installs the lab machines has finished, a second at(1)-job is started that will call the script at the official ending time of the exercise. This second at(1)-job's id is stored in the `buchungen` table's "`at_id_end`" field, and it will be either ran by the cron/at job scheduling system part of the NetBSD system if the user doesn't log into the web frontend, or it will be canceled to not run when the user shows up and finishes the exercise by pressing the "Fertig!" (Done) button.

In either case, retrieving which check scripts to run is extracted from the database:

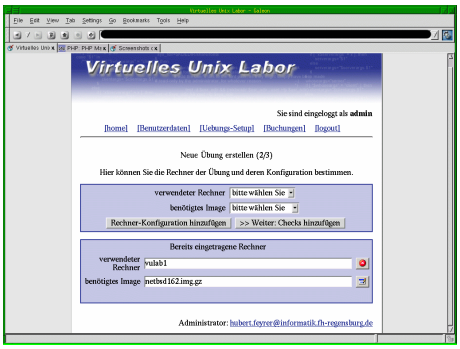
```
$dbh = DBI->connect("dbi:Pg:", "vulab", "", { AutoCommit => 1 })
        or die "cannot connect to DB";

$sth = $dbh->prepare("SELECT uebungs_checks.check_id AS check_id, ".
                     "       uebungs_checks.script AS script, ".
                     "       uebungs_checks.parameter AS parameter, ".
                     "       uebungs_checks.rechner AS rechner, ".
                     "       uebungs_checks.uebung_id AS uebung_id ".
                     "FROM buchungen, uebungs_checks  ".
                     "WHERE uebungs_checks.uebung_id = buchungen.uebung_id ".
                     "      AND buchungs_id = $buchungs_id ".
                     "ORDER BY check_id");
```

a)



b)



c)

Figure 6: Defining an exercise: a) general properties, b) which image to deploy on which machine, and c) what checks to run on which machine.

```
$sth->execute();
while ( @row = $sth->fetchrow_array) {
        ( $check_id, $script, $parameter, $rechner, $uebung_id ) = @row;
        ...
```

After running the check script, the results are stored in the database using the PostgreSQL database interface routines from perl again:

```
$sth2 = $dbh->prepare("INSERT INTO ergebnis_checks ".
                "(buchungs_id, check_id, erfolg) ".
                "VALUES ($buchungs_id, $check_id, $erfolg)");
$sth2->execute();
```

Determine interpreter for check scripts: In the first implementation step of the Virtual Unix Lab, part of the check scripts were written as Bourne shell (/bin/sh) scripts, others were written in Perl. An early prototype of the Virtual Unix Lab code had a separate field for the language (interpreter) reserved in the database's `uebung_checks` table to identify which interpreter to use, but that field was removed in the first implementation. Instead, before running the script (actually, before sending it to the remote machine, see below), the first line of the script was examined, which – following common Unix tradition – contained either "`#!/bin/sh`" to indicate a Bourne shell script, or "`#!/usr/local/bin/perl`" for Perl scripts. As such, if the string "perl" occurred in the first line of a check script, it was assumed to be a Perl script, else a Bourne shell script:

```
# Determine Interpreter:
open(S, "$checkscript_path/$script");
$i = <S>;
close(S);

if ($i =~ /perl/) {
        $interpreter = "perl";
} elsif ($i =~ /[^c]sh/) {
        $interpreter = "sh";
}
#print "interpreter=  $interpreter\n";
```

Using these heuristics proved to be good enough to be employed also in later versions of the Virtual Unix Lab with only minor changes, see section 3.2.

Execution of check scripts: An obvious way to run a check script is to copy it to the remote machine, execute it, collect the result value, and remove the script again.

An optimisation was done to avoid copying and removing the file. Every command line interpreter that can be used by writing the first line of the script as "`#!/path/to/interpreter`" gets invoked by the system with the script as it's first argument[1]. The fact that interpreters also allow feeding program text to be run via the standard input can be used for further optimizing, i.e.

[1] [Stevens, 1992] pp. 217

```
perl < somescript
sh   < somescript
```

and

```
cat somescript | perl
cat somescript | sh
```

are equal, and the connection of the script to standard input of the interpreter can even be extended via a pipe to a remote system, i.e.

```
cat somescript | rsh remotehost perl
cat somescript | rsh remotehost sh
```

are valid and work as expected. The script's output is expected via it's standard output, and analyzed for a certain string in the last line to indicate either success (indicated by saying either "`0`", "`ok`" or "`success`") or failure (else):

```
$cmd="cat $checkscript_path/$script ";
if ($rechner eq 'localhost') {
        $cmd.="| ";
} else {
        $cmd.="| rsh -p 9999 $rechner ";
}
$cmd .= "$interpreter 2>&1";

print "check_id=$check_id:\n";
print "cmd=$cmd\n";
open(CMD, "$cmd |");
while($rc = <CMD>){
        $lastline=$rc;
        chomp($lastline);
        print "\t> $lastline\n";
}
$erfolg = ($lastline eq "0"
           or $lastline =~ /^success/i
           or $lastline =~ /^ok/i) ? "true" : "false";
```

3 Step II: Instructions and Checks coupled

Step II of the Virtual Unix Lab consists of a number of components that build upon the design and implementation described in the precious sections. Besides improvements made to the check script infrastructure and the stereotypes they provide as base for a domain specific language, the most important change in step II of the Virtual Unix Lab is that checks are coupled with exercise text, which improves the feedback given to users. This section describes the design of step II, followed by selected aspects of their implementation.

3.1 Design overview

This section describes how check scripts were improved, and how coupling of exercise text and checks was achieved by using a domain specific language. This also supports giving elaborated feedback, and it allowed creating a system front-end with check primitives. For these aspects, integration and interaction with the existing system is illustrated. The topic is covered in more detail in [Feyrer, 2008].

3.1.1 Improved Check Primitives

The check scripts that implement the verification primitives were improved in several ways. To be of more general use the checks were implemented using one common language and framework for all scripts. Common tasks were then identified and expressed in more generic scripts that were taught how to handle parameters to accommodate to the specific tests. The following items describe the changes that were made to the check scripts in detail.

Rewrite all check scripts in perl: For step II of the Virtual Unix Lab, all check scripts were rewritten in perl, and named to indicate their scope of applicability, i.e. if they can be used on all systems, on Unix systems, or only on specific Unix(like) systems, by giving them common filename prefixes.

Extend check scripts (etc.) to handle parameters: Passing parameters into a check script involved several of the Virtual Unix Lab components: Besides the check scripts that needed changes to accept parameters, the parameters had to be stored in the database, the (web based) interface to store and edit the check data in the database had to be adjusted, and an interface had to be defined to pass the parameters to the scripts when running it. Furthermore, an interface was introduced to query a check script for it's purpose and the parameters it supported.

Improvements of check scripts: The check scripts used in step I of the Virtual Unix Lab only tested one aspect of the system as has been discussed before, and testing two similar aspects following the same concept required two separate shell scripts. Following the description of Cocke and Schwartz, the check scripts were improved to provide "indicative subpatterns" to be embedded into exercise texts and that provide the "contextually implied information", i.e. they act as a collection of subroutines that can be called for specific check tasks when needed[1].

As a summary, the initial set of task-specific check scripts was changed into a set of check scripts that test specific aspects of the system. Parameters can be

[1] [Cocke and Schwartz, 1970] pp. 10

Figure 7: Extended web interface to enter parameters for check script

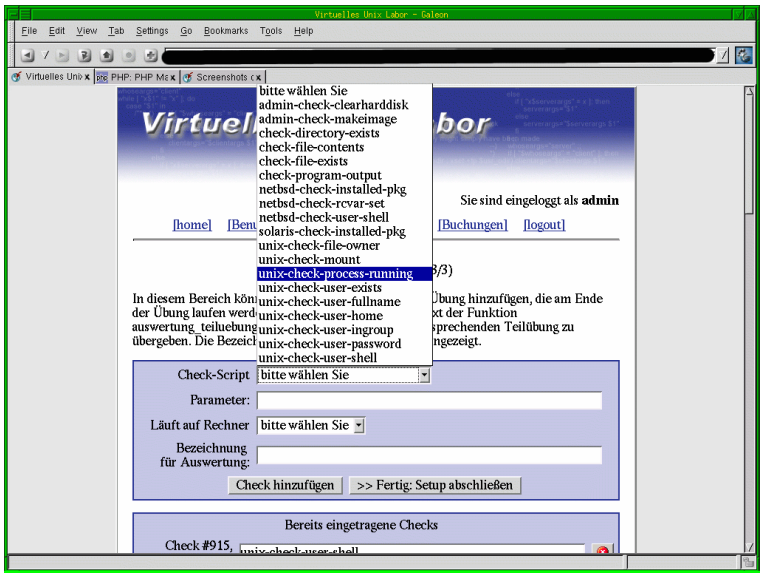


Figure 8: Listing existing checks

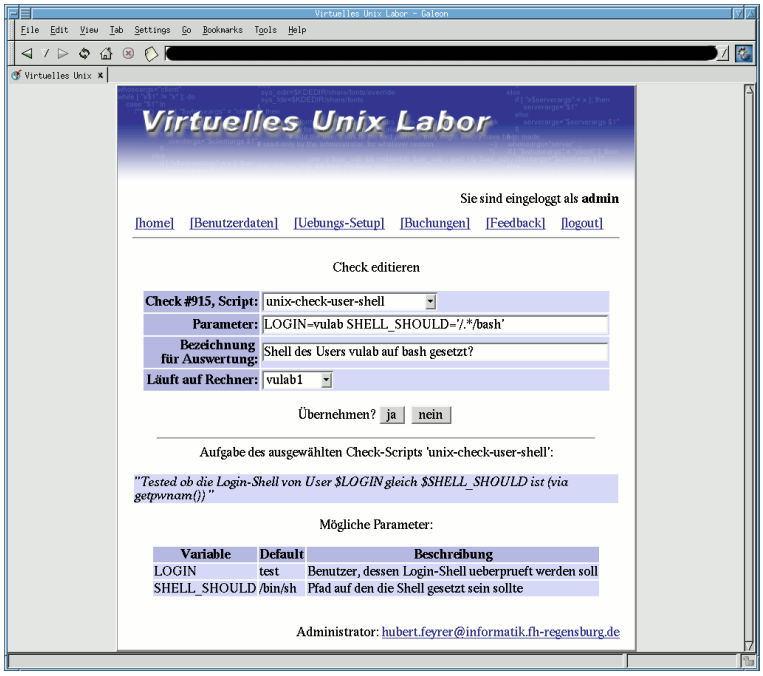


Figure 9: Possible parameters of a check script, and their description

given to the scripts to specify which aspects of the specific subsystem to examine closer.

3.1.2 Coupling of Exercise Text and Checks

To give detailed feedback on each task of the exercise, an association needs to be made between the textual description of a specific task of the exercise, and the check(s) that verify the results of that task.

Figure 10 a) illustrates the uncoupled exercise text and checks used in step I of the Virtual Unix Lab, while figure 10 b) shows the coupling realized in step II.

So far, the per-check data was stored in the `uebungs_checks` table. The idea for improvement was now to place this data into the exercise text, to keep check data near the exercise text, e.g. have something like:

```
1. Perform some task on host vulab1 with parameters x and y.
   // Check 1: run check-task-done (no parameters) on host vulab1
   //          Feedback: ‘‘Was the task performed successfully?’’
   // Check 2: run check-task-parm (PARM=x) on host vulab1
   //          Feedback: ‘‘Does the task use parameter x?’’
   // Check 3: run check-task-parm (PARM=y) on host vulab1
```

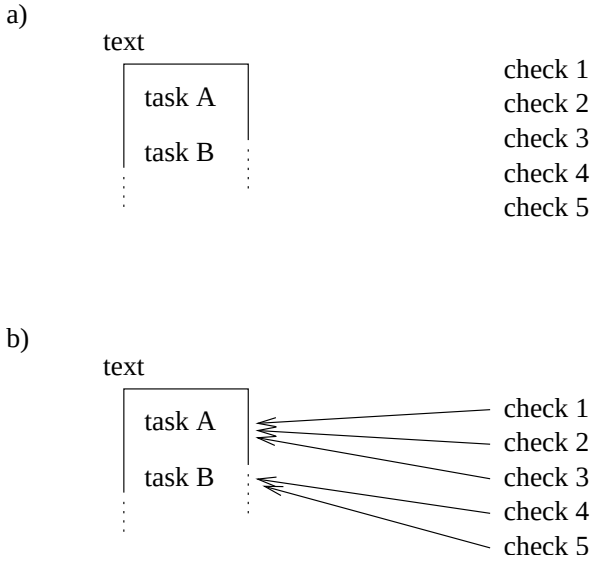


Figure 10: Exercise text and checks: a) uncoupled in step I, b) coupled in step II

```
//          Feedback: ‘‘Does the task use parameter y?’’
```

This example first describes the task to the user in textual form, then contains some comments in the PHP text to indicate the check data. While preventing the displaying of the check data was possible by using HTML or PHP comments, there were two problems given with this approach. First, how to extract the check data for running when the exercise is over, and second how to display the feedback to the user.

The first problem was solved by a processor using the “data structure representation” pattern[1] and an appropriate processor. The processor – realized as a perl-script called “`uebung2db`” – is described in more detail in section 3.1.5.

To give feedback to the user after the exercise, the final exercise design consists of the exercise text being written in HTML text with PHP functions included that control printing of evaluation as well as hints stored in a PHP comment which give the data for the associated check, and which is stored in the database by the `uebung2db` processor.

[1] [Spinellis, 2001]

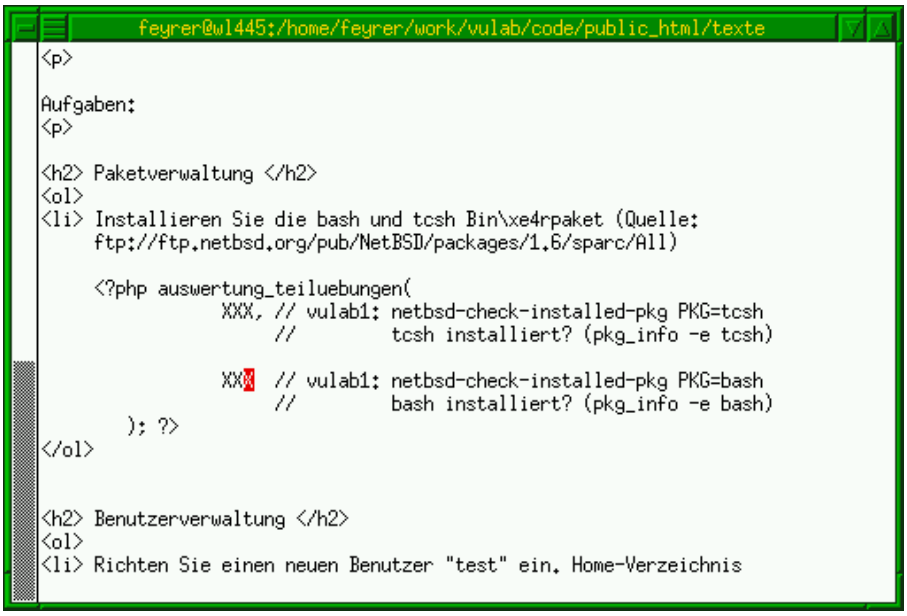


Figure 11: Example exercise text with check data

3.1.3 Feedback

Giving proper feedback on what tasks of an exercise where solved successfully and which were not was one of the primary design goals of step II of the Virtual Unix Lab. Given the exercise design described in the previous section, it was easy to realize giving feedback to both single users as well as teachers.

Key elements for giving feedback are the PHP functions embedded into the exercise text as shown in figure 11, which allow controlling information on what checks are printed for feedback:

- `auswertung_ueberschrift()`
- `auswertung_teiluebungen()`
- `auswertung_zusammenfassung()`

For presenting the exercise text to the user when previewing and while taking the exercise, these functions don't print anything at all.

To give feedback for a user after the exercise, these functions are defined differently. While `auswertung_ueberschrift()` and `auswertung_zusammenfassung()` give general information including a header and footer for the exercise, the main work is done by `auswertung_teiluebungen()`. The

function takes a list of check IDs, and it retrieves and prints the corresponding textual description of the check (from the "`bezeichnung`" field of the `uebungs_checks` table) as well as the result (stored in the "`erfolg`" field of the `ergebnis_checks` table).

3.1.4 Creating a System Front-End with Check Scripts

Besides exercise texts, the other important part of an exercise is the machine setup provided for an exercises that users start with. This machine setup is stored in form of harddisk images that are written to the lab machines before the exercise starts.

Up to step II, the process of creating or updating a harddisk image was done manually by first preventing any exercises being taken for some time (usually by disabling logins in the Virtual Unix Lab), then – when updating an existing image – issuing the command to deploy an existing image, or installing a machine from CDROM and configure it so that it's configuration can be used for the exercise in mind. After that, the machine had to be shut down and netbooted, and from the netboot environment, the harddisk image was taken and written to the Virtual Unix Lab master machine via NFS. After entering the newly created image file into the `images` table with an appropriate SQL statement, the new/updated image was ready to be used in newly created exercises.

Looking closer at the process, doing most of these steps automatically is easily possible though: The normal exercise system can be used to book a certain "admin-type" exercise, which will prevent users from interrupting the process, and which will also allow installing a predefined image on the machine for updating (if wanted). Normal users are prevented from booking the exercise. When the exercise time arrives, the machine will be prepared, and instead of doing a predefined exercise, the administrator changes the client machine as needed.

The evaluation consists of two special "admin" check scripts that will care to do the postprocessing done manually before, `admin-check-clearharddisk`[1] and `admin-check-makeimage`[2]. The first script cleans up any unused space on the lab machine's harddisk and prepares it to be better compressable, while the second script does all the real work of shutting down the lab machine, taking precautions so a netboot will create a harddisk image in a given file, perform the netboot, wait until the image file is created and storing the newly created image's filename in the `images` table.

[1] See appendix F.2.1
[2] See appendix F.2.2

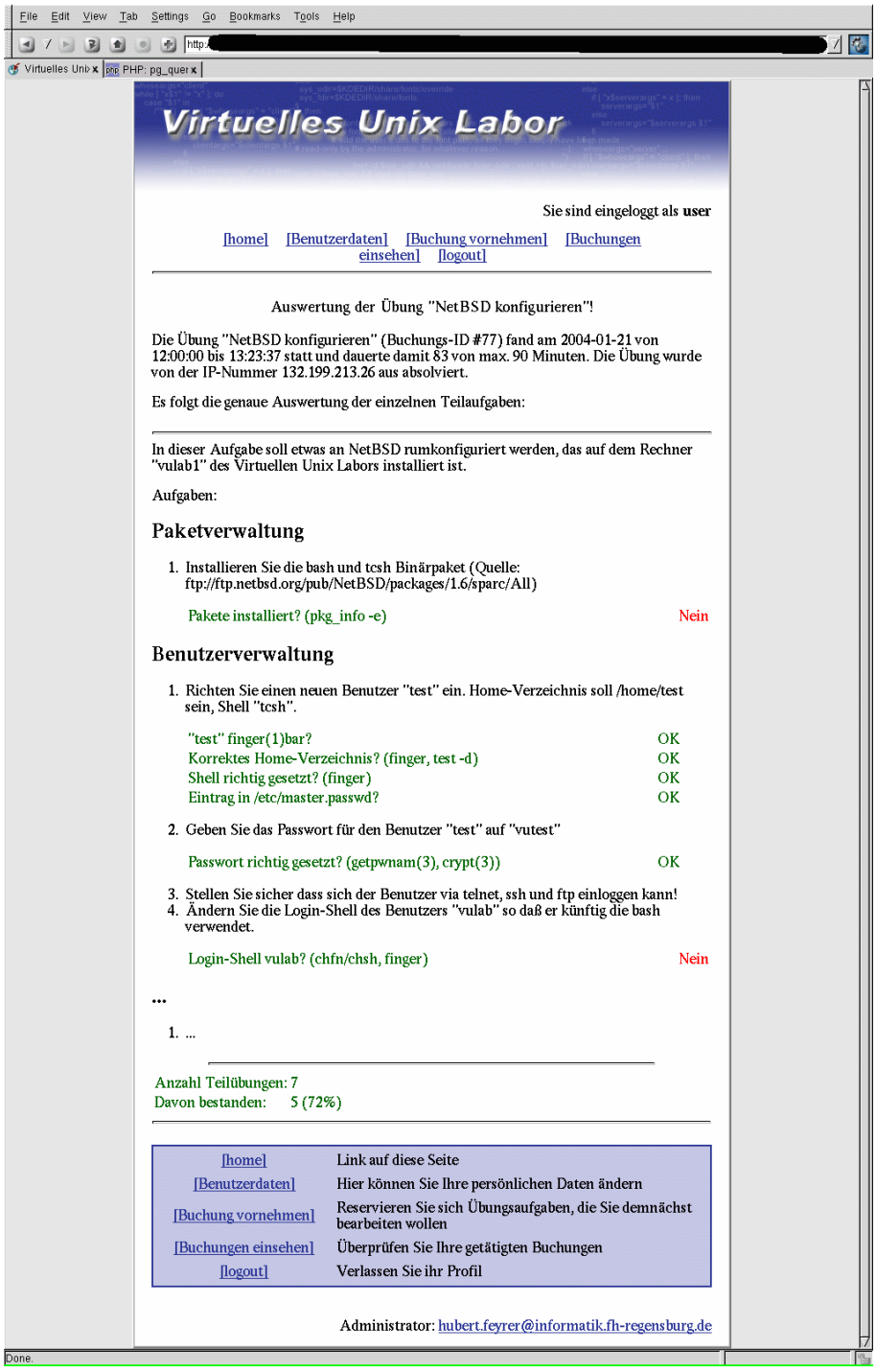


Figure 12: Giving feedback on an exercise for a single user

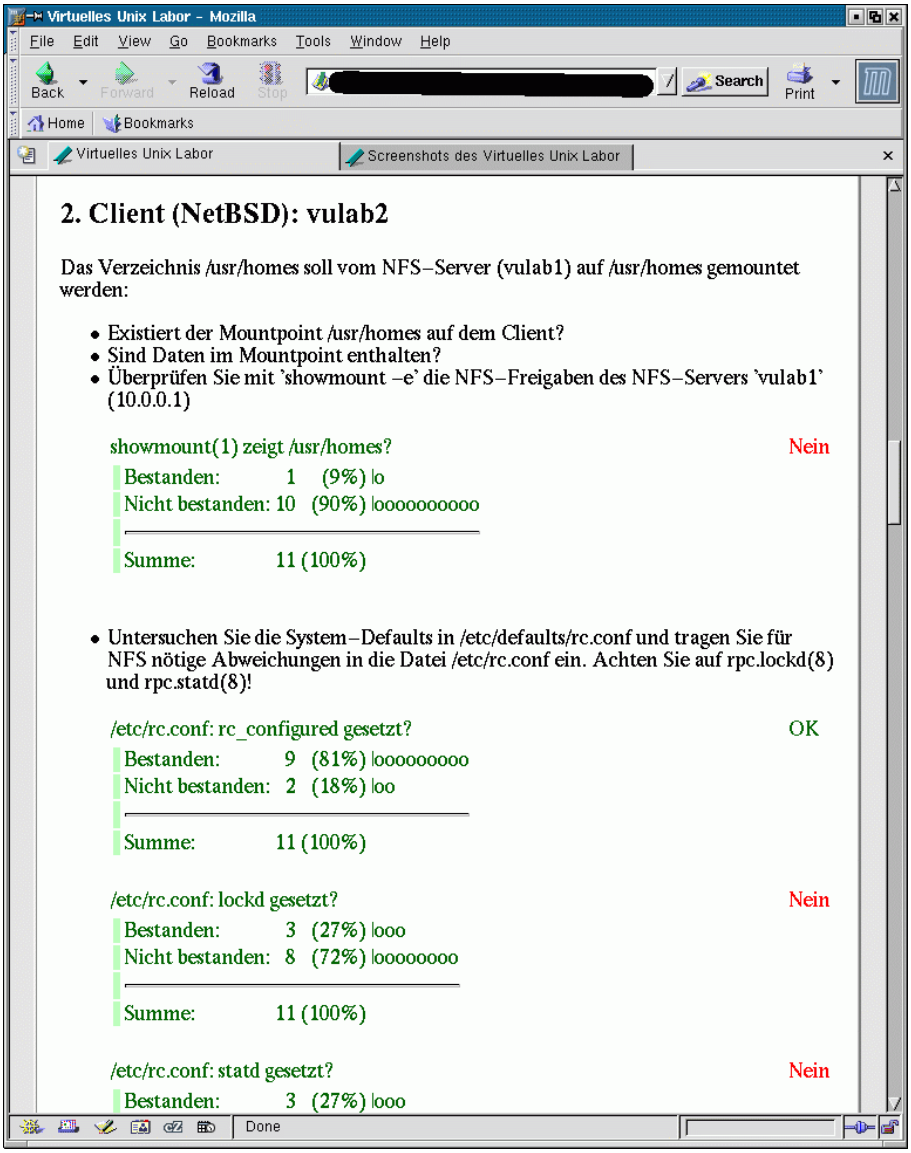


Figure 13: Giving teacher/admin feedback for all users which took an exercise

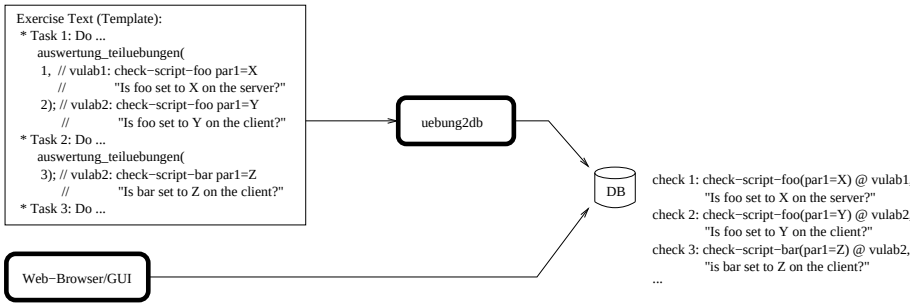


Figure 14: Step II: Preparation

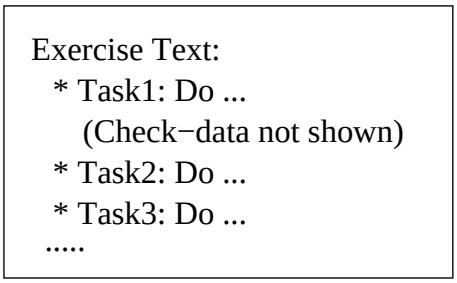


Figure 15: Step II: Exercise

3.1.5 Integration and Interaction

After describing all the major new components and features of step II of the Virtual Unix Lab, this section illustrates their integration and interaction, see figures 14, 15, 16, and 17 for an overview.

Preparation: Creation of an exercise in step II of the Virtual Unix Lab is similar to step I with a few changes in detail:

- Define the exercise with it's general parameters by using the web frontend as displayed in figure 6 of step I.
- Write the exercise text in HTML as in step I with hints for result verification and giving feedback embedded as comments comes next
- After writing the exercise text, the hints are extracted into the database by running the script "`uebung2db`" as shown in figure 18 b).
- After the updated exercise text has been reviewed, it needs to be put into place.

After these steps – define general properties, write exercise text, fill database from exercise text, move updated exercise text into place – the exercise is prepared, and it can be used for exercises by students.

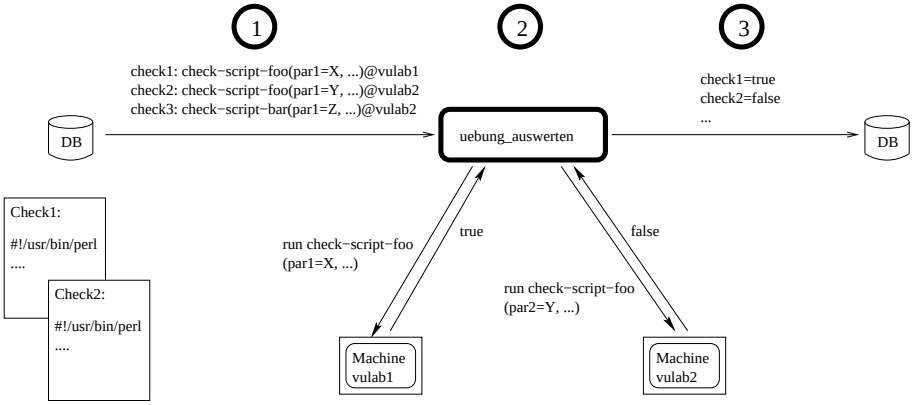


Figure 16: Step II: Verification

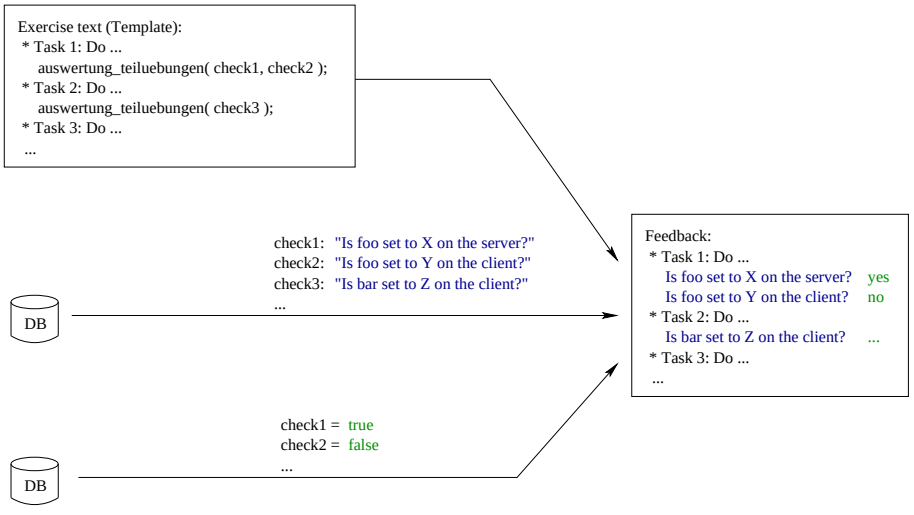
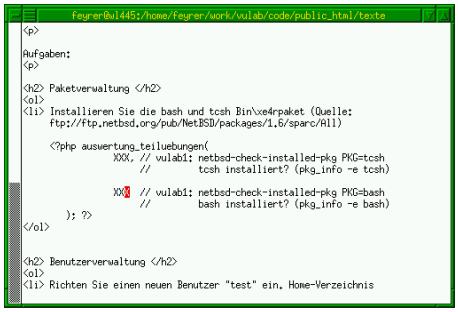
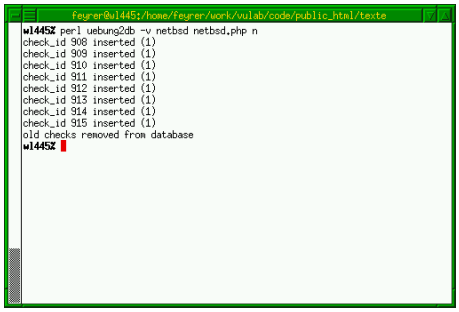


Figure 17: Step II: Feedback

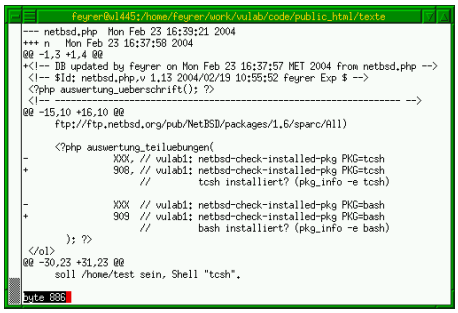
a)



b)



c)



d)

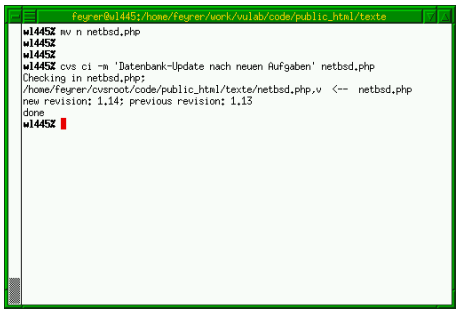


Figure 18: a) Writing exercise text and hints, b) extracting hints into database and writing new text with check-numbers retrieving results for feedback, c) comparing original and updated exercise text and d) moving the updated exercise text into place and saving to the CMS

Exercise: The exercise text is stored in a HTML file with calls to PHP function `auswertung_ueberschrift()`, `auswertung_teiluebungen()` and `auswertung_zusammenfassung()` as described in section 3.1.3.

Verification: Verification of the exercise results consists of almost the same procedure as in step I, as illustrated in figures 14, 15, 16, and 17:

1. Retrieve which check scripts to run, what parameters to pass to them, and on which machine to run them from the database.
   An additional field "`parameter`" was added to the `uebungs_checks` table, where the "`uebung2db`" script stored the parameters for the call of the shell script. This field is retrieved in addition to the data already required in step I.
2. Run the check script with the parameters from the database, and collect the result.
   The same procedure is used as in step I, i.e. when the script should run on a lab machine, it is sent to an interpreter running on the lab machine, and the output printed is collected to see if the check was success or failure. Parameters for the scripts are passed as environment variables when starting the interpreter, so the check scripts can retrieve and use the parameters.
3. Store the check script's result into the database.
   There is no changes from step I here. The textual output of the check script is scanned for an indicator of success or failure, and the boolean "`erfolg`" (success) field of the `ergebnis_checks` table is set accordingly.

Analysis: At any time, a list of all booked exercises ever can be retrieved. Exercises that have already been taken do have a button that can be used to analyze that particular exercise and retrieve feedback it.

As described in section 3.1.3, the Virtual Unix Lab system then displays the exercise text's HTML file, and runs the embedded PHP functions to show not only the textual descriptions of the tasks, but also details on what the checks tested and if the tests were successful or not.

In detail, the PHP code in the exercise text first calls `auswertung_ueberschrift()` and prints a header with general information about the exercise: date and time of start and end, duration in minutes and the IP number from which the exercise was taken as stored by the firewall configuration when the lab was entered for the exercise. The exercise text containing a textual description of the tasks to perform on the lab machines is displayed next, augmented with calls to `auswertung_teiluebungen()` which does the main job of giving feedback.

The `auswertung_teiluebungen()` function takes a variable number of arguments, each representing a check-number. For each of the check numbers, it retrieves

- the textual description of the check as stored in the "`bezeichnung`" (description) field of the `uebungs_checks` table, and

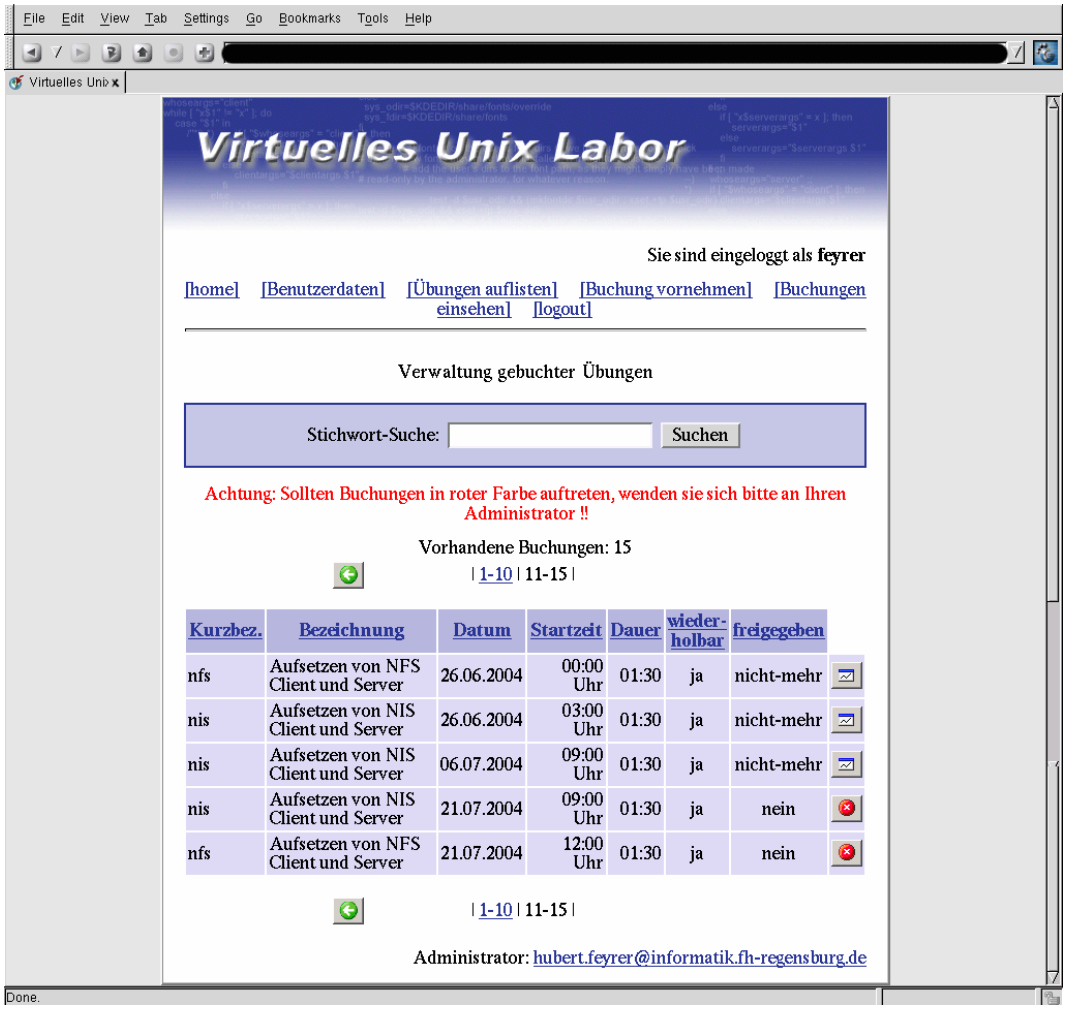


Figure 19: The list of booked exercises contains both completed exercises for which feedback can be requested as well as uncompleted exercises that have not yet started

- the result of the check as stored in the "`erfolg`" (success) field of the `ergebnis_checks` table.

If the feedback is not requested by a "normal" user but by one with administrator privileges in the Virtual Unix Lab, an overview of all students' performance as in figure 13 is shown in addition to the single user's result. For the admin-feedback, the numbers of students are determined who did pass and fail the check. The sum of both gives the total number of students who have taken that exam (100%), and both absolute numbers and percentage of the students who passed and failed on the check in question are printed. In addition, small bars of "`o`"s are printed besides each result which represent one student each, to allow a quick graphical overview on the result.

From the description of these phases, it can be seen that there are a number of small and medium size changes, but that the general design and implementation of the Virtual Unix Lab result verification architecture could have been kept for step II.

3.2 Implementation details

The previous sections have given a conceptual, user-level view of step II of the Virtual Unix Lab, with an emphasis on the changes to address the issues found in the first implementation. This section provides details on the implementation of a number of key components of step II to illustrate some of the details described so far in more depth and to improve understanding of the various components, their design and interaction.

Naming of check scripts: Check scripts were changed to check varying aspects of the target system as described in section 3.1.1, with details passed as parameters to the check script. In order to show the aspect that the script checks as well as the scope it can be used on – all operating systems, all Unix or only specific Unix systems, Microsoft Windows – the naming scheme for check scripts was changed to first give the scope, followed by a fixed `-check-` which can be used to find scripts easily followed by aspects checked in the scope, possibly with more details if e.g. several aspects of a user account can be checked (existence, specific fullname, home directory, ...).

The following scripts are available in step II of the Virtual Unix Lab to check aspects ...

- ... on Windows and Unix systems:

```
smaug# cd /vulab
smaug# ls check-*
```

```
check-directory-exists    check-file-exists         check-template.pl
check-file-contents       check-program-output      check-template.sh
```

- ... on Unix systems only:

```
smaug# cd /vulab
smaug# ls unix-check-*
unix-check-file-owner                   unix-check-user-home
unix-check-mount                        unix-check-user-ingroup
unix-check-process-running              unix-check-user-password
unix-check-user-exists                  unix-check-user-shell
unix-check-user-fullname
```

- ... on a specific Unix(like) system only:

```
smaug# cd /vulab
smaug# ls {solaris,netbsd}-check-*
netbsd-check-installed-pkg              netbsd-check-user-shell
netbsd-check-rcvar-set                  solaris-check-installed-pkg
```

Framework for check-scripts: Although each of the check scripts checks an unique aspect of the target system, the code used to implement the script contains a few parts that are common to all check scripts. These common parts implement option parsing, parsing of parameters, printing the purpose of the script and the parameters it accepts. On the other hand, the only parts specific to every script are:

- the description of what the check script does, stored in the `$WHATIS` variable, for printing if the "`whatis`" command line argument is given.
  Example:

```
$WHATIS='
***
*** Tested ob FILE den regulären Ausdruck CONTENT_SHOULD enthält
***
';
```

- the possible parameters it takes, stored in the array `@vars`, for both processing them during normal operation of the check script as well as giving a description when "`listparms`" is given as a command line argument.
  Example:

```
@vars = (
         [ "FILE", "/etc/motd",
           "zu durchsuchende Datei, absoluter Pfad" ],
         [ "CONTENT_SHOULD", "Hallo Welt!",
           "zu suchender Regulärer Ausdruck" ]
         );
```

- the code to verify the aspect of the system in question, contained in the `check()` function, which is executed if the check script is not run with any of the "`whatis`" and "`listparms`" command line arguments. Parameters are available as perl variables with the same names as the parameters passed as environment variables, see the description of the `init()` function below.
  Example:

```
# Check-Spezifisch:
sub check()
{
    print "FILE=$FILE\n";
    print "CONTENT_SHOULD=$CONTENT_SHOULD\n";
    print "";

    $rc = "wrong";
    if (open(F, "$FILE")) {
        while(<F>){
            chomp();
            #print "$_\n";
            if(/$CONTENT_SHOULD/){
                $rc = "ok";
                last;
            }
        }
        close(F);
    }

    print "$rc\n";
}
```

The above example code is taken from the `check-file-contents` check script, which is available in appendix F.2.3. A full list with descriptions of all check scripts and their parameters is listed in appendix E. Assignment of the unique parts of the check script needs to retain a certain format for further processing, as shown in the above list.

The common code to handle this data specific to each check script consists of the following parts:

- a visually impacting comment to make an optical barrier between the part of a check script that should be changed, and the part that should not:

  ```
  ###############################################################################
  ### AB HIER NICHTS MEHR VERAENDERN !!!
  ###############################################################################
  ### Common code:
  ```

- A helper function to see if any of the accepted parameters (as listed in the `@vars` variable) exists in the environment, and if so, to assign the corresponding perl variable. E.g. if the environment variable `LOGIN` is set, the perl variable `$LOGIN` will be set to the same value for further processing:

  ```
  # Variablen übernehmen
  sub init()
  {
          for($i=0; $i<=$#vars; $i++) {
                  if(exists($ENV{$vars[$i][0]})) {
                          ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                  } else {
                          ${$vars[$i][0]} = $vars[$i][1];
                  }
          }
  }
  ```

- A check if "`listparms`" is passed as command line argument, and if so, the list of possible parameters as stored in the `@vars` variable is printed. For each parameter, it's name, default and description is printed, separated by a "`|`" for easy further processing:

```
# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
```

- If "`whatis`" is given as command line parameter, the content of the `$WHATIS` variable is printed, after removing the components that make it nice to read in the check script's source code:

```
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
```

- If `-h` is given, an list of all command line parameters and a description of what they do is printed:

```
}elsif($ARGV[0] eq "-h") {
       print "whatis     Kurzbeschreibung des Scripts\n";
       print "listparms  Listet Variablen mit Default und Beschreibung\n";
       print "-h         Alle Parameter\n";
       print "sonst      Check-Script wird ausgefuehrt\n";
```

- If no special command line argument is given, the `check()` function itself is called after initializing any possible parameters with the help of the `init()` function described above:

```
} else {
    init();
    check();
}
```

The common code described here is used in all perl-based check scripts used in step II of the Virtual Unix Lab.

A possible area for optimisation for future versions of the Virtual Unix Lab would be to remove the common code stored in all check scripts so far, store it in a file on it's own, and pass if after the check-specific (first) part of the script: "`cat foo-check-bar check-common-functions | perl`".

Extracting check-data from exercise texts: The `uebung2db` program acts as processor of the Verification Unit Domain Specific Language, its operation and functionality are described in section 3.1.5.

The program implements the "data structure representation" pattern introduced in [Spinellis, 2001] by using simple lexical analysis of the exercise text to extract check-related data, store it into the relevant database tables, and write an updated exercise text with updates of the related check-numbers for the `auswertung_teiluebungen()` filled in where only

placeholders (XXX) were used before. The following text describes a few details of the implementation of the uebung2db script, which is listed in full length in appendix C.

The script expects a number of command line parameters as described in the "Preparations" part of section 3.1.5 and shown in figure 18 b). Most important are the input exercise text and the name of the output file. As the exercise ID is given to the "uebung2db" script as well, any checks associated with previous versions of the exercise can be removed from the database's uebungs_checks table.

The first action of uebung2db is to write a commend header when the exercise's checks were extracted and stored in the database:

```
sub header() {
    local($now);
    chomp($now = ‘date‘);
    print OUTPUT "<!-- DB updated by $ENV{’USER’} on $now from $template -->\n";
}
```

Following that, the main routine reads through exercise text, and does lexical matching of any lines indicating a check script call, which are identified by either a check number (when an exercise already has checks stored in the database) or XXX (if the exercise hasn't been added to the database so far), followed by the PHP commend introducer //, the machine on which to run the check on, the check script name and any possible parameters:

```
    open(T, $template) or die "can’t read $template: $!\n";
    while(<T>) {
        chomp;
        ($check_id, $komma, $rechner, $script, $parameter) =
            m@\s*([0-9X?]+)([,  ])?\s*//\s+([a-zA-Z0-9_]*):\s+([^ ]*check-[^ ]+)\s+(.*)@;

        if ($rechner eq "") {
            print OUTPUT "$_\n"
                if !/Generated by.* on .* from/; # skip header
            next;
        }
```

See figure 11 and appendix B for examples of exercise texts. Lines not matching a check-script "call" are copied verbatim into the updated exercise file.

If a check script "call" is identified, a consistency checks is performed to verify that if the check script exists:

```
        ### 1. Syntax-Check etc.

        ## Check if script present
        if ( ! -f "$checkscript_path/$script" ) {
            warning("missing check-script ’$script’");
            next;
        }
```

This test assumes that the “uebung2db” script is ran on the Virtual Unix Lab master machine, as the check scripts need to be available, plus access to the Virtual Unix Lab database must be available.

Before further consistency checks are made, the textual description of the check as printed for feedback purpose is stored into the perl variable $bezeichnung:

```
chomp($bezeichnung = <T>);
if ($bezeichnung !~ m@^\s*//\s*\S+\s*@) {
    warning("no comment for $check_id ($rechner: $script $parameter)");
}
$bezeichnung =~ s,^\s*//\s*,,;
$bezeichnung =~ s,\s*$,,;
```

The next two consistency checks test ...

- ... if the machine on which the check script should run, which is stored in the $rechner perl variable, is known in the rechner (“machines”) database table, which lists the valid Virtual Unix Lab hosts:

  ```
  ## Rechner bekannt?
  $sth = $dbh->prepare("SELECT * ".
                           "FROM rechner ".
                           "WHERE bezeichnung='$rechner'");
  $sth->execute();
  while(@row = $sth->fetchrow_array) {
      if ($row[0] eq $rechner ) {
          print "    rechner OK: $rechner\n"
              if $debug;
      } else {
          warning("rechnercheck unknown host: $rechner");
      }
  }
  ```

- ... if the parameters for the check script, stored in the $parameter perl variable, match the parameters that the check script claims to support in the list it prints when called with the listparms command line argument.

  To do so, the list of supported parameters is determined calling the check script with listparms first:

  ```
  ## Check parameters
  # Get possible parms
  open(P, "$interpreter $checkscript_path/$script listparms |")
      or die "Can't listparms for $script: $!\n";
  while(<P>) {
      @p = split(/\|/);
      $par{$p[0]} = $p[1];
      #print " $p[0]";
  }
  close(P);
  #print "\n";
  ```

  Next, the parameters from the exercise text are passed to the Bourne shell (implicitly called by the perl open() function here) and the env(1) command, which sets and lists environment variables and their values

set[1]. The list printed by the env(1) command is checked against the known variables from the previous step to detect syntax errors:

```
# Parse into variables using sh & env
open(P, "env -i $parameter env |")
    or die "Can't env(1) $parameter";
while(<P>) {
    chomp();
    ($var, $val) = /([a-zA-Z0-9_]+)=(.*)/;
    #print "   $var -> $val\n";

    if (exists($par{$var})) {
        print "    varcheck OK: $var=$val"
            if $debug;
        if ("$par{$var}" eq "$val") {
            print " (default)"
                if $debug;
        }
        print "\n"
            if $debug;
    } else {
        warning("varcheck unknown variable: $var=$val");
    }
}
close(P);
```

At this point, the check data is known to be good, and an entry in the `uebungs_checks` database table can be made or updated. If there's a number stored for the check id, it is assumed that there is already a check with that number in the database, and the associated values are checked:

```
### 2. Check & Insert/Update things into DB
if ($check_id =~ /\d+/) {
    # Might be already-existing check, make sure...
    $sth = $dbh->prepare("SELECT check_id, uebung_id, script, ".
                         "       bezeichnung, rechner, parameter ".
                         "FROM uebungs_checks ".
                         "WHERE check_id='$check_id' " .
                         "      AND uebung_id='$uebung_id'");
    $sth->execute();

    $cnt=0;
    while ( @row = $sth->fetchrow_array ) {
        # Check already there, update!
        ( $db_check_id, $db_uebung_id, $db_script, $db_bezeichnung,
          $db_rechner, $db_parameter ) = @row;
```

In that situation, if either the check script name, textual description, lab machine to run the check on or parameters have changed, the record in the database is updated. If all parameters are identical between the database and the exercise text's data, the record in the database is left unchanged:

```
        if ($script ne $db_script
            or $bezeichnung ne $db_bezeichnung
            or $rechner ne $db_rechner
            or $parameter ne $db_parameter) {
```

[1] [The Open Group, 2004] Base Specifications Issue 6: "env - set the environment for command invocation"

```
                    update_db($check_id, $uebung_id, $script,
                              $bezeichnung, $rechner, $parameter);
                    print "check_id $check_id updated\n" if $verbose;
                } else {
                    print "check_id $check_id unchanged\n" if $verbose;
                }

                $cnt++;
```

The procedure `update_db()` is simple, as the check_id is already known, and a simple 'update' SQL command is sufficient:

```
sub update_db() {
    local($check_id, $uebung_id, $script, $bezeichnung, $rechner,
          $parameter) = @_;
    local($sth);

    $parameter =~ s/\\/\\\\/g;
    $parameter =~ s/'/\\'/g;
    $bezeichnung =~ s/\\/\\\\/g;
    $bezeichnung =~ s/'/\\'/g;

    $sql = "UPDATE uebungs_checks ".
           "SET ".
           "    uebung_id='$uebung_id', ".
           "    script='$script', ".
           "    bezeichnung='$bezeichnung', ".
           "    rechner='$rechner', ".
           "    parameter='$parameter' ".
           "WHERE ".
           "    check_id='$check_id'";
    print "    SQL: $sql;\n"
        if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();
}
```

If the database doesn't have a record for the number given in the exercise text of if no number (but XXX) was given instead, then a new record is made for the check:

```
            if ($cnt == 0) {
                # Check not there, insert new!
                $check_id = insert_into_db($uebung_id, $script,
                                           $bezeichnung, $rechner,
                                           $parameter);
                print "check_id $check_id inserted (1)\n" if $verbose;
            }

        } else {
            # Check not there, insert new!
            $check_id = insert_into_db($uebung_id, $script,
                                       $bezeichnung, $rechner,
                                       $parameter);
            print "check_id $check_id inserted (2)\n" if $verbose;
        }
```

Inserting the new record into the `uebungs_checks` table is done by the `insert_into_db()` function, which first inserts the given data into the

database, and then does a query to find out the “check_id” number under which the record was stored, to return it for further processing:

```
sub insert_into_db() {
    local($uebung_id, $script, $bezeichnung, $rechner,
          $parameter) = @_;
    local($sth);

    $parameter =~ s/\\/\\\\/g;
    $parameter =~ s/'/\\'/g;
    $bezeichnung =~ s/\\/\\\\/g;
    $bezeichnung =~ s/'/\\'/g;

    # 1. insert new
    $sql = "INSERT INTO uebungs_checks ".
           "    ( uebung_id, script, bezeichnung, ".
           "      rechner, parameter ) ".
           "VALUES ".
           "    ( '$uebung_id', '$script', '$bezeichnung', ".
           "      '$rechner', '$parameter' )";
    print "    SQL: $sql;\n"
        if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();

    # 2. find $new_check_id
    $sql = "SELECT check_id ".
           "FROM uebungs_checks ".
           "WHERE uebung_id='$uebung_id' ".
           "      AND script='$script' ".
           "      AND bezeichnung='$bezeichnung' ".
           "      AND rechner='$rechner' ".
           "      AND parameter='$parameter'";
    print "    SQL: $sql;\n"
        if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();

    while(@row = $sth->fetchrow_array) {
        $new_check_id = $row[0];
    }
    print "    new check_id=$new_check_id\n"
        if $debug;

    return $new_check_id;
}
```

After collecting data from the input exercise text and (possibly) updating the database, the `main()` routine writes an updated check-call to the output file which has all the check-data from the input exercise text, but the check number (check_id) matches the record in the database’s `uebungs_checks` table.

The list of checks belonging to the exercise are stored in the `%checks_done` hash table. This is used in the `delete_old()` code, where the list is used to find all checks *not* associated with an exercise, and delete them:

```
sub delete_old() {
    $ids=join(", ", sort keys %checks_done );
    $sql="DELETE FROM uebungs_checks ".
         "WHERE check_id NOT IN ( $ids ) ".
```

```
          "      AND uebung_id='$uebung_id'";
    print "    SQL: $sql;\n"
        if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();
    print "old checks removed from database\n"
        if $verbose;
}
```

At this point, the updated exercise text is written in the new file and can be used to replace the original exercise text, as the database also contains the necessary data for the checks to be evaluated and called.

The key items of the lexical processing here are the identification of checks in the input exercise text using perl, and the parsing of the parameter string stored there using a combination of the Bourne shell and the env(1) program to verify that only valid parameters are set. Using the `uebung2db` script, the check data can be stored in compact form near in the exercise text for easy maintenance, and it can be transformed into the database's own storage methods for further processing without a need to deal with each exercise manually.

Improved Web-GUI for Editing Check Scripts: When the checks belonging to an exercise needs to be changed, one way is to edit the exercise text and run `uebung2db` again. To do small changes to single items and to verify that the contents of the database are correct, the web-based user interfaced already present in step I of the Virtual Unix Lab(see figure 7 in section 3.1.1) was extended to use the new check script facilities available in step II. The web GUI now does not only print the check script name, parameters, machine on which to perform the check and a description for the feedback. As the web user interface to the database runs on the same machine where the check scripts are stored, the check scripts can be easily queried to print their overall purpose and a list of parameters they support, as described above. An example display of a check including it's description and allowed parameters can be seen in figure 9 in section 3.1.1.

For this display of the check script's purpose and it's possible parameters, the `public_html/admin/uebung_setup/uebung3_edit.php` script was extended to first determine in which language the script was written. Allowing shell scripts still being written not only in Perl but e.g. as Bourne shell scripts, looking at the script's interpreter stored in the first line would tell how to run the script. The following code fragment has error detection replaced by "`...`" to concentrate on the application code:

```
Aufgabe des ausgewählten Check-Scripts
'<?php print "$script"; ?>':
<p>

<table>
<tr><td class='vulab'>
"<i><?php
        // Interpreter bestimmen
```

```
                $sf=fopen("$checkscriptPath/$script", "r");
                ...
                $s = fgets($sf, 1024);
                fclose($sf);
                if (preg_match("/perl/", $s)) {
                        $interpreter = "perl";
                } else {
                        $interpreter = "sh";
                }
```

After the interpreter is known, it can be run with the check script known to the web user interface to ask for it's purpose by giving the `whatis` command line argument:

```
                // Script laufen lassen, Aufgabe ermitteln
                $cmd="$interpreter $checkscriptPath/$script whatis 2>&1";
                if ($debug) {
                        print "<tr><td colspan='3'>";
                        print "cmd='$cmd'<br>\n";
                        print "</td></tr>\n";
                }
                $pf=popen("$cmd", "r");
                ...
                // Output auslesen und formatiert ausgeben
                while($s = fgets($pf, 1024)) {
                        print "$s";
                }
        ?></i>"</td></tr>
        </table>
```

Giving a buffer size (1024) to `fgets()` here was important, as reading from a pipe opened with `popen()` did not return proper values otherwise. The output of the check script is printed to appear on the web user interface, giving a description of the check script.

Next, the check script is called again with `listparms` as command line argument to retrieve a list of parameters it accepts. The lines returned by the check script have to be split using `|` as a separator, and the three parts – parameter name, default and description – are printed as HTML table then:

```
        Mögliche Parameter:
        <p>

        <table>
        <tr>
                <th class="vulab">Variable</th>
                <th class="vulab">Default</th>
                <th class="vulab">Beschreibung</th>
        </tr>

<?php
        // Script laufen lassen, Parameter ermitteln
        $cmd="$interpreter $checkscriptPath/$script listparms 2>&1";
        if ($debug) {
                print "<tr><td colspan='3'>";
                print "cmd='$cmd'<br>\n";
                print "</td></tr>\n";
        }
        $pf=popen("$cmd", "r");
```

```
        ...

        // Output auslesen und formatiert ausgeben
        while($s = fgets($pf, 1024)) {
                list($var, $def, $bez) = split('\|', $s);

                print "<tr>\n";
                print "<td class='vulab'>$var</td>\n";
                print "<td class='vulab'>$def</td>\n";
                print "<td class='vulab'>$bez</td>\n";
                print "</tr>\n";
        }
        pclose($pf);
?>
        </table>
```

The above example shows how the interface introduced to the check scripts can not only be used to verify the arguments passed to them as described in the description of of the data structure representation frontend with `uebung2db`, but that the same interface can be used to give on-screen help for manual data entry.

The same method can be used to verify that the parameters entered are all allowed, but this was not implemented in step II of the Virtual Unix Lab.

Passing Parameters to Check-Scripts: Section 2.2 describes how check scripts are run on the lab and master machines in step I of the Virtual Unix Lab, which does not include any handling of parameters to the scripts. For step II, passing parameters as environment variables was chosen as an interface, as described in section 3.1.1.

The whole result verification is implemented in the Perl script, which is called with the ID of a booked exercise ("`buchungs_id`").

For the result verification, all the checks and their parameters belonging to be booked exercise identified by the `$buchungs_id` variable are queried from the database and stored in corresponding Perl variables:

```
$sth = $dbh->prepare("SELECT uebungs_checks.check_id AS check_id, ".
                     "       uebungs_checks.script AS script, ".
                     "       uebungs_checks.parameter AS parameter, ".
                     "       uebungs_checks.rechner AS rechner, ".
                     "       uebungs_checks.uebung_id AS uebung_id ".
                     "FROM buchungen, uebungs_checks  ".
                     "WHERE uebungs_checks.uebung_id = buchungen.uebung_id ".
                     "      AND buchungs_id = $buchungs_id ".
                     "ORDER BY check_id");
$sth->execute();
while ( @row = $sth->fetchrow_array) {
        ( $check_id, $script, $parameter, $rechner, $uebung_id ) = @row;
```

With the data for the check known, the call of the check script can be constructed in the `$cmd` variable. As a reminder, in step I of the Virtual Unix Lab the following code was used:

```
                $cmd="cat $checkscript_path/$script ";
                if ($rechner eq 'localhost') {
```

```
        $cmd.="| $interpreter";
} else {
        $cmd.="| $vulabbin_path/rsh-wrapper -p 9999 $rechner $interpreter";
}
$cmd .= " 2>&1";
```

The code in step II looks very similar:

```
$cmd="cat $checkscript_path/$script ";
if ($rechner eq 'localhost')                          $cmd.="| ";
$cmd .= "env ". quotemeta("$parameter"). " $interpreter";
$cmd .= " 2>&1";
```

The difference in step II is that the interpreter is not called directly, but that the env(1) command is used to pass the given string of parameters to the interpreter[1]. That way, only a small change was needed to pass parameters to the check scripts. With the changes to the check scripts themselves described above, implementation of parameters for check scripts was very easy, and the whole result verification architecture of the Virtual Unix Lab gained a lot of flexibility and power.

System Front-End with Special Admin-Only Check-Scripts: Section 3.1.4 describes how check scripts can be used to create a system front-end to the image deployment subsystem of the Virtual Unix Lab. The two check scripts which can be used to update an existing harddisk image are `admin-check-clearharddisk` (see appendix F.2.1) and `admin-check-makeimage` (see appendix F.2.2), and some implementation details should be illustrated here.

`admin-check-clearharddisk`:

This script is used to fill the unused blocks of a harddisk with zero bits, so the harddisk image can be compressed to the minimum size. To not depend on the filesystem structure, the operation consists of two basic operations using the standard filesystem interface[2].

The script needs to be run on the machine that needs to be cloned, running whatever operating system that machine runs at that moment. The current implementation only works for Unix systems, other implementation for Windows systems written in Pascal or an implementation written in perl which will run on any platforms are available[3].

1. Open a file for writing, and fill it with 0-bytes until the disk is full, i.e. all (formerly) unused disk blocks are filled with 0-bytes:
   ```
   echo Cleaning empty blocks...
   dd if=/dev/zero of=0 bs=1048576
   ```
2. After the file has grown to fill up all available space on the disk with 0-bytes, the file can be deleted again using the operating system's calls, assuming only the file's administrative meta-data will

[1] [The Open Group, 2004] Base Specifications Issue 6: "env - set the environment for command invocation"

[2] [Feyrer, 2007] Section "5.10 Reducing the image size"

[3] [Feyrer, 2007] Section "5.10 Reducing the image size"

be adjusted, and that the data blocks will be left untouched on the disk, which allow it to be compressed well when creating the harddisk image:

```
echo Cleaning up...
rm -f 0
```

`admin-check-makeimage`:

The check script to create a new harddisk image from an existing installation on one of the lab machines needs not only create the harddisk image, but do some bookkeeping as well. As such, it needs to be run on the Virtual Unix Lab master machine, and be provided with parameters for which host's harddisk image to take, what harddisk to clone (using NetBSD's name for the disk. e.g. sd0, wd0, etc.) as well as the filename of the harddisk image to create, relative to `/vulab` on the master machine.

The disk image creation process itself is performed by the startup process when the lab machines are booted not from harddisk for an exercise, but from network to perform maintenance like taking a harddisk. To signal the boot process that a harddisk image should be taken, a file `mkimg-$RECHNER` is created on the filesystem that will be used as the filesystem for the lab machine's network filesystem:

```
# Define which image to create
echo ${DISK} ${IMGFILE} >mkimg-${RECHNER}
```

All parameters here - which disk, name of the image file to create and which machine to make the image for are passed to the `admin-check-makeimage` script. Before initiating the netboot to do the actual image-creation, a check is made to see if the lab machine is running properly:

```
if [ "‘$ssh $RECHNER echo READY‘" != READY ]
then
        echo "Machine $RECHNER didn’t respond properly via ’$ssh’"
        echo failed
        exit 0
fi
```

If the lab machine doesn't return the "READY" string, it is assumed to not work properly, and no disk image can be created. Else, the lab machine is commanded to reboot, as the lab machine still runs the operating system modified for an exercise at that time. The reboot contains an option "net" which tells the Sun SPARCstation 4 used as lab machine for step II of the Virtual Unix Lab to not boot from local disk but from network. The reboot command takes special care to not keep any network connections open, and operate in the background without any waiting performed at that moment:

```
# Kick client into netboot
echo "Starting netboot on $RECHNER in background..."
# Path for Solaris is /usr/sbin/reboot,
# redirection is shell-dependent !!!
$ssh $RECHNER "env PATH=/usr/sbin:/sbin /bin/sh -c ’reboot -- net’ \
                </dev/null 2>/dev/null >/dev/null" \
               </dev/null 2>/dev/null >/dev/null &
echo "done. (rc=$?)"
```

After initiating the netboot, a pause must be made, in which the lab system is reset, netbooted, and starting to create the system. This process is monitored via the `$client_log` log file, which is created by the netboot system's image creation facility and tagged with strings indicating that the image-process has started ("`Starting ...`") and ended (`$deployment_done_cookie`).

```
# Wait for client to startup on netboot properly
echo "Waiting a bit to get to /etc/rc..."
sleep 120 # takes about 70 seconds, plus some extra
if ! grep ^Starting $client_log >/dev/null 2>/dev/null
then
        echo "Client $RECHNER didn't do netboot properly, aborting."
        exit 1
else
        echo "$RECHNER properly netbooted."
fi

# Client's running, now wait for it to be done
while ! grep -q "$deployment_done_cookie" $client_log
do
        echo 'date': waiting for $RECHNER to finish: 'tail -1 $client_log'
        sleep $deployment_poll_interval
done
echo done.
```

At that point, the lab machine shuts down the operating system, performs a hardware reset and starts booting up again, but with a network filesystem instead of the local disk as a source for its operating system. The netboot environment can be found in the file `/vulab/etc/rc`[1], and for the image creation process, it checks if a file with the name `mkimg-`*`machine`* is present for the machine, and starts creating a disk image using the `disk2img` command with the parameters from the `mkimg`-file:

```
machine='uname -n'
...
echo Starting deployment: 'date' | log
...
if [ -f mkimg-${machine} ]
then
# For "unix-check-makeimage" check
#
echo starting disk2img 'cat mkimg-${machine}' | log
countdown 5
./disk2img 'cat mkimg-${machine}'
        ...
fi
...

echo Deployment done: 'date' | log
echo Rebooting... | log

echo $deployment_done_cookie | log

reboot
```

The file written to by the "`log`" command here is the same file that the `admin-check-makeimage` check script is reading to monitor

[1] [Feyrer, 2004]

the progress of the image creation process.

After the `$deployment_done_cookie` string is written in the log file, the check script running on the master machine will continue it's operation by first checking again that the lab machine survived the reboot (using the same method shown above), and then notes that the new image file is now available in the `image` database table:

```
# Image in Tabelle 'images' eintragen:
echo -n Remember image $IMGFILE in database:
echo "INSERT INTO images (bezeichnung) VALUES ('"$IMGFILE"');" \
| psql -U vulab
```

Finally, the lab machine will be ready for new exercises again, and the new harddisk image can be used to define new exercises created after the "admin" exercise.

By using the combination of these two check scripts as system front-end, updating a harddisk image is much easier than performing the necessary steps manually, as no knowledge about internals is necessary.

Ensuring that a perl binary is available: Not every operating system ships with perl in the default operation these days. Some (like Solaris and many Linux distributions, but not all) do, others (like NetBSD, Windows) don't. As check scripts are written in perl, they require a perl interpreter to be available. For systems that don't have perl installed by default, a perl binary can be put into a place like `/root/vulab/perl`, but special care must still be taken to run it, if no other binary is available. Other systems that do ship with perl may have it in various places – `/usr/bin/perl`, `/usr/local/bin/perl` and `/usr/pkg/bin/perl` are only a few possibilities which need to be taken into account.

One way to find the perl binary is to make the result verification script `uebung_auswerten` look for the binary before sending the check script to the lab machine. A better solution is to use the shell's feature of looking for a given binary in a number of places using the `PATH` environment variable with a fallback like `/root/vulab/perl` provided on all systems for safety, which was added to step II of the Virtual Unix Lab:

```
if ($i =~ /perl/) {
# perl may not be installed system-wide!
$interpreter="PATH='\${PATH}':/root/vulab perl";
```

Using this mechanism, it was possible to use the perl binary that comes with Solaris 9. This approach saves some disc space, which is always tight on the clients, and it is one item less to remember when setting up a new system.

Printing feedback: Depending on the context, the PHP functions used in the exercise text files (see section 3.1.3) either do

- not print feedback at all (i.e. before and during exercises),
- print feedback for the result a single user made (i.e. when a user requests feedback after taking a particular exercise), or

- print results how all users performed on a particular exercise (when an administrator requests feedback on a particular exercise).

The PHP functions for giving feedback are defined in the `public_html/-functions-uebung.php` file, which is pulled in by the various places which display exercise texts with or without feedback:

- Displaying only the exercise text with no feedback data printed is done by `public_html/user/uebungen/show.php`
- For the exercise, the time remaining, "finished"-button and exercise text are printed by `public_html/user/ueben/start.php`
- Feedback for single users is printed in `public_html/user/buchungen/auswertung.php`
- Admin-feedback for all users is printed in `public_html/admin/buchungen/auswertung.php`

Each of these files pulls in the exercise text file, which in turn calls the two functions `auswertung_ueberschrift()` and `auswertung_zusammenfassung()` to print headers and footers, and `auswertung_teiluebungen()` to do the main work of (not) printing feedback.

The decision if feedback should be printed is determined by the file pulling in the `functions-uebung.php` and exercise text file by setting the variable `$auswertung` if feedback (for either single or all users) should be printed. The functions `auswertung_ueberschrift()` and `auswertung_zusammenfassung()` check this variable, and do nothing if it is not set:

```
function auswertung_ueberschrift () {
        global $auswertung;
        if (! $auswertung) return;
        ...
}

function auswertung_zusammenfassung() {
        global $auswertung;
        if (! $auswertung) return;
        ...
}
```

The former function actually only initializes some variables for keeping statistics, and leaves printing the header with exercise statistics like date, duration etc. to the `public_html/user/buchungen/auswertung.php`.

The main work of printing feedback is done by `auswertung_teiluebungen()` function, which in turn calls `teiluebung_print()` for each of the check numbers, stored in `$check_id`, it's called:

```
        $nargs = func_num_args();
        for ($i=0; $i < $nargs; $i++) {
                $check_id = func_get_arg($i);
                teiluebung_print($check_id);
        }
```

The `teiluebung_print()` function does the main work of giving feedback. It first retrieves the textual description of the given check number, and stores it in the `$bezeichnung` (“description”) variable:

```
$sql = "SELECT bezeichnung ".
       "FROM uebungs_checks ".
       "WHERE check_id='$check_id' ".
       "        AND uebung_id='$uebung_id'";
$msg = "Beim Zugriff auf die Datenbank ist ein Fehler ".
       "aufgetreten.<br>".
       "Wenden Sie sich an Ihren Administrator.";
$result = pg_exec($connection, $sql);
if ($result) {
        $array = pg_fetch_array($result, 0);
        if ($array) {
                $bezeichnung = $array['bezeichnung'];
        } else {
                $bezeichnung = "";
        }
} else {
        $bezeichnung = "N/A";
}
```

Then, the result for the particular check number (stored in `$check_id`) of a previously booked exercise (stored in `$buchungs_id`) is retrieved to see if the task was completed successfully or not, and stored in `$erfolg` (“success”):

```
$sql="SELECT erfolg ".
     "FROM ergebnis_checks ".
     "WHERE buchungs_id='$buchungs_id' ".
     "  AND check_id='$check_id'";
$result = pg_exec($connection, $sql);
if ($result) {
        $array = pg_fetch_array($result, 0);
        if ($array) {
                $erfolg = $array['erfolg'];
        } else {
                $erfolg = "notfound";
        }
} else {
        $erfolg = "notfound";
}
```

After the textual description and the result of the corresponding exercise is known, the result is printed:

```
print "<tr>\n";
print "<td width='95%'><font color='darkgreen'>$bezeichnung</font></td>\n";
print "<td>$erfolg</td>\n";
```

Feedback is printed this way for every part of an exercise. If the variable `$auswertung` is set, this results in feedback as displayed in figure 12, if the variable is not set the result from the right side of figure 5 is shown.

If feedback is requested by a user logged into the Virtual Unix Lab with administrator privileges, more information is printed as discussed in section 3.1.3 and shown in figure 13.

The admin feedback is printed by `public_html/admin/buchungen/auswertung.php`, which pulls in the code for printing feedback from `public_html/user/buchungen/auswertung.php`:

```
% cat public_html/admin/buchungen/auswertung.php
<?php
include "$absoluteProjectPath/user/buchungen/auswertung.php";
?>
```

That way, the code is kept in a single place. It is still possible to determine if the access was from a user or administrator, as the `$menu` PHP variable will be set accordingly, which is used to determine if admin feedback should be printed:

```
        global $menu;
        if ($menu == 'admin') {
```

The admin feedback first determines how many users completed the task in question (identified by the exercise type in `$uebung_id` and the check number in `$check_id`) successfully (`$ok`), and how many did not (`$nok`):

```
$sql="SELECT count(*) ".
     "FROM ergebnis_checks, buchungen ".
     "WHERE ergebnis_checks.buchungs_id=buchungen.buchungs_id ".
     "      AND buchungen.uebung_id='". $uebung_id ."' ".
     "      AND check_id='". $check_id ."' ".
     "      AND erfolg='t'";
$msg = "Beim Zugriff auf die Datenbank ist ein Fehler ".
       "aufgetreten.<br>".
       "Wenden Sie sich an Ihren Administrator!<br> (sql=$sql)";
$aresult = executeStatement($connection, $sql, $msg);
$aarray = fetchArray($aresult, 0);
$ok=$aarray['count'];

$sql="SELECT count(*) ".
     "FROM ergebnis_checks, buchungen ".
     "WHERE ergebnis_checks.buchungs_id=buchungen.buchungs_id ".
     "      AND buchungen.uebung_id='". $uebung_id ."' ".
     "      AND check_id='". $check_id ."' ".
     "      AND erfolg='f'";
$msg = "Beim Zugriff auf die Datenbank ist ein Fehler ".
       "aufgetreten.<br>".
       "Wenden Sie sich an Ihren Administrator!<br> (sql=$sql)";
$aresult = executeStatement($connection, $sql, $msg);
$aarray = fetchArray($aresult, 0);
$nok=$aarray['count'];
```

The total number of students participating in an exercise is obvious:

```
$sum = $ok + $nok;
```

For each number – successful completions `$ok`, unsuccessful completions `$nok` and total number of participants `$sum` – a line with absolute and relative number as well as a bar of `o`s is printed:

```
$c = $ok; $t = 'Bestanden';
  ...
$c = $nok; $t = 'Nicht bestanden';
  ...
$c = $sum; $t = 'Summe';
  ...
```

The code to print the number `$c` and text of the line `$t` is the same for each line, it prints the text, absolute number and percentage, followed by as many `o`s as students were in the counter `$c`:

```
print "<tr><td bgcolor='". $bgcol ."'> </td>\n";
print "    <td><font color='darkgreen'>$t:</font></td>\n";
print "    <td align='right'><font color='darkgreen'>". $c ."</font></td>\n";
print "    <td align='right'><font color='darkgreen'>".
sprintf("(%d%%)", 100*$c/$sum) .
"</font></td>\n";
print "    <td><font color='darkgreen'>|";
for($i=0; $i<$c; $i++) print "o";
print     "</font></td>\n";
print "</td></tr>";
```

This gives three lines of feedback containing details on how all users taking a particular exercise performed, which can then be compared against the individual result also printed. A comparison for all the tasks of an exercise is possible as the feedback is given for each of the tasks of an exercise.

3.3 Summary of Step II

Comparing the improvements intended for step II of the Virtual Unix Lab and the changes made, the conclusion can be drawn that the goals were met within the given requirements. Step II of the Virtual Unix Lab as described here was realized and used as a base for evaluation. During the implementation and evaluation of step II, a number of possible improvements were identified, which can be addressed in future implementation steps of the Virtual Unix Lab and which will be listed in the conclusions drawn on result verification of exercise results using Domain Specific Languages.

4 Summary

This paper has focused on the two iterative steps of the Virtual Unix Lab design and implementation. The basic design in step I was extended in step II, improving verification, definition of the exercise text, and tuning of the result verification architecture. As a side effect, a subsystem was created that uses the result verficication subsystem for updating harddisk image files for new exercise setups.

Specific emphasis was given on the implementation details in this paper. The related steps of creating a domain specific language – the Verification Unit Domain Specific Language (VUDSL) – along with the related works on the processing and management infrastructure can be found in [Feyrer, 2008].

A Exercise texts for users

The exercise texts displayed in this section are the plain text given to the user for practicing. They were the same for step I and II of the Virtual Unix Lab, and were rendered from HTML into plain text using "`lynx -dump`".

A.1 Network Information System (NIS) exercise

The following text displays the NIS exercise's text:

```
                      Übung: NIS Master und Client Setup

   In dieser Übung soll auf den beiden vulab-Rechner der Network
   Information Service (NIS) installiert werden. Dabei wird auf dem
   Rechner "vulab1" der NIS-Master, auf dem Rechner "vulab2" der
   NIS-Client installiert.

1. Master (Solaris): vulab1

     * Stellen Sie sicher dass die nötigen Pakete (SUNWypr, SUNWypu,
       SUNWsprot, ...) installiert sind.
     * Setzen Sie den NIS-Domänenname auf "vulab" (/etc/defaultdomain &
       domainname(1))
     * Setzen Sie den Rechner mit "ypinit -m" als NIS Master auf
     * Sorgen Sie dafür dass die nötigen Serverprozesse (ypbind, ypserv,
       ...) beim booten gestartet werden.
     * Starten Sie die Serverdienste!
     * Welcher NIS-Server wird verwendet?
     * Welche Datei wird für die Gruppen-Daten verwendet?
     * Welche Datei wird für die Passwort-Daten verwendet?
     * Überprüfen Sie ob Gruppen- und Passwort-Informationen über NIS
       abgefragt werden können.
     * Vergleichen Sie den Passwort-Eintrag des Benutzers "vulab" im NIS
       und in den /etc-Dateien. Was stellen Sie fest?
     * Sorgen Sie dafür, dass die Passwort-Informationen künftig in der
       Datei /var/yp/passwd gehalten werden. Die existierenden Logins
       sollen dabei nicht übernommen werden.
     * Legen Sie im NIS eine Kennung "ypuser" mit eindeutiger UID,
       Home-Verzeichnis "/usr/homes/ypuser", Korn-Shell als Login-Shell,
       und Passwort "ypuser" an.
     * Stellen Sie sicher dass der User "ypuser" via finger(1) sichtbar
       ist
     * Stellen Sie sicher dass sich der User "ypuser" via telnet, ssh und
       ftp einloggen kann!
     * Stellen Sie sicher, dass der User "ypuser" sein Passwort mit
       yppasswd(1) ändern kann.

2. Client (NetBSD): vulab2

     * Setzen Sie den Domainnamen auf den selben Namen wie beim
       NIS-Master oben.
     * Ist das aufsetzen des Clients mit "ypinit -c" nötig? Ist es
       sinnvoll? Warum (nicht)?
     * Stellen Sie sicher dass die nötigen Dienste (ypbind, ...) beim
       booten gestartet werden.
     * Starten Sie die Dienste!
```

```
     * Welcher NIS-Server wird verwendet?
     * Stellen Sie sicher dass die NIS Maps (group, hosts, ...) abgerufen
       werden können
     * Stellen  Sie  sicher dass die NIS-Benutzer mit finger(1) abgefragt
       werden können
     * Stellen  Sie sicher dass sich der oben angelegte Benutzer "ypuser"
       auf  dem Client einloggen kann. Erstellen Sie das Home-Verzeichnis
       dazu vorerst manuell.
     * Betrachten  Sie  das  Passwort-Feld  der  Passwort-Datei des Users
       "ypuser" auf dem NIS Master!.
     * Ändern  Sie  das  Passwort  von "ypuser" vom Client aus im NIS auf
       ‘‘myn1spw’’.
     * Betrachten  Sie  das  Passwort-Feld  der  Passwort-Datei des Users
       "ypuser" auf dem NIS Master erneut. Was stellen Sie fest?

3. Diverses

     * Setzen  Sie  den  "Full  Name"  des  Benutzers  "ypuser"  auf "NIS
       Testbenutzer". Verifizieren Sie das Ergebnis mit finger(1). Welche
       Methoden  zum setzen existieren auf dem NIS Master? Welche auf dem
       NIS Client?
     * Legen  Sie eine NIS-Gruppe "benutzer" an, und machen Sie diese zur
       (primären)  Gruppe  des Benutzers "ypuser". Welche Group-ID wählen
       Sie? Warum?
     * Legen  Sie  im  Home-Verzeichnis  des  Benutzers  "ypuser" auf dem
       Master  und  dem Client eine Datei an, und überprüfen Sie, welcher
       Gruppe sie gehört.
     * Sorgen  Sie dafür dass der Benutzer "ypuser" auf dem NetBSD-System
       mittels  su(1)  root-Rechte  erhalten  kann.  Er  muss dazu (unter
       NetBSD) zusätzlich Mitglied der Gruppe "wheel" sein.
     * Wie bewerten Sie die Tatsache dass das root-Passwort alleine nicht
       reicht,    sondern    auch   die   richtige   Gruppenzugehörigkeit
       Voraussetzung  für  einen  su(1)  auf  root  ist?  Vergleichen Sie
       zwischen NetBSD, Solaris und Linux!
     * Der  Rechner "tab" (IP-Nummer: 194.95.108.32) soll via NIS bekannt
       gemacht  werden.  Tragen  Sie  den  Rechner  auf dem Server in die
       entsprechende  Hosts-Datei  ein, aktualisieren Sie die NIS-Map und
       verifizieren  Sie das Ergebnis mittels ypcat(1) und ping(1) sowohl
       auf  dem  NIS-Master  als  auch  auf  dem  NIS-Client.

Hinweise:

     * Solaris-Pakete  für  bash  und tcsh liegen in /cdrom, Installation
       mit pkgadd(1).
     * NetBSD-Pakete   für   bash  und  tcsh  (und  weitere)  liegen  auf
       ftp://ftp.de.netbsd.org/pub/NetBSD/packages/1.6.1/sparc/All,
       Installation mit pkg_add(1).
```

A.2 Network File System (NFS) exercise

The following text displays the NFS exercises’s text:

```
                        Übung: NFS Server und Client Setup

   In  dieser  Übung  soll  auf den beiden vulab-Rechner das Network File
   System  (NFS)  installiert werden. Dabei wird auf dem Rechner "vulab1"
   der NFS-Server, auf dem Rechner "vulab2" der NFS-Client installiert.

1. Server (Solaris): vulab1
```

Das Dateisystem /usr/homes soll für den zweiten Rechner ’vulab2’ per NFS exportiert werden:

* Sichern Sie die Datei, in der bisher die NFS-Exports notiert sind
* Das Verzeichnis /usr/homes soll für den Rechner "vulab2" freigegeben werden. Tragen Sie dies in die richtige Datei ein.
* Laufen die nötigen Serverprozesse? Starten Sie sie ggf. mit Hilfe der passenden Start-Scripten aus /etc/*.d.
* Sorgen Sie dafür dass die Datei (neu) eingelesen wird
* Überprüfen Sie mit ’showmount -e’ ob die Freigabe besteht!

2. Client (NetBSD): vulab2

Das Verzeichnis /usr/homes soll vom NFS-Server (vulab1) auf /usr/homes gemountet werden:

* Existiert der Mountpoint /usr/homes auf dem Client?
* Sind Daten im Mountpoint enthalten?
* Überprüfen Sie mit ’showmount -e’ die NFS-Freigaben des NFS-Servers ’vulab1’ (10.0.0.1)
* Untersuchen Sie die System-Defaults in /etc/defaults/rc.conf und tragen Sie für NFS nötige Abweichungen in die Datei /etc/rc.conf ein. Achten Sie auf rpc.lockd(8) und rpc.statd(8)!
* Starten Sie alle nötigen Hintergrundprozesse.
* Überprüfen Sie, ob das Verzeichnis /usr/homes von vulab1 testweise auf /mnt gemountet werden kann. Unmounten Sie es anschliessend wieder!
* Sorgen Sie dafür daß das Verzeichnis /usr/homes vom NFS-Server "vulab1" beim Systemstart auf /usr/homes gemountet wird, tragen Sie dies in die passenden Konfigurationsdatei ein
* Mounten Sie alle noch nicht gemounteten NFS-Verzeichnisse!
* Überprüfen Sie mit df(1) und mount(8) daß das Verzeichnis gemountet ist!

3. Zugriffsrechte

3.1 Rechnerbasiert

* Legen Sie als root auf dem NIS-Client ein Verzeichnis /usr/homes/nfsuser an! Wie reagiert das System, und warum?
* Lesen Sie auf dem NFS-Server die Manpage zu dfstab(4) und den darin unter "SEE ALSO" verwiesenen Befehlen (etc.), und sorgen Sie dafür, daß Sie als root auf dem NFS-Client vollen Zugriff habe
* Machen Sie die nötige Änderung in /etc/dfs/dfstab.
* Lesen Sie die Datei neu ein!
* Welche Sicherheitsimplikationen hat der eben vorgenommene Konfigurationsschritt? Macht er in der Praxis Sinn? Wie kann man ihn umgehen?
* Legen Sie das Verzeichnis /usr/homes/nfsuser an!

3.2 Benutzerbasiert

Es soll ein Benutzer "nfsuser" auf beiden Systemen angelegt werden, der auf jedem System lokal vermerkt ist (Login, Passwort etc. in /etc/...), das Home-Verzeichnis /usr/homes/nfsuser soll aber auf beiden Rechnern mittels NFS verfügbar sein!

* Legen Sie auf vulab1 den User an: ‘‘useradd -d /usr/homes/nfsuser nfsuser’’
* Legen Sie auf vulab2 denselben User an: ‘‘useradd -d /usr/homes/nfsuser nfsuser’’
* Geben Sie dem Benutzer auf beiden Systemen (getrennt) mittels passwd(1) ein Passwort
* Geben Sie das Verzeichnis /usr/homes/nfsuser mittels chown(1) dem Benutzer "nfsuser".

```
     * Loggen  Sie  sich  auf  beiden  Rechner als User "nfsuser" ein und
       legen  Sie  eine  Datei "hallo-von-vulab1" bzw. "hallo-von-vulab2"
       an.
     * Welches Problem besteht?
     * Geben  Sie  auf beiden Rechnern dem Benutzer "nfsuser" die User-ID
       2000,  stellen Sie sicher dass das Home-Verzeichnis (inkl. Inhalt)
       auch dem User gehört, und legen Sie die beiden Dateien erneut an.

Hinweise:

     * Solaris-Pakete  für  bash  und tcsh liegen in /cdrom, Installation
       mit pkgadd(1).
     * NetBSD-Pakete   für   bash  und  tcsh  (und  weitere)  liegen  auf
       ftp://ftp.de.netbsd.org/pub/NetBSD/packages/1.6.1/sparc/All,
       Installation mit pkg_add(1).
```

B Exercises including text and check data

The exercise texts displayed in this section are from step II of the Virtual Unix Lab. They contain the exercise text as well as data for the checks to be run.

B.1 Network Information System (NIS) exercise

```
<!-- DB updated by feyrer on Sun Feb 22 23:53:01 MET 2004 from nis.php -->
<!-- Id: nis.php,v 1.23 2004/06/03 10:27:12 feyrer Exp  -->
<?php auswertung_ueberschrift(); ?>
<!-- ------------------------------------------------------------------- -->

<h1> NIS Master und Client Setup</h1>

In dieser Übung soll auf den beiden vulab-Rechner der Network
Information Service (NIS) installiert werden. Dabei wird auf dem
Rechner "vulab1" der NIS-Master, auf dem Rechner "vulab2" der
NIS-Client installiert.
<p>

<h2>1. Master (Solaris): vulab1</h2>

<ul>
<li> Stellen Sie sicher dass die nötigen Pakete (SUNWypr, SUNWypu,
     SUNWsprot, ...) installiert sind.
<li> Setzen Sie den NIS-Domänenname auf "vulab" (/etc/defaultdomain &
     domainname(1))

     <?php auswertung_teiluebungen(
                 774, // vulab1: check-file-contents FILE=/etc/defaultdomain CONTENT_SHOULD='vulab'
                      //         Domäne in /etc/defaultdomain gesetzt?

                 775  // vulab1: check-program-output PROGRAM=domainname OUTPUT_SHOULD='vulab'
                      //         Domäne im laufenden System (domainname(1)) gesetzt?
        ); ?>

<li> Setzen Sie den Rechner mit "ypinit -m" als NIS Master auf

     <?php auswertung_teiluebungen(
                 776, // vulab1: check-file-exists FILE=/var/yp/Makefile
                      //         Existiert /var/yp/Makefile?

                 777, // vulab1: check-file-exists FILE=/var/yp/binding/vulab/ypservers
                      //         Existiert /var/yp/binding/vulab/ypservers?

                 778  // vulab1: check-file-exists FILE=/var/yp/passwd.time
                      //         Existiert /var/yp/passwd.time?

        ); ?>

<li> Sorgen Sie dafür dass die nötigen Serverprozesse (ypbind, ypserv,
```

```
     ...) beim booten gestartet werden.
<li> Starten Sie die Serverdienste!
<li> Welcher NIS-Server wird verwendet?

     <?php auswertung_teiluebungen(
                 779  // vulab1: check-program-output PROGRAM=ypwhich OUTPUT_SHOULD='vulab1'
                      //         Gibt ypwhich(1) 'vulab1' zurück?

        ); ?>

<li> Welche Datei wird für die Gruppen-Daten verwendet?
<li> Welche Datei wird für die Passwort-Daten verwendet?
<li> Überprüfen Sie ob Gruppen- und Passwort-Informationen über NIS
     abgefragt werden können.

     <?php auswertung_teiluebungen(
                 780, // vulab1: check-program-output PROGRAM='ypcat passwd | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in passwd-Map vorhanden?

                 781, // vulab1: check-program-output PROGRAM='ypcat hosts  | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in host-Map vorhanden?

                 782  // vulab1: check-program-output PROGRAM='ypcat group  | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in group-Map vorhanden?

        ); ?>

<li> Vergleichen Sie den Passwort-Eintrag des Benutzers "vulab" im NIS
     und in den /etc-Dateien. Was stellen Sie fest?
<li> Sorgen Sie dafür, dass die Passwort-Informationen künftig in der
     Datei /var/yp/passwd gehalten werden. Die existierenden Logins
     sollen dabei nicht übernommen werden.

     <?php auswertung_teiluebungen(
                 783, // vulab1: check-file-contents FILE=/var/yp/Makefile CONTENT_SHOULD='^PWDIR.*=.*/var/yp'
                      //         PWDIR in /var/yp/Makefile auf /var/yp gesetzt?

                 784  // vulab1: check-file-exists FILE=/var/yp/passwd
                      //         Existiert /var/yp/passwd?

        ); ?>

<li> Legen Sie im NIS eine Kennung "ypuser" mit eindeutiger UID,
     Home-Verzeichnis "/usr/homes/ypuser", Korn-Shell als Login-Shell,
     und Passwort "ypuser" an.


     <?php auswertung_teiluebungen(
                 785, // vulab1: check-directory-exists DIR=/usr/homes/ypuser
                      //         Verzeichnis /usr/homes/ypuser existiert?

                 786, // vulab1: unix-check-user-shell LOGIN=ypuser SHELL_SHOULD="/.*/ksh"
                      //         Shell von ypuser auf ksh gesetzt?

                 787, // vulab1: check-program-output PROGRAM='cat /var/yp/passwd | grep ypuser: | wc -l' OUTPUT_SHOULD=1
                      //         User ypuser in /var/yp/passwd eingetragen?

                 788  // vulab1: check-program-output PROGRAM='ypcat passwd       | grep ypuser: | wc -l' OUTPUT_SHOULD=1
                      //         User ypuser in passwd NIS Map vorhanden?

        ); ?>


<li> Stellen Sie sicher dass der User "ypuser" via finger(1) sichtbar
     ist

     <?php auswertung_teiluebungen(
                 789, // vulab1: unix-check-user-exists LOGIN=ypuser
                      //         User existiert (getpwnam(3))?

                 790, // vulab1: check-file-contents FILE=/etc/nsswitch.conf CONTENT_SHOULD='passwd:.*nis'
                      //         passwd-Information wird in NIS gesucht (/etc/nsswitch.conf)?

                 791, // vulab1: check-file-contents FILE=/etc/nsswitch.conf CONTENT_SHOULD='group:.*nis'
                      //         group-Information wird in NIS gesucht (/etc/nsswitch.conf)?

                 792  // vulab1: check-file-contents FILE=/etc/nsswitch.conf CONTENT_SHOULD='hosts:.*nis'
                      //         hosts-Information wird in NIS gesucht (/etc/nsswitch.conf)?

        ); ?>

<li> Stellen Sie sicher dass sich der User "ypuser" via telnet, ssh und
     ftp einloggen kann!
<li> Stellen Sie sicher, dass der User "ypuser" sein Passwort mit
     yppasswd(1) ändern kann.
</ul>
```

```
<h2>2. Client (NetBSD): vulab2</h2>

<ul>
<li> Setzen Sie den Domainnamen auf den selben Namen wie beim NIS-Master
     oben.

     <?php auswertung_teiluebungen(
                 793, // vulab2: check-file-contents FILE=/etc/defaultdomain CONTENT_SHOULD='vulab'
                      //         Domainname in /etc/defaultdomain gesetzt?

                 794  // vulab2: check-program-output PROGRAM=domainname OUTPUT_SHOULD='vulab'
                      //         Domainname im laufenden System gesetzt? (domainname(1))

        ); ?>

<li> Ist das aufsetzen des Clients mit "ypinit -c" nötig? Ist es
     sinnvoll? Warum (nicht)?
<li> Stellen Sie sicher dass die nötigen Dienste (ypbind, ...) beim
     booten gestartet werden.

     <?php auswertung_teiluebungen(
                 795, // vulab2: netbsd-check-rcvar-set RCVAR=rc_configured
                      //         /etc/rc.conf: rc_configured gesetzt?

                 796, // vulab2: netbsd-check-rcvar-set RCVAR=rpcbind
                      //         /etc/rc.conf: rpcbind gesetzt?

                 797  // vulab2: netbsd-check-rcvar-set RCVAR=ypbind
                      //         /etc/rc.conf: ypbind gesetzt?

        ); ?>

<li> Starten Sie die Dienste!

     <?php auswertung_teiluebungen(
                 798, // vulab2: unix-check-process-running PROCESS=rpcbind
                      //         rpcbind läuft?

                 799  // vulab2: unix-check-process-running PROCESS=ypbind
                      //         ypbind läuft?

        ); ?>

<li> Welcher NIS-Server wird verwendet?

     <?php auswertung_teiluebungen(
                 800  // vulab2: check-program-output PROGRAM=ypwhich OUTPUT_SHOULD='vulab1'
                      //         Wird vulab1 als NIS-Server verwendet? (ypwhich(1))

        ); ?>

<li> Stellen Sie sicher dass die NIS Maps (group, hosts, ...) abgerufen
     werden können

     <?php auswertung_teiluebungen(
                 801, // vulab2: check-program-output PROGRAM='ypcat passwd | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in passwd-Map vorhanden?

                 802, // vulab2: check-program-output PROGRAM='ypcat hosts  | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in hosts-Map vorhanden?

                 803  // vulab2: check-program-output PROGRAM='ypcat group  | wc -l' OUTPUT_SHOULD='[^0]*'
                      //         Daten in group-Map vorhanden?

        ); ?>

<li> Stellen Sie sicher dass die NIS-Benutzer mit finger(1) abgefragt
     werden können

     <?php auswertung_teiluebungen(
                 804  // vulab2: unix-check-user-exists LOGIN=ypuser
                      //         Existiert Benutzer ypuser?

        ); ?>

<li> Stellen Sie sicher dass sich der oben angelegte Benutzer "ypuser"
     auf dem Client einloggen kann. Erstellen Sie das Home-Verzeichnis
     dazu vorerst manuell.

     <?php auswertung_teiluebungen(
                 805  // vulab2: check-directory-exists DIR=/usr/homes/ypuser
                      //         Existiert Home-Verzeichnis?

        ); ?>

<li> Betrachten Sie das Passwort-Feld der Passwort-Datei des Users
     "ypuser" auf dem NIS Master!.
<li> Ändern Sie das Passwort von "ypuser" vom Client aus im NIS auf
```

```
     ‘‘myn1spw’’.

     <?php auswertung_teiluebungen(
                 806  // vulab2: unix-check-user-password LOGIN=ypuser PASSWD_SHOULD=myn1spw
                      //         Paßwort richtig gesetzt?

        ); ?>

<li> Betrachten Sie das Passwort-Feld der Passwort-Datei des Users
     "ypuser" auf dem NIS Master erneut. Was stellen Sie fest?
</ul>


<h2>3. Diverses</h2>

<ul>
<li> Setzen Sie den "Full Name" des Benutzers "ypuser" auf "NIS
     Testbenutzer". Verifizieren Sie das Ergebnis mit finger(1).
     Welche Methoden zum setzen existieren auf dem NIS Master? Welche
     auf dem NIS Client?

     <?php auswertung_teiluebungen(
                 807  // vulab2: unix-check-user-fullname LOGIN=ypuser FULLNAME_SHOULD=’NIS Testbenutzer’
                      //         Fullname richtig gesetzt?

        ); ?>

<li> Legen Sie eine NIS-Gruppe "benutzer" an, und machen Sie diese zur
     (primären) Gruppe des Benutzers "ypuser". Welche Group-ID wählen
     Sie? Warum?

     <?php auswertung_teiluebungen(
                 808, // vulab2: unix-check-user-ingroup LOGIN=ypuser GROUP_SHOULD=benutzer
                      //         Benutzer ’ypuser’ Mitglied der Gruppe ’benutzer’?

                 809  // vulab2: check-program-output PROGRAM=’ypcat group’ OUTPUT_SHOULD=’benutzer:’
                      //         Gruppe ’benutzer’ existiert in der group NIS-Map?

        ); ?>

<li> Legen Sie im Home-Verzeichnis des Benutzers "ypuser" auf dem
     Master und dem Client eine Datei an, und überprüfen Sie, welcher
     Gruppe sie gehört.
<li> Sorgen Sie dafür dass der Benutzer "ypuser" auf dem NetBSD-System
     mittels su(1) root-Rechte erhalten kann. Er muss dazu (unter
     NetBSD) zusätzlich Mitglied der Gruppe "wheel" sein.

     <?php auswertung_teiluebungen(
                 810  // vulab2: check-file-contents FILE=/etc/group CONTENT_SHOULD=’"^wheel:.*ypuser"’
                      //         ypuser in wheel-Gruppe in /etc/group?

        ); ?>

<li> Wie bewerten Sie die Tatsache dass das root-Passwort alleine nicht
     reicht, sondern auch die richtige Gruppenzugehörigkeit
     Voraussetzung für einen su(1) auf root ist? Vergleichen Sie
     zwischen NetBSD, Solaris und Linux!
<li> Der Rechner "tab" (IP-Nummer: 194.95.108.32) soll via NIS bekannt
     gemacht werden. Tragen Sie den Rechner auf dem Server in die
     entsprechende Hosts-Datei ein, aktualisieren Sie die
     NIS-Map und verifizieren Sie das Ergebnis mittels ypcat(1)
     und ping(1) sowohl auf dem NIS-Master als auch auf dem NIS-Client.

     <?php auswertung_teiluebungen(
                 811, // vulab2: check-program-output PROGRAM=’ypcat hosts’ OUTPUT_SHOULD=’194.95.108.65.*tab’
                      //         Eintrag mit IP-Nummer und Rechnername in hosts NIS-Map?

                 812  // vulab2: check-program-output PROGRAM=’/sbin/ping -c 1 tab 2>&1 ; echo result:$?’ OUTPUT_SHOULD=’^result:0$’
                      //         ’tab’ pingbar?

        ); ?>

</ul>


<h2>Hinweise:</h2>

<ul>
<li> Solaris-Pakete für bash und tcsh liegen in /cdrom, Installation
     mit pkgadd(1).

     <?php auswertung_teiluebungen(
                 898, // vulab1: solaris-check-installed-pkg PKG=SUNWtcsh
                      //         tcsh auf Solaris installiert? (pkginfo SUNWtcsh)

                 899  // vulab1: solaris-check-installed-pkg PKG=SUNWbash
                      //         bash auf Solaris installiert? (pkginfo SUNWbash)
        ); ?>
```

```
<li> NetBSD-Pakete für bash und tcsh (und weitere) liegen auf
     ftp://ftp.de.netbsd.org/pub/NetBSD/packages/1.6.1/sparc/All,
     Installation mit pkg_add(1).

     <?php auswertung_teiluebungen(
                 900, // vulab2: netbsd-check-installed-pkg PKG=tcsh
                      //         tcsh auf NetBSD installiert? (pkg_info -e tcsh)

                 901  // vulab2: netbsd-check-installed-pkg PKG=bash
                      //         bash auf NetBSD installiert? (pkg_info -e bash)
        ); ?>

</ul>

<!-- ------------------------------------------------------------------ -->
<?php auswertung_zusammenfassung(); ?>
```

B.2 Network File System (NFS) exercise

```
<!-- DB updated by feyrer on Sun Feb 22 23:54:29 MET 2004 from nfs.php -->
<!-- Id: nfs.php,v 1.15 2004/06/03 10:27:12 feyrer Exp  -->
<?php auswertung_ueberschrift(); ?>
<!-- ------------------------------------------------------------------ -->

<h1> NFS Server und Client Setup</h1>

In dieser Übung soll auf den beiden vulab-Rechner das Network File
System (NFS) installiert werden. Dabei wird auf dem Rechner "vulab1"
der NFS-Server, auf dem Rechner "vulab2" der NFS-Client installiert.
<p>

<h2>1. Server (Solaris): vulab1</h2>

Das Dateisystem /usr/homes soll für den zweiten Rechner 'vulab2' per
NFS exportiert werden:
<p>

<ul>
<li> Sichern Sie die Datei, in der bisher die NFS-Exports notiert sind
<li> Das Verzeichnis /usr/homes soll für den Rechner "vulab2"
     freigegeben werden. Tragen Sie dies in die richtige Datei ein.

     <?php auswertung_teiluebungen(
                 864  // vulab1: check-file-contents FILE=/etc/dfs/dfstab CONTENT_SHOULD='share.*nfs.*/usr/homes'
                      //         'share nfs /usr/homes' in /etc/dfs/dfstab?

        ); ?>

<li> Laufen die nötigen Serverprozesse? Starten Sie sie ggf. mit Hilfe
     der passenden Start-Scripten aus /etc/*.d.

     <?php auswertung_teiluebungen(
                 865, // vulab1: unix-check-process-running PROCESS=rpcbind
                      //         Läuft rpcbind?

                 866, // vulab1: unix-check-process-running PROCESS=mountd
                      //         Läuft mountd?

                 867, // vulab1: unix-check-process-running PROCESS=nfsd
                      //         Läuft nfsd?

                 868, // vulab1: unix-check-process-running PROCESS=statd
                      //         Läuft statd?

                 869, // vulab1: unix-check-process-running PROCESS=lockd
                      //         Läuft lockd?

                 870  // vulab1: check-file-exists FILE='/etc/rc3.d/S15nfs.server'
                      //         NFS-Server wird im Runlevel 3 gestartet?

        ); ?>

<li> Sorgen Sie dafür dass die Datei (neu) eingelesen wird

     <?php auswertung_teiluebungen(
                 871  // vulab1: check-program-output PROGRAM='share' OUTPUT_SHOULD='/usr/homes'
                      //         share(1M) listet /usr/homes? (unportabel!)

        ); ?>
```

```
<li> Überprüfen Sie mit ’showmount -e’ ob die Freigabe besteht!

     <?php auswertung_teiluebungen(
                 872  // vulab1: check-program-output PROGRAM=’showmount -e localhost’ OUTPUT_SHOULD=’/usr/homes’
                      //         showmount(1) zeigt /usr/homes?

        ); ?>

</ul>


<h2>2. Client (NetBSD): vulab2</h2>

Das Verzeichnis /usr/homes soll vom NFS-Server (vulab1) auf /usr/homes
gemountet werden:
<p>

<ul>
<li> Existiert der Mountpoint /usr/homes auf dem Client?
<li> Sind Daten im Mountpoint enthalten?
<li> Überprüfen Sie mit ’showmount -e’ die NFS-Freigaben des NFS-Servers
     ’vulab1’ (10.0.0.1)

     <?php auswertung_teiluebungen(
                 873  // vulab2: check-program-output PROGRAM=’showmount -e vulab1’ OUTPUT_SHOULD=’/usr/homes’
                      //         showmount(1) zeigt /usr/homes?

        ); ?>

<li> Untersuchen Sie die System-Defaults in /etc/defaults/rc.conf und
     tragen Sie für NFS nötige Abweichungen in die Datei /etc/rc.conf
     ein. Achten Sie auf rpc.lockd(8) und rpc.statd(8)!

     <?php auswertung_teiluebungen(
                 874, // vulab2: netbsd-check-rcvar-set RCVAR=rc_configured
                      //         /etc/rc.conf: rc_configured gesetzt?

                 875, // vulab2: netbsd-check-rcvar-set RCVAR=lockd
                      //         /etc/rc.conf: lockd gesetzt?

                 876, // vulab2: netbsd-check-rcvar-set RCVAR=statd
                      //         /etc/rc.conf: statd gesetzt?

                 877  // vulab2: netbsd-check-rcvar-set RCVAR=nfs_client
                      //         /etc/rc.conf: nfs_client gesetzt?

        ); ?>

<li> Starten Sie alle nötigen Hintergrundprozesse.

     <?php auswertung_teiluebungen(
                 878, // vulab2: unix-check-process-running PROCESS=rpcbind
                      //         Läuft rpcbind?

                 879, // vulab2: unix-check-process-running PROCESS=rpc.lockd
                      //         Läuft rpc.lockd?

                 880  // vulab2: unix-check-process-running PROCESS=rpc.statd
                      //         Läuft rpc.statd?

        ); ?>

<li> Überprüfen Sie, ob das Verzeichnis /usr/homes von vulab1 testweise
     auf /mnt gemountet werden kann. Unmounten Sie es anschliessend
     wieder!

     <?php auswertung_teiluebungen(
                 881  // vulab2: unix-check-mount MOUNT_FROM=vulab1:/usr/homes MOUNT_ON=/mnt
                      //         Manueller mount erfolgreich?

        ); ?>

<li> Sorgen Sie dafür daß das Verzeichnis /usr/homes vom NFS-Server
     "vulab1" beim Systemstart auf /usr/homes gemountet wird, tragen Sie
     dies in die passenden Konfigurationsdatei ein

     <?php auswertung_teiluebungen(
                 882  // vulab2: check-file-contents FILE=/etc/fstab CONTENT_SHOULD=’vulab1:/usr/homes.*/usr/homes.*nfs.*rw’
                      //         Passender Eintrag in /etc/fstab?

        ); ?>

<li> Mounten Sie alle noch nicht gemounteten NFS-Verzeichnisse!
<li> Überprüfen Sie mit df(1) und mount(8) daß das Verzeichnis gemountet
     ist!

     <?php auswertung_teiluebungen(
                 883, // vulab2: check-program-output PROGRAM=’df -k | grep :’ OUTPUT_SHOULD=’^vulab1:/usr/homes.*/usr/homes$’
```

```
                    //         Mount ist im df(1) Output sichtbar?

                884  // vulab2: check-program-output PROGRAM='mount | grep nfs' OUTPUT_SHOULD='^vulab1:/usr/homes on /usr/homes'
                    //         Mount ist im mount(8) Output sichtbar?

        ); ?>

</ul>


<h2>3. Zugriffsrechte</h2>

<h3>3.1 Rechnerbasiert</h3>

<ul>
<li> Legen Sie als root auf dem NFS-Client ein Verzeichnis
     /usr/homes/nfsuser an! Wie reagiert das System, und warum?
<li> Lesen Sie auf dem NFS-Server die Manpage zu dfstab(4) und den darin
     unter "SEE ALSO" verwiesenen Befehlen (etc.), und sorgen Sie dafür,
     daß Sie als root auf dem NFS-Client vollen Zugriff habe
<li> Machen Sie die nötige Änderung in /etc/dfs/dfstab.

     <?php auswertung_teiluebungen(
                885  // vulab1: check-file-contents FILE=/etc/dfs/dfstab CONTENT_SHOULD='root='
                    //         'root=' Eintrag in dfstab?

        ); ?>

<li> Lesen Sie die Datei neu ein!

     <?php auswertung_teiluebungen(
                886  // vulab1: check-program-output PROGRAM='share' OUTPUT_SHOULD='/usr/homes.*root='
                    //         share(1M) exportiert /usr/homes für root zugreifbar?

        ); ?>

<li> Welche Sicherheitsimplikationen hat der eben vorgenommene
     Konfigurationsschritt? Macht er in der Praxis Sinn? Wie kann man
     ihn umgehen?
<li> Legen Sie das Verzeichnis /usr/homes/nfsuser an!

     <?php auswertung_teiluebungen(
                887  // vulab1: check-directory-exists DIR=/usr/homes/nfsuser
                    //         Existiert Verzeichnis /usr/homes/nfsuser?

        ); ?>

</ul>

<h3>3.2 Benutzerbasiert</h3>

Es soll ein Benutzer "nfsuser" auf beiden Systemen angelegt werden,
der auf jedem System lokal vermerkt ist (Login, Passwort etc. in
/etc/...), das Home-Verzeichnis /usr/homes/nfsuser soll aber auf
beiden Rechnern mittels NFS verfügbar sein!

<ul>
<li> Legen Sie auf vulab1 den User an: ''useradd -d /usr/homes/nfsuser
     nfsuser''

     <?php auswertung_teiluebungen(
                888  // vulab1: unix-check-user-exists LOGIN=nfsuser
                    //         Benutzer 'nfsuser' existiert auf vulab1?

        ); ?>

<li> Legen Sie auf vulab2 denselben User an: ''useradd -d
     /usr/homes/nfsuser nfsuser''

     <?php auswertung_teiluebungen(
                889  // vulab2: unix-check-user-exists LOGIN=nfsuser
                    //         Benutzer 'nfsuser' existiert auf vulab2?

        ); ?>

<li> Geben Sie dem Benutzer auf beiden Systemen (getrennt) mittels
     passwd(1) ein Passwort
<li> Geben Sie das Verzeichnis /usr/homes/nfsuser mittels chown(1) dem
     Benutzer "nfsuser".

     <?php auswertung_teiluebungen(
                890, // vulab1: unix-check-file-owner FILE=/usr/homes/nfsuser OWNER_SHOULD=nfsuser
                    //         Gehört /usr/homes/nfsuser dem Benutzer 'nfsuser' auf vulab1?

                891  // vulab2: unix-check-file-owner FILE=/usr/homes/nfsuser OWNER_SHOULD=nfsuser
                    //         Gehört /usr/homes/nfsuser dem Benutzer 'nfsuser' auf vulab2?

        ); ?>
```

```
<li> Loggen Sie sich auf beiden Rechner als User "nfsuser" ein und legen Sie
     eine Datei "hallo-von-vulab1" bzw. "hallo-von-vulab2" an.
<li> Welches Problem besteht?
<li> Geben Sie auf beiden Rechnern dem Benutzer "nfsuser" die User-ID
     2000, stellen Sie sicher dass das Home-Verzeichnis (inkl. Inhalt)
     auch dem User gehört, und legen Sie die beiden Dateien erneut an.

     <?php auswertung_teiluebungen(
                 892, // vulab1: unix-check-file-owner FILE=/usr/homes/nfsuser/hallo-von-vulab1 OWNER_SHOULD=nfsuser
                      //         hallo-von-vulab1 gehört nfsuser auf vulab1?

                 893, // vulab2: unix-check-file-owner FILE=/usr/homes/nfsuser/hallo-von-vulab1 OWNER_SHOULD=nfsuser
                      //         hallo-von-vulab1 gehört nfsuser auf vulab2?

                 894, // vulab1: unix-check-file-owner FILE=/usr/homes/nfsuser/hallo-von-vulab2 OWNER_SHOULD=nfsuser
                      //         hallo-von-vulab2 gehört nfsuser auf vulab1?

                 895  // vulab2: unix-check-file-owner FILE=/usr/homes/nfsuser/hallo-von-vulab2 OWNER_SHOULD=nfsuser
                      //         hallo-von-vulab2 gehört nfsuser auf vulab2?

        ); ?>
</ul>


<h2>Hinweise:</h2>

<ul>
<li> Solaris-Pakete für bash und tcsh liegen in /cdrom, Installation
     mit pkgadd(1).

     <?php auswertung_teiluebungen(
                 902, // vulab1: solaris-check-installed-pkg PKG=SUNWtcsh
                      //         tcsh auf Solaris installiert? (pkginfo SUNWtcsh)

                 903  // vulab1: solaris-check-installed-pkg PKG=SUNWbash
                      //         bash auf Solaris installiert? (pkginfo SUWNbash)
        ); ?>

<li> NetBSD-Pakete für bash und tcsh (und weitere) liegen auf
     ftp://ftp.de.netbsd.org/pub/NetBSD/packages/1.6.1/sparc/All,
     Installation mit pkg_add(1).

     <?php auswertung_teiluebungen(
                 904, // vulab2: netbsd-check-installed-pkg PKG=tcsh
                      //         tcsh auf NetBSD installiert? (pkg_info -e tcsh)

                 905  // vulab2: netbsd-check-installed-pkg PKG=bash
                      //         bash auf NetBSD installiert? (pkg_info -e bash)
        ); ?>

</ul>

<!-- ---------------------------------------------------------------- -->
<?php auswertung_zusammenfassung(); ?>
```

C The VUDSL processor: uebung2db

```
#!/usr/pkg/bin/perl

use DBI;
use Getopt::Std;

$checkscript_path="/vulab";            # check-script
#HF#$checkscript_path="/home/feyrer/work/vulab/docs/hubertf/code";


getopts('dv');

$debug=1
    if $opt_d;
$verbose=1
    if $opt_v or $opt_d;

$uebung_id = $ARGV[0];
$template = $ARGV[1];
$output = $ARGV[2];

die "Usage: $0 [-dv] uebung_id uebung.php-template neue_uebung.php\n"
if $uebung_id eq ""
```

```
   or $template eq ""
   or $output eq ""
           or $template eq $output;

open(OUTPUT, ">$output")
    or die "Can't write $output: $!\n";

#$dbh = DBI->connect("dbi:Pg:", "vulab", "", { AutoCommit => 0 })
# or die "cannot connect to DB";
$dbh = DBI->connect("dbi:Pg:dbname=vulab;host=smaug", "vulab", "vulab", { AutoCommit => 0 })
or die "cannot connect to DB";

%checks_done = ();
$warnings = 0;

header();
main();
delete_old();

close(OUTPUT);

if ($warnings > 0 and !$debug) {
    print "Transaction rolled back, output file removed due to warnings.\n";
    print "Please fix!\n";
    unlink($output);

    $dbh->rollback;
} else {
    $dbh->commit;
}

$dbh->disconnect();

exit(0);


##############################################################################
sub warning {
    print "WARNING: @_\n";
    $warnings++;
}

##############################################################################
sub header() {
    local($now);
    chomp($now = 'date');
    print OUTPUT "<!-- DB updated by $ENV{'USER'} on $now from $template -->\n";
}

##############################################################################
sub main() {
    open(T, $template) or die "can't read $template: $!\n";
    while(<T>) {
chomp;
($check_id, $komma, $rechner, $script, $parameter) =
    m@\s*([0-9X?]+)([,  ])?\s*//\s+([a-zA-Z0-9_]*):\s+([^ ]*check-[^ ]+)\s+(.*)@;

if ($rechner eq "") {
    print OUTPUT "$_\n"
if !/Generated by.* on .* from/; # skip header
    next;
}

### 1. Syntax-Check etc.

## Check if script present
if ( ! -f "$checkscript_path/$script" ) {
    warning("missing check-script '$script'");
    next;
}

$interpreter = get_interpreter("$checkscript_path/$script");
print "$check_id: cat $script | ssh $rechner env $parameter '$interpreter'\n"
    if $debug;

chomp($bezeichnung = <T>);
if ($bezeichnung !~ m@^\s*//\s*\S+\s*@) {
    warning("no comment for $check_id ($rechner: $script $parameter)");
}
$bezeichnung =~ s,^\s*//\s*,,;
$bezeichnung =~ s,\s*$,,;
print "    bezeichnung=\"$bezeichnung\"\n"
    if $debug;
print "\n"
    if $debug;

## Rechner bekannt?
$sth = $dbh->prepare("SELECT * ".
```

```
 "FROM rechner ".
 "WHERE bezeichnung='$rechner'");
$sth->execute();
while(@row = $sth->fetchrow_array) {
    if ($row[0] eq $rechner ) {
print "    rechner OK: $rechner\n"
    if $debug;
    } else {
warning("rechnercheck unknown host: $rechner");
    }
}


## Check parameters
# Get possible parms
open(P, "$interpreter $checkscript_path/$script listparms |")
    or die "Can't listparms for $script: $!\n";
while(<P>) {
    @p = split(/\|/);
    $par{$p[0]} = $p[1];
    #print " $p[0]";
}
close(P);
#print "\n";

# Parse into variables using sh & env
open(P, "env -i $parameter env |")
    or die "Can't env(1) $parameter";
while(<P>) {
    chomp();
    ($var, $val) = /([a-zA-Z0-9_]+)=(.*)/;
    #print "   $var -> $val\n";

    if (exists($par{$var})) {
print "    varcheck OK: $var=$val"
    if $debug;
if ("$par{$var}" eq "$val") {
    print " (default)"
if $debug;
}
print "\n"
    if $debug;
    } else {
warning("varcheck unknown variable: $var=$val");
    }
}
close(P);


### 2. Check & Insert/Update things into DB
if ($check_id =~ /\d+/) {
    # Might be already-existing check, make sure...
    $sth = $dbh->prepare("SELECT check_id, uebung_id, script, ".
 "       bezeichnung, rechner, parameter ".
 "FROM uebungs_checks ".
 "WHERE check_id='$check_id' " .
 "      AND uebung_id='$uebung_id'");
    $sth->execute();

    $cnt=0;
    while ( @row = $sth->fetchrow_array ) {
# Check already there, update!
( $db_check_id, $db_uebung_id, $db_script, $db_bezeichnung,
  $db_rechner, $db_parameter ) = @row;

if ($debug) {
    print "\n";
    print "    In DB, check_id=$check_id:\n    ";
    print "cat $db_script | ssh $db_rechner env $db_parameter interp\n";
    print "    bezeichnung=\"$db_bezeichnung\"\n";
    print "\n";
}

if ($script ne $db_script
    or $bezeichnung ne $db_bezeichnung
    or $rechner ne $db_rechner
    or $parameter ne $db_parameter) {
    update_db($check_id, $uebung_id, $script,
      $bezeichnung, $rechner, $parameter);
    print "check_id $check_id updated\n" if $verbose;
} else {
    print "check_id $check_id unchanged\n" if $verbose;
}

$cnt++;
    }

    if ($cnt == 0) {
```

```
# Check not there, insert new!
$check_id = insert_into_db($uebung_id, $script,
       $bezeichnung, $rechner,
       $parameter);
print "check_id $check_id inserted (1)\n" if $verbose;
    }

} else {
    # Check not there, insert new!
    $check_id = insert_into_db($uebung_id, $script,
   $bezeichnung, $rechner,
   $parameter);
    print "check_id $check_id inserted (2)\n" if $verbose;
}


### 3. Write out new file w/ check_ids added
print OUTPUT "\t\t". sprintf("%4d", $check_id).
             "$komma // $rechner: $script $parameter\n";
print OUTPUT "\t\t      //         $bezeichnung\n";

$checks_done{$check_id} = 1;

print "\n"
    if $debug;
    }
    close(T);
}


##############################################################################
sub delete_old() {
    $ids=join(", ", sort keys %checks_done );
    $sql="DELETE FROM uebungs_checks ".
 "WHERE check_id NOT IN ( $ids ) ".
         "       AND uebung_id='$uebung_id'";
    print "    SQL: $sql;\n"
if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();
    print "old checks removed from database\n"
if $verbose;
}

##############################################################################
sub update_db() {
    local($check_id, $uebung_id, $script, $bezeichnung, $rechner,
  $parameter) = @_;
    local($sth);

    $parameter =~ s/\\/\\\\/g;
    $parameter =~ s/'/\\'/g;
    $bezeichnung =~ s/\\/\\\\/g;
    $bezeichnung =~ s/'/\\'/g;

    $sql = "UPDATE uebungs_checks ".
   "SET ".
   "    uebung_id='$uebung_id', ".
   "    script='$script', ".
   "    bezeichnung='$bezeichnung', ".
   "    rechner='$rechner', ".
   "    parameter='$parameter' ".
   "WHERE ".
   "    check_id='$check_id'";
    print "    SQL: $sql;\n"
if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();
}

##############################################################################
sub insert_into_db() {
    local($uebung_id, $script, $bezeichnung, $rechner,
  $parameter) = @_;
    local($sth);

    $parameter =~ s/\\/\\\\/g;
    $parameter =~ s/'/\\'/g;
    $bezeichnung =~ s/\\/\\\\/g;
    $bezeichnung =~ s/'/\\'/g;

    # 1. insert new
    $sql = "INSERT INTO uebungs_checks ".
   "    ( uebung_id, script, bezeichnung, ".
   "      rechner, parameter ) ".
   "VALUES ".
   "    ( '$uebung_id', '$script', '$bezeichnung', ".
   "      '$rechner', '$parameter' )";
```

```
    print "    SQL: $sql;\n"
if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();

    # 2. find $new_check_id
    $sql = "SELECT check_id ".
   "FROM uebungs_checks ".
   "WHERE uebung_id='$uebung_id' ".
   "      AND script='$script' ".
   "      AND bezeichnung='$bezeichnung' ".
   "      AND rechner='$rechner' ".
   "      AND parameter='$parameter'";
    print "    SQL: $sql;\n"
if $debug;
    $sth = $dbh->prepare($sql);
    $sth->execute();

    while(@row = $sth->fetchrow_array) {
$new_check_id = $row[0];
    }
    print "    new check_id=$new_check_id\n"
if $debug;

    return $new_check_id;
}

##########################################################################
sub get_interpreter() {
    local($file) = @_;
    local($i, $rc);

    die "No such file: $file\n"
if ( ! -f $file );

    open(F, "$file") or die "can't open $file: $!\n";
    $i = <F>;
    close(F);

    if ($i =~ /perl/ ) {
#$rc="perl || /root/vulab/perl";
$rc="perl";
    } elsif ($i =~ /[^c]sh/) {
$rc = "sh";
    }

    return $rc;
}
```

D Complete lists of checks used in exercises

This section provides complete lists of checks that are performed for both thee Network Information System (NIS) and the Network File System (NFS) exercises. The data is retrieved from the Virtual Unix Lab's database, and the SQL queries and their results are shown.

D.1 Network Information System (NIS) exercise

This section lists all the checks that are performed by the NIS exercise as stated in the Virtual Unix Lab's database.

```
vulab=> select check_id,bezeichnung from uebungs_checks where uebung_id='nis';
 check_id |                          bezeichnung
----------+-------------------------------------------------------------
```

```
      775 | Domäne im laufenden System (domainname(1)) gesetzt?
      776 | Existiert /var/yp/Makefile?
      777 | Existiert /var/yp/binding/vulab/ypservers?
      778 | Existiert /var/yp/passwd.time?
      779 | Gibt ypwhich(1) 'vulab1' zurück?
      780 | Daten in passwd-Map vorhanden?
      781 | Daten in host-Map vorhanden?
      782 | Daten in group-Map vorhanden?
      784 | Existiert /var/yp/passwd?
      785 | Verzeichnis /usr/homes/ypuser existiert?
      786 | Shell von ypuser auf ksh gesetzt?
      787 | User ypuser in /var/yp/passwd eingetragen?
      788 | User ypuser in passwd NIS Map vorhanden?
      789 | User existiert (getpwnam(3))?
      790 | passwd-Information wird in NIS gesucht (/etc/nsswitch.conf)?
      791 | group-Information wird in NIS gesucht (/etc/nsswitch.conf)?
      792 | hosts-Information wird in NIS gesucht (/etc/nsswitch.conf)?
      793 | Domainname in /etc/defaultdomain gesetzt?
      795 | /etc/rc.conf: rc_configured gesetzt?
      796 | /etc/rc.conf: rpcbind gesetzt?
      797 | /etc/rc.conf: ypbind gesetzt?
      798 | rpcbind läuft?
      799 | ypbind läuft?
      800 | Wird vulab1 als NIS-Server verwendet? (ypwhich(1))
      801 | Daten in passwd-Map vorhanden?
      802 | Daten in hosts-Map vorhanden?
      803 | Daten in group-Map vorhanden?
      804 | Existiert Benutzer ypuser?
      805 | Existiert Home-Verzeichnis?
      806 | Paßwort richtig gesetzt?
      807 | Fullname richtig gesetzt?
      808 | Benutzer 'ypuser' Mitglied der Gruppe 'benutzer'?
      809 | Gruppe 'benutzer' existiert in der group NIS-Map?
      810 | ypuser in wheel-Gruppe in /etc/group?
      811 | Eintrag mit IP-Nummer und Rechnername in hosts NIS-Map?
      774 | Domäne in /etc/defaultdomain gesetzt?
      794 | Domainname im laufenden System gesetzt? (domainname(1))
      783 | PWDIR in /var/yp/Makefile auf /var/yp gesetzt?
      812 | 'tab' pingbar?
      898 | tcsh auf Solaris installiert? (pkginfo SUNWtcsh)
      899 | bash auf Solaris installiert? (pkginfo SUNWbash)
      900 | tcsh auf NetBSD installiert? (pkg_info -e tcsh)
      901 | bash auf NetBSD installiert? (pkg_info -e bash)
(43 rows)
```

D.2 Network File System (NFS) exercise

This section describes the checks that are performed for the NFS exercise.

```
vulab=> select check_id,bezeichnung from uebungs_checks where uebung_id='nfs';
 check_id |                          bezeichnung
----------+------------------------------------------------------------
      864 | 'share nfs /usr/homes' in /etc/dfs/dfstab?
      865 | Läuft rpcbind?
      866 | Läuft mountd?
      867 | Läuft nfsd?
      868 | Läuft statd?
      869 | Läuft lockd?
      870 | NFS-Server wird im Runlevel 3 gestartet?
```

```
      874 | /etc/rc.conf: rc_configured gesetzt?
      875 | /etc/rc.conf: lockd gesetzt?
      876 | /etc/rc.conf: statd gesetzt?
      877 | /etc/rc.conf: nfs_client gesetzt?
      878 | Läuft rpcbind?
      879 | Läuft rpc.lockd?
      880 | Läuft rpc.statd?
      881 | Manueller mount erfolgreich?
      882 | Passender Eintrag in /etc/fstab?
      883 | Mount ist im df(1) Output sichtbar?
      884 | Mount ist im mount(8) Output sichtbar?
      885 | 'root=' Eintrag in dfstab?
      887 | Existiert Verzeichnis /usr/homes/nfsuser?
      888 | Benutzer 'nfsuser' existiert auf vulab1?
      889 | Benutzer 'nfsuser' existiert auf vulab2?
      890 | Gehört /usr/homes/nfsuser dem Benutzer 'nfsuser' auf vulab1?
      891 | Gehört /usr/homes/nfsuser dem Benutzer 'nfsuser' auf vulab2?
      892 | hallo-von-vulab1 gehört nfsuser auf vulab1?
      893 | hallo-von-vulab1 gehört nfsuser auf vulab2?
      894 | hallo-von-vulab2 gehört nfsuser auf vulab1?
      895 | hallo-von-vulab2 gehört nfsuser auf vulab2?
      871 | share(1M) listet /usr/homes? (unportabel!)
      872 | showmount(1) zeigt /usr/homes?
      873 | showmount(1) zeigt /usr/homes?
      886 | share(1M) exportiert /usr/homes für root zugreifbar?
      902 | tcsh auf Solaris installiert? (pkginfo SUNWtcsh)
      903 | bash auf Solaris installiert? (pkginfo SUWNbash)
      904 | tcsh auf NetBSD installiert? (pkg_info -e tcsh)
      905 | bash auf NetBSD installiert? (pkg_info -e bash)
(36 rows)
```

E List of check scripts and parameters

This section lists check scripts available in step II of the Virtual Unix Lab, a textual description of what they do as printed by the `whatis` parameter and a list of parameters as printed by the `listparms` parameter. As step II was in German language, so are the descriptions given here. A future implementation of the Virtual Unix Lab may pay attention to internationalization.

admin-check-clearharddisk: Festplatte zum Komprimieren optimieren (mit 0-Bits beschreiben)

Parameters:

- (none)

admin-check-makeimage: Muss auf localhost laufen! Erzeugt Plattenimage von $DISK von $RECHNER in Datei $IMGFILE.img; Zeit ca. 30min

Parameters:

- RECHNER (Default: 'unset'): Rechner dessen Platte in IMGFILE verpackt werden soll (vulab1, ...)
- IMGFILE (Default: 'unset'): Imagefile, relativ zu /vulab
- DISK (Default: 'sd0'): Platte, von der das Image gemacht werden soll

netbsd-check-installed-pkg: Tested ob Paket $PKG unter NetBSD installiert ist

Parameters:

- PKG (Default: 'tcsh'): Package-Pattern fuer pkg_info -e

netbsd-check-rcvar-set: Testet ob Variable RCVAR in /etc/rc.conf gesetzt ist (NetBSD)

Parameters:

- RCVAR (Default: 'rc_configured'): Variable, die überprüft werden soll

netbsd-check-user-shell: Tested ob Shell von User $LOGIN gleich $SHELL_SHOULD in /etc/master.passwd

Parameters:

- LOGIN (Default: 'test'): Benutzer, dessen Shell ueberprueft werden soll
- SHELL_SHOULD (Default: '/.*/tcsh'): Regulaerer Ausdruck, gegen den verglichen werden soll.

solaris-check-installed-pkg: Tested ob Paket $PKG unter Solaris installiert ist

Parameters:

- PKG (Default: 'tcsh'): Package-Pattern fuer pkginfo

unix-check-file-owner: Prueft ob FILE dem Benutzer OWNER_SHOULD (Login oder UID) gehoert.

Parameters:

- FILE (Default: '/etc/passwd'): Datei oder Verzeichnis, absolut.
- OWNER_SHOULD (Default: 'root'): Login-Name oder numerische User-ID

unix-check-mount: Versucht MOUNT_FROM auf MOUNT_ON zu mounten

Parameters:

- MOUNT_FROM (Default: 'foo'): Erstes Argument fuer mount(8)
- MOUNT_ON (Default: '/mnt'): Mountpoint, muss existieren
- MOUNT_ARGS (Default: 'none'): Parameter fuer mount(8)

unix-check-process-running: Tested ob PROCESS läuft (Regulärer Ausdruck gegen ps(1)-Output)

Parameters:

- PROCESS (Default: 'init'): Regulärer Ausdruck, gegen den der Output von ps -elf/aux verglichen wird.

unix-check-user-exists: Tested ob der Benutzer $LOGIN existiert (via getpwnam())

Parameters:

- LOGIN (Default: 'test'): Benutzer, dessen Home-Dir ueberprueft werden soll

unix-check-user-fullname: Tested ob der volle Name von LOGIN gleich FULLNAME_SHOULD ist (via getpwnam())

Parameters:

- LOGIN (Default: 'root'): Benutzer, dessen Fullname überprüft werden soll
- FULLNAME_SHOULD (Default: 'Charlie Root'): String auf den der Fullname gesetzt sein sollte

unix-check-user-home: Tested ob das Home-Verzeichnis von User $LOGIN gleich $HOME_SHOULD ist (via getpwnam())

Parameters:

- LOGIN (Default: 'test'): Benutzer, dessen Home-Dir ueberprueft werden soll
- HOME_SHOULD (Default: '*'): Pfad auf den das Home-Verzeichnis gesetzt sein sollte

unix-check-user-ingroup: Tested ob User $LOGIN in Gruppe GROUP_SHOULD ist (primary oder supplementary)

Parameters:

- LOGIN (Default: 'test'): Login-Name
- GROUP_SHOULD (Default: 'wheel'): Primäre oder Supplementäre Gruppe, in der der Benutzer sei sollte

unix-check-user-password: Tested ob Passwort von User $LOGIN gleich $PASSWD_SHOULD (plain)

Parameters:

- LOGIN (Default: 'test'): Benutzer, dessen Passwort ueberprueft werden soll
- PASSWD_SHOULD (Default: '*'): Plaintext-Passwort (unverschluesselt), gegen das geprueft wird

unix-check-user-shell: Tested ob die Login-Shell von User $LOGIN gleich $SHELL_SHOULD ist (via getpwnam())

Parameters:

- LOGIN (Default: 'test'): Benutzer, dessen Login-Shell ueberprueft werden soll
- SHELL_SHOULD (Default: '/bin/sh'): Pfad auf den die Shell gesetzt sein sollte

F Selected check scripts

This section shows the full source code of selected check scripts from step I, i.e. before optimizing, and from step II, i.e. after optimizing.

F.1 Step I

F.1.1 netbsd-check-finger.sh

```
#!/bin/sh
#
# Checks if $user exists,  NetBSD-specific

user=test

if [ ‘finger $user | grep Shell: | wc -l‘ = 1 ]
then
        rc=ok
else
        rc=failed
fi

echo $rc
```

F.1.2 netbsd-check-masterpw.sh

```
#!/bin/sh
#
# Checks if $user exists,  NetBSD-specific

user=test

grep -l $user /etc/master.passwd 2>&1 >/dev/null
rc=$?

echo $rc
```

F.1.3 netbsd-check-pkginstalled.sh

```
#!/bin/sh

pkg_info -qe tcsh
tcsh_installed=$?

pkg_info -qe bash
bash_installed=$?

echo tcsh_installed=$tcsh_installed
echo bash_installed=$bash_installed

if [ $tcsh_installed = 0 -a $bash_installed = 0 ]
then
        rc=ok
else
        rc=failed
fi

echo $rc
```

F.1.4 netbsd-check-pw.pl

```
#!/usr/local/bin/perl

$user="test";
$should_pwu="vutest";

$is_pwe=(getpwnam($user))[1];
($salt) = ($is_pwe =~ /^(..)/);
$should_pwe=crypt($should_pwu, $salt);

print "is_pwe=$is_pwe\n";
print "salt='$salt'\n";
print "should_pwe=$should_pwe\n";

if ( $is_pwe eq $should_pwe ) {
        $rc = "ok";
} else {
        $rc = "failed";
}

print "$rc\n";
```

F.1.5 netbsd-check-usershell2.sh

```
#!/bin/sh
#
# Tests if shell of user $user is set to $should_shell

user=vulab
should_shell='/.*/bash'

### NO CHANGES FROM HERE

is_shell=`finger $user | grep Shell | awk '{print $4}'`

echo is_shell=$is_shell
echo should_shell=$should_shell

if expr "$is_shell" : "$should_shell" >/dev/null
then
        rc=ok
else
        rc=wrong
fi

echo $rc
```

F.1.6 check-program-output

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Tested ob Ausgabe von PROGRAM den regulären Ausdruck OUTPUT_SHOULD enthält
***
```

```
';

# Based on work by Thomas Ernst <herr.ernst@gmx.de>

################################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "PROGRAM", "true",
           "Programm für sh -c '$PROGRAM'" ],
         [ "OUTPUT_SHOULD", "Hallo Welt!",
           "zu suchender Regulärer Ausdruck" ],
         [ "VERBOSE", "",
           "Ausgabe ausgeben" ]
         );

################################################################################
# Check-Spezifisch:
sub check()
{
    print "PROGRAM='$PROGRAM'\n";
    print "OUTPUT_SHOULD='$OUTPUT_SHOULD'\n";
    print "VERBOSE='$VERBOSE'\n";
    print "\n";

    $rc = "wrong";
    if (open(F, "$PROGRAM 2>&1 |")) {
        while(<F>){
            if(/$OUTPUT_SHOULD/){
                print "Match: $_\n";

                $rc = "ok";
                last;
            } elsif ($VERBOSE) {
                print "$_";
            }
        }
        close(F);
    }

    print "$rc\n";
}

################################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
```

```
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

F.2 Step II

F.2.1 admin-check-clearharddisk

```
#!/bin/sh
################################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
WHATIS='
***
*** Festplatte zum Komprimieren optimieren (mit 0-Bits beschreiben)
***
'

################################################################################
### Parameter:

vars=""

################################################################################
### Check-Spezifisch:
check()
{
cd /

echo Cleaning empty blocks...
dd if=/dev/zero of=0 bs=1048576

sleep 1
echo ""
echo Cleaning up...
rm -f 0

echo Done.
echo ok
}

################################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
################################################################################
### Common code:
# Variablen uebernehmen
for var in $vars ; do
eval "$var=\"\${$var:=\${${var}_def}}\""
done
```

```
# "Hauptprogramm"
if [ "$1" = 'listparms' ]; then
for var in $vars ; do
eval "echo \"$var|\${${var}_def}|\${${var}_bez}\""
done
elif [ "$1" = "whatis" ]; then
     echo "$WHATIS" | sed -e 's/^\*\*\* //g' | grep -v '^[  ]*$'
elif [ "$1" = "-h" ]; then
     echo "Usage: $0 [whatis|listargs|-h]"
else
check
fi
```

F.2.2 admin-check-makeimage

```
#!/bin/sh
#
# Basiert in guten Teilen auf deploy1
#
# Sollte nur fuer einmalige Uebungen zur Imageerzeugung benutzt werden
# (anschliessend Uebung im VUlab loeschen!)
#

################################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
WHATIS='
***
*** Muss auf localhost laufen! Erzeugt Plattenimage von $DISK von $RECHNER in Datei $IMGFILE.img; Ze
***
'

################################################################################
### Parameter:

vars="RECHNER IMGFILE DISK"

RECHNER_def=unset
RECHNER_bez="Rechner dessen Platte in IMGFILE verpackt werden soll (vulab1, ...)"

IMGFILE_def=unset
IMGFILE_bez="Imagefile, relativ zu /vulab"

DISK_def='sd0'
DISK_bez="Platte, von der das Image gemacht werden soll"

################################################################################
### Check-Spezifisch:
check()
{
        imagePath="/vulab"
        imageHost=smaug

        client_log=logs/$RECHNER.log
        deployment_done_cookie="VULab Deployment Done"
        deployment_poll_interval=60             # seconds
        ssh="./rsh-wrapper -p 9999"

        if [ $RECHNER = unset ]; then
                echo RECHNER unset
                echo failed
```

```
        exit 0
fi

if [ $IMGFILE = unset ]; then
        echo IMGFILE unset
        echo failed
        exit 0
fi

if [ "‘uname -n‘" != $imageHost ]; then
        echo muss auf \"localhost\" laufen
        echo failed
        exit 0
fi

cd $imagePath

echo "RECHNER=$RECHNER"
echo "DISK=$DISK"
echo "IMGFILE=$IMGFILE"
echo "imagePath=$imagePath"
echo ""

echo Starting deployment: ‘date‘

# Define which image to create
echo ${DISK} ${IMGFILE} >mkimg-${RECHNER}

if [ "‘$ssh $RECHNER echo READY‘" != READY ]
then
        echo "Machine $RECHNER didn’t respond properly via ’$ssh’"
        echo failed
        exit 0
fi

# Setup client logfile
rm -f $client_log
install -m 777 /dev/null $client_log

# Kick client into netboot
echo "Starting netboot on $RECHNER in background..."
# Pfad fuer Solaris ist /usr/sbin/reboot, redirection Shell-abhaengig !!!
$ssh $RECHNER "env PATH=/usr/sbin:/sbin /bin/sh -c ’reboot -- net’ </dev/null 2>/dev/null >/
echo "done. (rc=$?)"

# Wait for client to startup on netboot properly
echo "Waiting a bit to get to /etc/rc..."
sleep 120 # takes about 70 seconds, plus some extra
if ! grep ^Starting $client_log >/dev/null 2>/dev/null
then
        echo "Client $RECHNER didn’t do netboot properly, aborting."
        exit 1
else
        echo "$RECHNER properly netbooted."
fi

# Client’s running, now wait for it to be done
while ! grep -q "$deployment_done_cookie" $client_log
do
        echo ‘date‘: waiting for $RECHNER to finish: ‘tail -1 $client_log‘
        sleep $deployment_poll_interval
done
echo done.

# Clean up
```

```
        echo Cleaning up ...
        rm -f mkimg-${RECHNER}
        echo done.

        echo Checking if $RECHNER was installed properly
        sleep 120 # time for reboot
        if [ "‘$ssh $RECHNER echo READY‘" != READY ]
        then
                echo "$0: machine $RECHNER didn’t respond properly via ’$ssh’ after installing"
                exit 1
        fi
        echo OK.

        # Image in Tabelle ’images’ eintragen:
        echo -n Remember image $IMGFILE in database:
        echo "INSERT INTO images (bezeichnung) VALUES (’"$IMGFILE"’);" \
        | psql -U vulab

        echo Deployment done: ‘date‘

        echo ok
}

################################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
################################################################################
### Common code:
# Variablen uebernehmen
for var in $vars ; do
        eval "$var=\"\${$var:=\${${var}_def}}\""
done

# "Hauptprogramm"
if [ "$1" = ’listparms’ ]; then
        for var in $vars ; do
                eval "echo \"$var|\${${var}_def}|\${${var}_bez}\""
        done
elif [ "$1" = "whatis" ]; then
        echo "$WHATIS" | sed -e ’s/^\*\*\* //g’ | grep -v ’^[   ]*$’
elif [ "$1" = "-h" ]; then
        echo "Usage: $0 [whatis|listargs|-h]"
else
        check
fi
```

F.2.3 check-file-contents

```
#!/usr/bin/perl
################################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS=’
***
*** Tested ob FILE den regulären Ausdruck CONTENT_SHOULD enthält
***
’;

# Based on work by Thomas Ernst <herr.ernst@gmx.de>

################################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
```

```
#
@vars = (
         [ "FILE", "/etc/motd",
           "zu durchsuchende Datei, absoluter Pfad" ],
         [ "CONTENT_SHOULD", "Hallo Welt!",
           "zu suchender Regulärer Ausdruck" ]
         );

################################################################################
# Check-Spezifisch:
sub check()
{
    print "FILE=$FILE\n";
    print "CONTENT_SHOULD=$CONTENT_SHOULD\n";
    print "";

    $rc = "wrong";
    if (open(F, "$FILE")) {
        while(<F>){
            chomp();
            #print "$_\n";
            if(/$CONTENT_SHOULD/){
                $rc = "ok";
                last;
            }
        }
        close(F);
    }

    print "$rc\n";
}

################################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if(exists($ENV{$vars[$i][0]})) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
```

```
    check();
}
```

F.2.4 unix-check-user-exists

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Tested ob der Benutzer $LOGIN existiert (via getpwnam())
***
';

##############################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "LOGIN", "test",
           "Benutzer, dessen Home-Dir ueberprueft werden soll" ],
         );

##############################################################################
# Check-Spezifisch:
sub check()
{
        $login_is=(getpwnam($LOGIN))[0];

        print "login_is=$login_is\n";
        print "LOGIN=$LOGIN\n";
        print "\n";

        if ( $login_is eq $LOGIN ) {
            $rc="ok";
        } else {
            $rc="wrong";
        }

        print "$rc\n";
}

##############################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
##############################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
```

```
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

F.2.5 unix-check-user-shell

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Tested ob die Login-Shell von User $LOGIN gleich $SHELL_SHOULD ist (via getpwnam())
***
';

##############################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "LOGIN", "test",
           "Benutzer, dessen Login-Shell ueberprueft werden soll" ],
         [ "SHELL_SHOULD", "/bin/sh",
           "Pfad auf den die Shell gesetzt sein sollte" ],
         );

##############################################################################
# Check-Spezifisch:
sub check()
{
        $shell_is=(getpwnam($LOGIN))[8];

        print "LOGIN=$LOGIN\n";
        print "shell_is=$shell_is\n";
        print "SHELL_SHOULD=$SHELL_SHOULD\n";
        print "\n";

        if ( $shell_is =~ $SHELL_SHOULD ) {
            $rc="ok";
        } else {
            $rc="wrong";
        }

        print "$rc\n";
}
```

```
##############################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
##############################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

F.2.6 unix-check-user-password

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Tested ob Passwort von User $LOGIN gleich $PASSWD_SHOULD (plain)
***
';

##############################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "LOGIN", "test",
           "Benutzer, dessen Passwort ueberprueft werden soll" ],
         [ "PASSWD_SHOULD", "*",
           "Plaintext-Passwort (unverschluesselt), gegen das geprueft wird" ],
         );

##############################################################################
```

```
# Check-Spezifisch:
sub check()
{
        $passwd_is_e=(getpwnam($LOGIN))[1];
        ($salt) = ($passwd_is_e =~ /^(..)/);
        $PASSWD_SHOULD_u= $PASSWD_SHOULD;
        $PASSWD_SHOULD_e=crypt($PASSWD_SHOULD_u, $salt);

        print "passwd_is_e=$passwd_is_e\n";
        print "salt='$salt'\n";
        print "PASSWD_SHOULD_u=$PASSWD_SHOULD_u\n";
        print "PASSWD_SHOULD_e=$PASSWD_SHOULD_e\n";
        print "\n";

        if ( $PASSWD_SHOULD_e ne "" and $passwd_is_e eq $PASSWD_SHOULD_e ) {
            $rc="ok";
        } else {
            $rc="wrong";
        }

        print "$rc\n";
}

################################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
################################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

F.2.7 unix-check-process-running

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Tested ob PROCESS läuft (Regulärer Ausdruck gegen ps(1)-Output)
***
';

##############################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "PROCESS", "init",
           "Regulärer Ausdruck, gegen den der Output von ps -elf/aux verglichen wird." ],
         );

##############################################################################
# Check-Spezifisch:
sub check()
{
        print "PROCESS=$PROCESS\n";

        $rc = "wrong";
        if (open(P, "ps -elf 2>&1 || ps -auxwww  2>&1 |")) {
                while(<P>) {
                        if (/$PROCESS/) {
                                print "Match: $_\n";
                                $rc = "ok";
                                last;
                        }
                }
                close(P);
        }

        print "$rc\n";
}

##############################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
##############################################################################
### Common code:

# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
```

```
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

F.2.8 netbsd-check-rcvar-set

```
#!/usr/bin/perl
##############################################################################
# Kurzbeschreibung: "Dieses Script ($0) $WHATIS\n." (1 Zeile)
$WHATIS='
***
*** Testet ob Variable RCVAR in /etc/rc.conf gesetzt ist (NetBSD)
***
';

##############################################################################
# Parameter:
# [ "Variable", "Default, "Beschreibung der Variable" ]
#
@vars = (
         [ "RCVAR", "rc_configured",
           "Variable, die überprüft werden soll" ],
         );

##############################################################################
# Check-Spezifisch:
sub check()
{
        $rc=system(". /etc/rc.subr; ".
                   ". /etc/rc.conf; ".
                   "checkyesno $RCVAR; ".
                   "exit \$?");

        print "RCVAR=$RCVAR\n";
        print "rc=$rc\n";
        print "\n";

        if ( $rc == 0 ) {
            $rc="ok";
        } else {
            $rc="wrong";
        }

        print "$rc\n";
}

##############################################################################
### AB HIER NICHTS MEHR VERAENDERN !!!
##############################################################################
### Common code:
```

```
# Variablen übernehmen
sub init()
{
        for($i=0; $i<=$#vars; $i++) {
                if($ENV{$vars[$i][0]}) {
                        ${$vars[$i][0]} = $ENV{$vars[$i][0]};
                } else {
                        ${$vars[$i][0]} = $vars[$i][1];
                }
        }
}

# "Hauptprogramm"
if($ARGV[0] eq "listparms") {
        for($i=0;$i<=$#vars;$i++) {
                print "$vars[$i][0]|$vars[$i][1]|$vars[$i][2]\n";
        }
} elsif($ARGV[0] eq "whatis") {
        $WHATIS=~s/^\n*//g;
        $WHATIS=~s/\n\*\*\* ?//g;
        $WHATIS=~s/^\*\*\* ?//g;
        $WHATIS=~s/\n*$//g;
        print "$WHATIS\n";
} elsif($ARGV[0] eq "-h") {
        print "whatis     Kurzbeschreibung des Scripts\n";
        print "listparms  Listet Variablen mit Default und Beschreibung\n";
        print "-h         Alle Parameter\n";
        print "sonst      Check-Script wird ausgefuehrt\n";
} else {
    init();
    check();
}
```

G Database structure

This section describes the database tables used in the Virtual Unix Lab, the SQL-statements that were used to create the tables in the PostgreSQL database, and example database records in a few selected cases.

G.1 Table: `benutzer`

This table describes a user in the Virtual Unix Lab.

```
CREATE TABLE benutzer (
        user_id serial NOT NULL,                    -- unique user id
        vorname varchar(50) NOT NULL,               -- first name
        nachname varchar(50) NOT NULL,              -- last name
        matrikel_nr numeric(10) NOT NULL,           -- student id
        email varchar(80) NOT NULL,                 -- contact email
        login varchar(80) NOT NULL,                 -- vulab login
        passwort varchar(15) NOT NULL,              -- password
        freischalt_secret varchar(30) NOT NULL,     -- initial secret
```

```
        anmeldedatum date NOT NULL,                -- sign-on date
        typ varchar(80) DEFAULT 'user' NOT NULL,
        PRIMARY KEY (user_id),
        UNIQUE (user_id),
        UNIQUE (login),
        UNIQUE (matrikel_nr)
);
```

G.2 Table: `rechner`

This table contains a list of all the lab machines.

```
CREATE TABLE rechner (
        bezeichnung varchar(30) NOT NULL,          -- hostname
        PRIMARY KEY (bezeichnung),
        UNIQUE (bezeichnung)
);
```

G.3 Table: `images`

This table contains a list of all possible images that can be installed on the lab machines.

```
CREATE TABLE images (
        bezeichnung varchar(150) NOT NULL,         -- filename
        PRIMARY KEY (bezeichnung),
        UNIQUE (bezeichnung)
);
```

G.4 Table: `uebungen`

This table lists all possible exercises with their basic properties.

G.4.1 Definition

```
CREATE TABLE uebungen (
        uebung_id varchar(40) NOT NULL,            -- exercise id
        bezeichnung varchar(150) NOT NULL,         -- description
        nur_fuer varchar(40),                      -- user-restriction
        vorlauf time NOT NULL,                     -- preparation time
        dauer time NOT NULL,                       -- exercise duration
        nachlauf time NOT NULL,                    -- time for checks
        wiederholbar boolean NOT NULL,             -- repeatable?
        text varchar(150) NOT NULL,                -- exercise text filename
        mehr_info varchar(150),                    -- more information (unused)
```

```
        PRIMARY KEY (uebung_id),
        UNIQUE (uebung_id)
);
```

G.4.2 Example records

```
vulab=> select uebung_id, bezeichnung, vorlauf, dauer, text from uebungen;
   uebung_id    |               bezeichnung               | vorlauf  |  dauer   |      text
----------------+-----------------------------------------+----------+----------+----------------
 pruefung       | Verwalten von Benutzern mit Hilfe von NIS | 00:45:00 | 01:00:00 | pruefung.html
 pruefung2      | Verwalten von Benutzern mit Hilfe von NFS | 00:45:00 | 01:00:00 | pruefung2.html
 nfs            | Aufsetzen von NFS Client und Server       | 00:45:00 | 01:30:00 | nfs.php
 solaris        | Solaris konfigurieren                     | 00:45:00 | 01:30:00 | solaris.php
 nis            | Aufsetzen von NIS Client und Server       | 00:45:00 | 01:30:00 | nis.php
 netbsd         | NetBSD konfigurieren                      | 00:45:00 | 01:30:00 | netbsd.php
 update-solaris | Solaris-Image updaten                     | 00:45:00 | 01:00:00 | solaris.php
```

G.5 Table: uebung_setup

This table lists the machines and their images associated with a certain exercise.

```
CREATE TABLE uebung_setup (
        uebung_id varchar(40) NOT NULL,             -- exercise id
        rechner varchar(30) NOT NULL,               -- hostname
        image varchar(150) NOT NULL,                -- image filename
        CONSTRAINT pk_uebung_setup
                PRIMARY KEY (uebung_id, rechner),
        CONSTRAINT fk_uebung
                FOREIGN KEY (uebung_id)
                REFERENCES uebungen (uebung_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT fk_rechner
                FOREIGN KEY (rechner)
                REFERENCES rechner (bezeichnung)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT fk_image
                FOREIGN KEY (image)
                REFERENCES images (bezeichnung)
                ON DELETE CASCADE
                ON UPDATE CASCADE
);
```

G.6 Table: uebungs_checks

This table contains a list of checks to make at the end of a certain exercise. The "`parameter`" field is only available in implementation step II.

G.6.1 Definition

```
CREATE TABLE uebungs_checks (
        check_id serial NOT NULL,                   -- check id
        uebung_id varchar(80) NOT NULL,             -- associated exercise
        script varchar(150) NOT NULL,               -- which script to run
        parameter varchar(300),                     -- parameters for script (Step II only!)
        rechner varchar(30) NOT NULL,               -- where to run script
        bezeichnung varchar(150) NOT NULL,          -- text for feedback
        PRIMARY KEY (check_id),
        UNIQUE (check_id),
        CONSTRAINT fk_uebung
                FOREIGN KEY (uebung_id)
                REFERENCES uebungen (uebung_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT fk_rechner
                FOREIGN KEY (rechner)
                REFERENCES rechner (bezeichnung)
                ON DELETE CASCADE
                ON UPDATE CASCADE
);
```

G.6.2 Example records

```
vulab=> select * from uebungs_checks where uebung_id='netbsd';
 check_id | uebung_id |          script           |               bezeichnung                | rechner
----------+-----------+---------------------------+------------------------------------------+--------
      909 | netbsd    | netbsd-check-installed-pkg | bash installiert? (pkg_info -e bash)     | vulab1
      910 | netbsd    | unix-check-user-exists     | Benutzer angelegt? (getpwnam(3))         | vulab1
      911 | netbsd    | unix-check-user-home       | Home-Directory richtig gesetzt?          | vulab1
      912 | netbsd    | unix-check-user-shell      | Shell auf tcsh gesetzt? (getpwnam(3))    | vulab1
      913 | netbsd    | netbsd-check-user-shell    | Shell auch in /etc/master.passwd gesetzt? | vulab1
      914 | netbsd    | unix-check-user-password   | Passwort richtig gesetzt? (getpwnam(3))  | vulab1
      915 | netbsd    | unix-check-user-shell      | Shell des Users vulab auf bash gesetzt?  | vulab1
      908 | netbsd    | admin-check-clearharddisk  | tcsh installiert? (pkg_info -e tcsh)     | vulab1
(8 rows)
```

G.7 Table: buchungen

This table contains entries for exercises actually booked by users, including time and date of the exercise and which exercise to practice.

G.7.1 Definition

```
CREATE TABLE buchungen (
        buchungs_id serial NOT NULL,                -- booked exercise id
        user_id int NOT NULL,                       -- for which user
        uebung_id varchar(40) NOT NULL,             -- which exercise
        datum date NOT NULL,                        -- when/date
        startzeit time NOT NULL,                    -- when/time
        freigegeben varchar(30) DEFAULT 'nein' NOT NULL,  -- exercise set up?
        endzeit time,                               -- when/ended
        at_id int,                                  -- setup at(1) job id
        at_id_end int,                              -- uebung_ende job id
```

```
        ip varchar(20),                             -- where user came from
        PRIMARY KEY (buchungs_id),
        UNIQUE (buchungs_id),
        CONSTRAINT fk_uebung
                FOREIGN KEY (uebung_id)
                REFERENCES uebungen (uebung_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT fk_user_id
                FOREIGN KEY (user_id)
                REFERENCES benutzer (user_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE
);
```

G.7.2 Example records

```
vulab=> select * from buchungen;
 buchu- | user | uebung |   datum    | startzeit | freigegeben | endzeit  | at_id | at_id |       ip
 ngs_id |  _id |    _id |            |           |             |          | at_id | _end  |
--------+------+--------+------------+-----------+-------------+----------+-------+-------+-----------------
    114 |   33 | nfs    | 2004-05-03 | 15:00:00  | nicht-mehr  | 15:06:18 |   234 |   235 | 132.199.213.37
    115 |   33 | nis    | 2004-05-11 | 21:00:00  | nicht-mehr  | 23:30:05 |   236 |   237 |
    116 |   33 | nfs    | 2004-05-18 | 11:45:00  | nicht-mehr  | 12:05:21 |   238 |   239 | 194.95.108.21
    117 |   34 | nfs    | 2004-05-20 | 18:00:00  | nicht-mehr  | 20:30:05 |   241 |   247 |
    123 |   35 | nfs    | 2004-05-21 | 15:00:00  | nicht-mehr  | 16:20:57 |   248 |   250 | 194.95.108.32
    124 |   35 | nis    | 2004-05-21 | 21:00:00  | nicht-mehr  | 22:28:11 |   249 |   251 | 194.95.108.32
    122 |   37 | nfs    | 2004-05-22 | 12:00:00  | nicht-mehr  | 13:07:14 |   246 |   252 | 194.95.108.32
    120 |   35 | nfs    | 2004-05-22 | 15:00:00  | nicht-mehr  | 16:10:11 |   244 |   253 | 194.95.108.38
    121 |   35 | nis    | 2004-05-23 | 12:00:00  | nicht-mehr  | 12:51:23 |   245 |   255 | 194.95.108.38
    126 |   35 | nis    | 2004-05-23 | 15:00:00  | nicht-mehr  | 15:01:08 |   256 |   257 | 194.95.108.32
    130 |   34 | nfs    | 2004-05-25 | 21:00:00  | nicht-mehr  | 21:16:40 |   261 |   262 | 194.95.108.32
    127 |   37 | nis    | 2004-05-26 | 15:00:00  | nicht-mehr  | 15:11:11 |   258 |   263 | 194.95.108.38
    129 |   38 | netbsd | 2004-05-26 | 18:00:00  | nicht-mehr  | 19:29:04 |   260 |   266 | 82.83.169.114
    131 |   44 | nis    | 2004-05-27 | 18:00:00  | nicht-mehr  | 19:25:03 |   264 |   271 | 132.199.227.122
    134 |   43 | nfs    | 2004-05-28 | 12:00:00  | nicht-mehr  | 14:30:04 |   268 |   273 | 194.95.108.68
    136 |   38 | netbsd | 2004-05-28 | 21:00:00  | nicht-mehr  | 23:30:05 |   270 |   274 |
    138 |   38 | netbsd | 2004-05-29 | 12:00:00  | nicht-mehr  | 12:37:46 |   275 |   277 | 194.95.108.32
    128 |   35 | nis    | 2004-05-29 | 15:00:00  | nicht-mehr  | 16:28:02 |   259 |   280 | 194.95.108.32
    139 |   37 | nis    | 2004-05-29 | 18:00:00  | nicht-mehr  | 19:38:45 |   276 |   281 | 194.95.108.38
    132 |   35 | nis    | 2004-05-29 | 21:00:00  | nicht-mehr  | 22:25:28 |   265 |   283 | 194.95.108.32
    141 |   38 | netbsd | 2004-05-30 | 12:00:00  | nicht-mehr  | 14:30:04 |   279 |   284 |
    140 |   38 | nfs    | 2004-05-30 | 15:00:00  | nicht-mehr  | 17:30:04 |   278 |   285 |
    143 |   39 | nfs    | 2004-05-31 | 21:00:00  | nicht-mehr  | 23:30:03 |   286 |   287 |
    144 |   38 | nfs    | 2004-06-01 | 21:00:00  | nicht-mehr  | 22:27:26 |   288 |   289 | 194.95.108.32
    142 |   37 | nis    | 2004-06-02 | 15:00:00  | nicht-mehr  | 16:13:50 |   282 |   290 | 194.95.108.38
    137 |   44 | nis    | 2004-06-02 | 18:00:00  | nicht-mehr  | 20:30:04 |   272 |   291 | 132.199.227.122
    146 |   44 | nis    | 2004-06-02 | 21:00:00  | nicht-mehr  | 23:30:03 |   293 |   294 | 132.199.227.122
    147 |   38 | nfs    | 2004-06-03 | 12:00:00  | nicht-mehr  | 13:36:55 |   295 |   296 | 194.95.108.132
    149 |   50 | nis    | 2004-06-07 | 15:00:00  | nein        |          |   298 |       |
    145 |   44 | nis    | 2004-06-03 | 18:00:00  | nicht-mehr  | 18:34:36 |   292 |   300 | 132.199.227.122
    150 |   48 | nis    | 2004-06-03 | 21:00:00  | nicht-mehr  | 22:24:17 |   299 |   302 | 132.199.227.122
    155 |   44 | nis    | 2004-06-07 | 21:00:00  | nein        |          |   306 |       |
    152 |   33 | nfs    | 2004-06-04 | 09:00:00  | nicht-mehr  | 11:30:05 |   303 |   307 |
    153 |   33 | nis    | 2004-06-04 | 12:00:00  | nicht-mehr  | 14:30:17 |   304 |   308 | 194.95.108.65
    156 |   38 | nis    | 2004-06-05 | 00:00:00  | nicht-mehr  | 01:01:08 |   309 |   310 | 194.95.108.32
    157 |   34 | nfs    | 2004-06-07 | 18:00:00  | nein        |          |   311 |       |
    154 |   48 | nis    | 2004-06-07 | 09:00:00  | nicht-mehr  | 10:29:55 |   305 |   312 | 132.199.227.122
    158 |   48 | nis    | 2004-06-08 | 12:00:00  | nein        |          |   313 |       |
    159 |   44 | nfs    | 2004-06-08 | 15:00:00  | nein        |          |   314 |       |
    148 |   50 | nfs    | 2004-06-07 | 12:00:00  | nicht-mehr  | 13:29:05 |   297 |   315 | 194.95.108.159
    160 |   52 | nfs    | 2004-06-08 | 21:00:00  | nein        |          |   316 |       |
    161 |   34 | nis    | 2004-06-10 | 15:00:00  | nein        |          |   317 |       |
(42 rows)
```

G.8 Table: `ergebnis_checks`

This table lists the results from the checks belonging to a certain booked exercise.

G.8.1 Definition

```
CREATE TABLE ergebnis_checks (
        buchungs_id int NOT NULL,                   -- booked exercise id
        check_id int NOT NULL,                      -- check id
        erfolg boolean NOT NULL,                    -- result
        CONSTRAINT pk_ergebnis_checks
                PRIMARY KEY (buchungs_id,
                        check_id),
        CONSTRAINT fk_check_id
                FOREIGN KEY (check_id)
                REFERENCES uebungs_checks (check_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE,
        CONSTRAINT fk_buchungs_id
                FOREIGN KEY (buchungs_id)
                REFERENCES buchungen (buchungs_id)
                ON DELETE CASCADE
                ON UPDATE CASCADE
);
```

G.8.2 Example records

```
vulab=> select * from ergebnis_checks where buchungs_id=129;
 buchungs_id | check_id | erfolg
-------------+----------+--------
         129 |      908 | t
         129 |      909 | f
         129 |      910 | f
         129 |      911 | f
         129 |      912 | f
         129 |      913 | f
         129 |      914 | f
         129 |      915 | f
(8 rows)

vulab=> select buchungs_id, ergebnis_checks.check_id, bezeichnung, erfolg
vulab-> from ergebnis_checks,uebungs_checks
vulab-> where buchungs_id=129 and ergebnis_checks.check_id=uebungs_checks.check_id;
 buchungs_id | check_id |                       bezeichnung                       | erfolg
-------------+----------+---------------------------------------------------------+--------
         129 |      908 | tcsh installiert? (pkg_info -e tcsh)                    | t
         129 |      909 | bash installiert? (pkg_info -e bash)                    | f
         129 |      910 | Benutzer angelegt? (getpwnam(3))                        | f
         129 |      911 | Home-Directory richtig gesetzt?                         | f
         129 |      912 | Shell auf tcsh gesetzt? (getpwnam(3))                   | f
         129 |      913 | Shell auch in /etc/master.passwd (via vipw(1)) gesetzt? | f
         129 |      914 | Passwort richtig gesetzt? (getpwnam(3))                 | f
         129 |      915 | Shell des Users vulab auf bash gesetzt?                 | f
(8 rows)
```

References

[Cocke and Schwartz, 1970] Cocke, J. and Schwartz, J. T. (1970). *Programming Languages and their Compilers*. Courant Institute of Mathematical Sciences, New York Universities, New York, NY, USA.

[Feyrer, 2004] Feyrer, H. Virtuelles Unix Labor - Deployment der Übungsrechner [online]. (2004) [cited 2007-10-05]. Available from: `http://vulab.fh-regensburg.de/~feyrer/vulab/hubertf/deployment`.

[Feyrer, 2007] Feyrer, H. g4u - Harddisk Image Cloning for PCs [online]. (2007) [cited 2007-10-05]. Available from: `http://www.feyrer.de/g4u/`.

[Feyrer, 2008] Feyrer, H. (2008). System Administration Training in the Virtual Unix Lab. preprint.

[Momjian, 2000] Momjian, B. (2000). *PostgreSQL*. Addison Wesley, Boston, MA, USA.

[Spinellis, 2001] Spinellis, D. (2001). Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99.

[Stevens, 1992] Stevens, R. W. (1992). *Advanced Programming in the Unix Environment*. Addison Wesley, Boston, MA, USA.

[The Open Group, 2004] The Open Group (2004). *Single Unix Specification*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. IEEE Std 1003.1, 2004 Edition. Available from: `http://www.opengroup.org/onlinepubs/007904975/toc.htm` [cited 2007-10-05].

[Zimmermann, 2003] Zimmermann, S. (2003). Webbasiertes User-Management des Virtuellen Unix Labors. Technical report, Fachhochschule Regensburg, Computer Science Department.