Virtualized Environments for the Harness Workbench

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

Master of Science

In

Network Centered Computing,
High Performance Computing

in the

Faculty of Science
The University of Reading

by

Bjoern Koenning

28.02.2007

Supervisors: Prof. Vassil Alexandrov, University of Reading
Christian Engelmann, Oak Ridge National Laboratory

1 Acknowledgment

In the first place, I would to thank Christian Engelmann for his great and all-embracing support during my stay in the USA and giving me his time and advice for my research at the Oak Ridge National Laboratory.

As well, thanks Stephen L. Scott and Christian for their invitation and the financial support.

Furthermore, merci and thanks to Geoffroy R. Vallee and Thomas Naughton for their kernel and Linux experiences.

Last but not least, thanks to Al Geist for showing me the magic side of the Laboratory.

This research is sponsored by the Mathematical, Information, and Computational Sciences Division; Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

2 Abstract

The expanded use of computational sciences today leads to a significant need of high performance computing systems. High performance computing is currently undergoing vigorous revival, and multiple efforts are underway to develop much faster computing systems in the near future. New software tools are required for the efficient use of petascale computing systems.

With the new Harness Workbench Project the Oak Ridge National Laboratory intends to develop an appropriate development and runtime environment for high performance computing platforms.

This dissertation project is part of the Harness Workbench Project, and deals with the development of a concept for virtualised environments and various approaches to create and describe them.

The developed virtualisation approach is based on the `chroot` mechanism and uses platform-independent environment descriptions. File structures and environment variables are emulated to provide the portability of computational software over diverse high performance computing platforms. Security measures and sandbox characteristic are integrable.

Contents

3 Introduction

The expanded use of computational sciences today leads to a significant need of high performance computing systems. High performance computing is currently undergoing vigorous revival. Spurred by science needs, technical advances, and national priorities, multiple efforts are underway to develop much faster computing systems in the near future.

These new systems, like current systems, will differ from each other in multiple aspects like architecture, hardware, interconnects, operating systems, compilers, library versions, runtime environments, debugging, and monitoring tools.

This diversification leads to two main challenges for scientists in the computational area. First, seamless portability of scientific programs is not given without tedious adaptation. And second, it is related to the fact that many programs are optimized to use a particular system. Portable program code is mostly not running efficiently on diverse systems. Therefore, computational scientists have to expend considerable time and effort dealing with development, deployment, and runtime interfacing activities. This burden results in inefficient use of hardware and human resources, and to a reduction of scientific discovery, e.g., it leads to higher costs.

Motivation of the Harness Workbench Project

With the new Harness Workbench Project [1], the Oak Ridge National Laboratory intends to develop an appropriate development and runtime environment for high performance computing platforms. The Harness Workbench will help to enhance the overall productivity of computational sciences on diverse high performance computing platforms by increasing the portability of scientific programs. It gives users unified and adaptive access to diverse high performance computing platforms and hides the complexity in the background.

The Harness Workbench consists of two main components. The first component is a virtualised command toolkit (VCT) for application building and execution that provides a common view across diverse HPC systems. It is a software backplane architecture, which presents a uniform, extensible interface, and is capable of interoperating with existing and third-party toolkits. It will interface to native platform functionality via plug-in modules that encapsulate vendor-specific knowledge and are further configurable at the system and application levels.

The second component of the Harness Workbench is a unified runtime environment that similarly consolidates access to runtime services via an adaptive framework for execution time and post processing activities.

The Harness Workbench Project addresses to the entire area in which substantial benefits can be derived by optimising the development and deployment process, and by facilitating software reuse.

Traditionally, programmers write code for a given runtime environment offering a defined set of functions and properties. The runtime environment itself can be seen as a layer between operating system and application, which emulates a unified environment on diverse operating systems.

A problem of this approach is that applications running in a runtime environment may experience a loss of performance. Other problems are caused by the fact that a lot of runtime environments are not available for all platforms, or it is not possible to make use of the full performance with them. Thus, by accepting a loss of portability many applications running on supercomputers are written for one specific supercomputer to achieve a maximum of performance.

The previous efforts in research of Harness tries to provide a unified runtime environment on supercomputers via pluggable runtime environment software modules.

It is the intention of the Harness Workbench Project to solve these problems by using a new way to ensure portability. The run-ability of an application on another supercomputer can be facilitated by using the Harness Workbench, which creates appropriate virtual environments required by applications. Therefore, Harness will simplify application development, deployment, and portability.

The crucial difference between the Harness Workbench approach and traditional runtime environments, like OpenRTE, and previous variants of Harness is based on its ability to create different virtual environments. It provides an automated adaptation of virtual environments according to the requirements of each specific program.

In other words, not the programmer has to fit the code to a given runtime environment standard, but the operating system with the help of Harness has to fit to the application.

This dissertation project is a part of the Harness Workbench effort. The overall software development of the entire life cycle of an application is considered, while traditionally, efforts for accelerating computing processes are only addressed to faster hardware, faster algorithms, and more optimized (lightweight) operating systems.

The objective of the Harness Workbench is to simplify software development and deploy-

ment by making scientific application software more portable using appropriate tools and runtime environments.

3.1 Project Description

As mentioned, the crucial difference between the Harness Workbench and traditional runtime environments is based on its ability to create different appropriate virtualised environments for specific applications. This dissertation project is a first step to enable the Harness Workbench to create such appropriate virtualised environments.

In opposite to traditional runtime environments, the Harness Workbench will create an environment, which is not fixed to a standard environment with a defined set of properties. It will create appropriate environments according to a given environment description based on application needs.

Therefore, this dissertation project deals with the development of a concept for virtualised environments and various approaches to create and describe them.

The development of a description concept for these virtualised environments represents the main part of the theoretical basis of this dissertation project. This description concept can be used as an approach-independent information format for software environments. This means other application in addition to Harness will use this concept.

The practical part of this thesis deals with the development and tests of tools for creating virtual environments according to a given description. Especially, the portability of such tools is important.

The abstract workflow starts with writing an application on an arbitrary computing system. If it is required to port the application to continue the work on another system, it is required to describe the original environment in an environment description. Finally, provided that Harness is available on the new system, it will create a new virtualised environment, in which the application can be compiled and executed, see Figure 3.1.

3.2 Key Problems and Specifications

After considering the projects main objectives, the following specifications and key problems can be summarized.

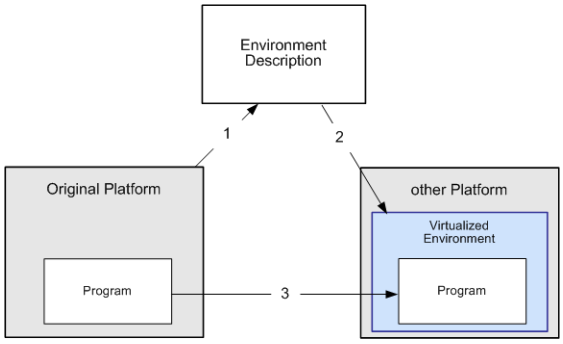

Figure 3.1: Abstract Situation

3.2.1 Virtualisation Approach Specifications

Portability

The main goal of the entire virtualisation approach is the portability of software. This can be achieved by emulating an environment in which a program can find their required functionality to be runnable. Normally, virtualised environments are emulated by runtime environment software which can be seen as a layer between the operating system and the application. The properties of these runtime environments are more or less fix defined. A software cannot be ported to such traditional environments if the original software environment is not equal or similar. Therefore, the required environment has to be described in a way that allows recreating it in a virtualised environment. This description contains all required information needed to create the virtualised environment on another operating system. For these environment descriptions a concept is needed.

In addition to ensure the portability of software over various operating systems, the approach itself has to be portable as well and should be runnable on almost all Unix-type operating systems.

Portability means as well that the virtualisation application can be easily installed or integrated in an operating system. Hence, kernel modifications cannot be required.

Furthermore, the portability of the virtualisation approach is also valid for operating systems of modern high performance computing platforms, as described in Section 3.3.1.

Portability to the Mircrosoft Windows operating system is not intended because it is not used for high performance computing [2]. However, in theory, portability to these systems could be realized with use of Cygwin [3]. Cygwin offers a Unix shell-like environment on Microsoft Windows platforms.

Applications for Creating and Starting Virtal Environments

Furthermore, command line based applications are required to create appropriate virtualised environments and to start given scientific applications in them. These command line based applications should provide suitable command line options for convenient use and process all the information read from an environment description file. As mentioned the entire dissertation should lead to an enrichment of the Harness Workbench. Therefore, these applications have to be integrated as part of the Virtual Command Toolkit (VCT) of Harness. It is intent to provide commands as noticed below.

```
vct -env install myDescr.conf
vct -env start myDescr.conf myApp
```

While the first one installs an environment according to the description file myDescr.conf the second line shows a command which starts the program myApp in the environment described in myDescr.conf.

Within the scope of this dissertation project, the creation of virtual environments is focused on file structures and environment variables. Currently, the Harness Workbench research at the Oak Ridge National Laboratory is not concentrated on package management, services, or network related issues.

Merged Virtual File Structures

For efficient and flexibility reasons the virtualisation of file structures should be performed by merging both, new files and local files already existing on the target system. This will save disk space and allows arbitrary combinations.

Virtual Environments in User-Space

The capability to create user-space-based virtual environments should be mentioned as a more specific purpose, see Figure 3.2. This feature could be very helpful to simplify system administration of supercomputers.

In a traditional high performance computing development and execution model, system administrators are solely responsible for system-wide installations of supporting software and scientific libraries. Users write, compile, and run codes that utilize these resources. While this model works perfect for general-purpose software components, like vendor-optimized MPI implementations, it is less appropriate for libraries that are unconventional and more problem specific. For example, in situations when only a few scientists in an organization need some packages, it might be more reasonable to enable them to perform user-specific installations in their home directories rather than placing that burden to on-site administrators. It is possible to use multiple library versions without conflicts by installing each version in a different virtual environment.

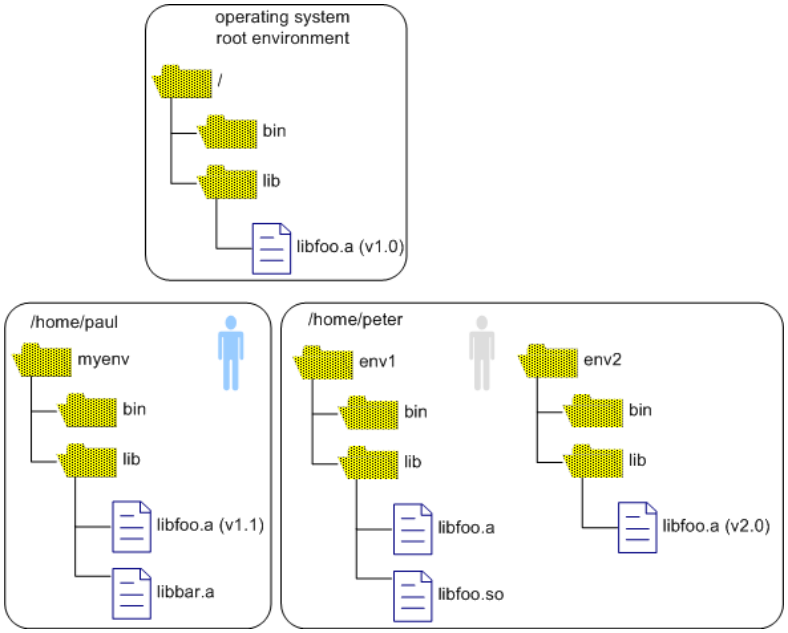

Figure 3.2: Virtualised Environment and User Space

Master Environemt Scope

The virtualisation approach has to work in a way that the system-level scope of an application running in a virtual environment can itself be considered as a master environment. For a program running in a virtual environment, the root of the file structure in a virtual environment should look like the original file structure root.

Security

To avoid security vulnerabilities a secure approach is needed. The virtualisation approach has to work without giving normal users special access rights. But it has to enable normal users to start applications in user-space-based virtual environments. Otherwise these users would be able to run programs with super user rights on their own virtual environment, being capable to destroy the entire system.

Support Shell-Less Systems

As described in Section 3.3.1, there are various high performance computing platforms with lightweight operating systems without any kind of Unix-type shell. In this case shell script-based applications cannot work. For supporting also such systems, it is useful to implement the functionality of the virtualisation approach in libraries. In this case a computational application itself has to change into a particular virtual environment by including these libraries and using their functionality. Reading of the description file and creating the virtual environment has to be performed in the same way.

Administration Support

Apart from normal users creating their own user-space-based virtual environment, a system administrator should be allowed to exploit these virtual environments. This could be beneficial in two situations.

First, if a normal user needs help from a system administrator for a complex installation in their virtual environment to complete it. And second, if a system administrator modifies libraries or files which were already integrated in a virtual environment it is, in some cases, necessary to modify the affected virtual environments to avoid security vulnerabilities.

3.2.2 Description Concept Specifications

Flexibility

As mentioned, the core of the theoretical aspect of this dissertation project is a description concept for virtual environments running on various high performance computing platforms.

This description concept should be as flexible as possible to allow all-embracing descriptions of environments. It should allow arbitrary combinations of environmental properties, like environment variables, library paths, package dependencies, and device files.

Extensibility

The description concept should be as flexible as possible to allow future extensions. This will make the concept powerful for all-embracing descriptions of nearly all thinkable combinations of existing and new environmental properties like new file permissions, library and package management, and configurations of specific services and network identities.

Therefore, the concept should be constructed in a way that future extensions can be easily included without reconstructing existing concept elements. Otherwise such extensions will lead to nonuniform concept elements and makes old environment descriptions unusable.

Level of Detail

Furthermore, the concept has to provide a hierarchical scheme. This scheme offers different levels of detail to simplify the description process. furthermore, a hierarchical order can be used to implement an inheritance principle.

Inheritance

The ability of virtual environments to inherit from one another is a preferable feature, which is important for both, the technical implementation and the description concept. This feature provides an easier way to describe virtual environments with regards to security issues.

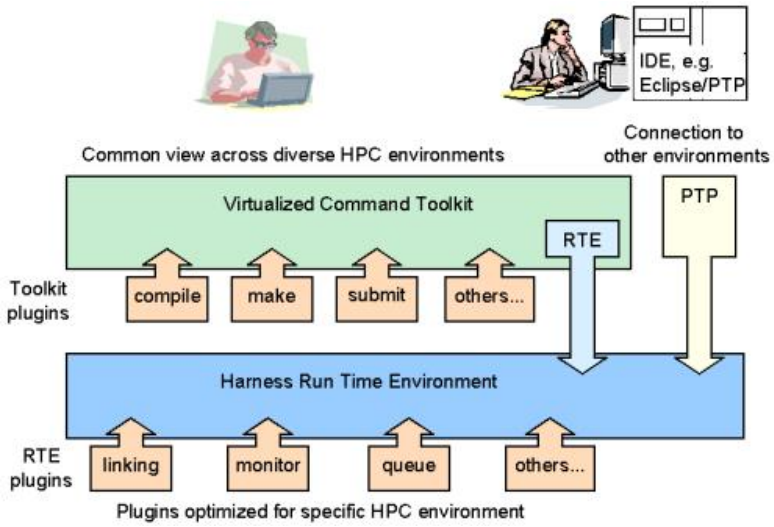

Figure 3.3: Inheritance of Environment Descriptions

XML

The concept has to be developed as a XML dialect. XML is the most widespread and supported meta language. XML is easier to develop and process than other meta languages and offers sufficient functionality for the implementation of the mentioned concept related requirements.

Usability

Usability deals with the question how convenient the concept can be used in practice. To enable a user to modify such XML-based environment descriptions, it is required that it is a human readable XML format and follows simple syntax rules.

3.2.3 Software System Strategy and Milestones

The outlined requirements of the abstract virtualisation approach and its used environment description concept lead to the following project strategy.

As shown in Figure 3.4, it is planned to implement two applications: one application for the creation of the virtualised environment, the veCreator (virtual environment creator).

And another one for starting applications in virtualised environments, veStarter (virtual environment starter). These applications are implemented as shell scripts.

As a consequence to the mentioned requirement the concept has to be implemented as an XML [4] dialect. This requirs XML parsers, like Expat [5], for reading, validating and interpreting environment description files. As an optimal solution these XML parsers should be used to transform the XML concept description in a shell-script-processable file format, called config file. This transformation can be described via XSLT [6]. Such config files can be read very comfortable by shell script applications.

After these achievements two preferable extra goals can be formulated to sophisticate the prototype more towards a production-type tool. For a convenient use of the description concept it is required to have a possibility to validate a created concept against syntax rules. The second additional goal is connected with the mentioned requirement to provide the virtualisation approach on shell less operating systems. The entire approach can be re-implemented as a library.

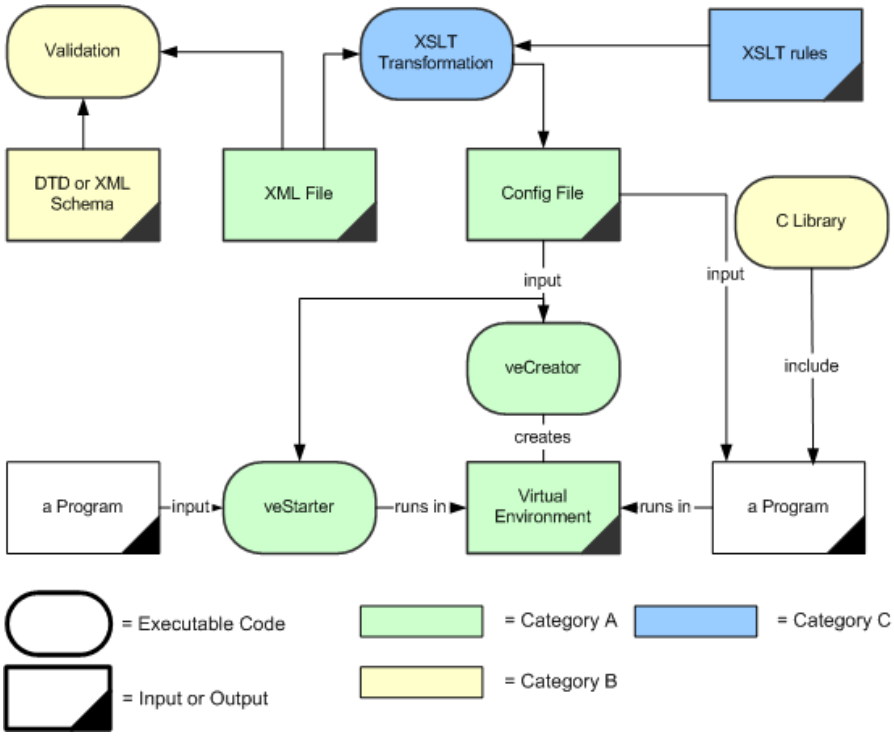

Figure 3.4: Milestones

In order to fulfill and achieve the described key problems, specifications and this project strategy, a dissertation project plan has to be organized. There are three different milestones categories, which outline the project development status:

Milestone Category A: A minimum set of the requirements and task will be fulfilled. The achievement of this category leads to a useful concept and a good working prototype which shows the feasibility of the concept and the integration of Unionfs [7].

Milestone Category B: This category includes an optimal set of requirements and tasks. The achievement of this category results to more sophisticated description concept and more functionality of the prototype application.

Milestone Category C: This category includes some extra project features, which will provide the validation of the description concept and will offer a library for shell-less operating systems.

The following table shows which tasks and requirements are involved in the three milestone categories.

Required Capability	**Milestone**
Preliminary description concept for file structures and environment variables	A
Defining notation rules for the config file format	A
Shell script application veCreator	A
Shell script application veStarter	A
Installation and configuration of Unionfs	A
Enable veCreator and veStarter to use Unionfs	A
Performance tests	A
Enhance description concept especially security features	B
XSLT Transformation description for XML to config file format	B
XSLT implementation by using an existing parser	B
XML Schema definition for validation of environment descriptions	C
Re-implementation of the shell scripts as a C library	C

Table 3.1:

3.2.4 Project Advancements

The achievements of this dissertation project lead to a portable virtualisation approach which is easy to integrate. It will benefit application development and deployment by increasing the portability of software, especially for scientific computation on modern high performance computing systems. Another important advancement is addressed to a simplified administration of supercomputers via installation in user-space. Finally, the most important advancement is a powerful description concept for virtualised environments, which will be used by the Harness Workbench and other virtualisation and software de-

velopments tools.

3.3 Background

At first a short description of the abstract architecture of modern high performance computing platforms is explained to give a better understanding of the advantages and disadvantages of related projects.

3.3.1 Modern Architecture of High Performance Operating Systems

The abstract architecture, which is described in this section, is not valid for all supercomputers. But it shows a high-level view of the latest design trend implemented, for instance, in the Cray XT3 and IBM Blue Gene/L which dominates in the Top 500 list of Supercomputers [2].

A Modern high performance computing system can have thousands of processors, called nodes. Currently the IBM Blue Gene/L with its 131.072 processors is the biggest and fastest supercomputer.

In the past, according to the original Beowulf cluster system architecture [8], the operating system of each node of a high performance computing system was designed for multi-purpose use. Each node runs a full operating system and a number of middleware services.

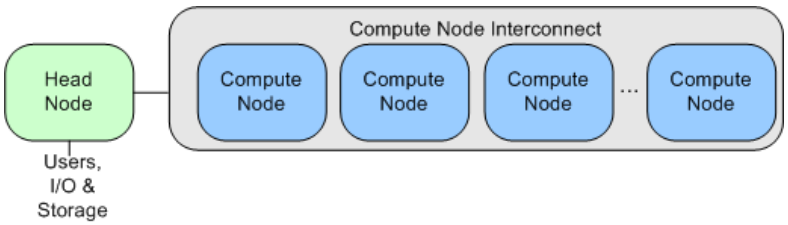

Figure 3.5: Beowulf [9]

The recent trend in modern high performance computing system architectures employs nodes with dedicated roles, each of them optimized to process its role more efficiently. Therefore, modern high performance computing systems can include the following node types:

- compute nodes,
- I/O nodes,
- front-end nodes,

- service nodes,
- file server nodes.

Compute nodes are dedicated to run application processes, whereas I/O nodes are dedicated to perform system functions. The operating system of a compute node supports often less then hundred system calls and has mostly no local file system. Service nodes run control software, which monitors all activities of the compute and I/O nodes as well power supplies and fans.

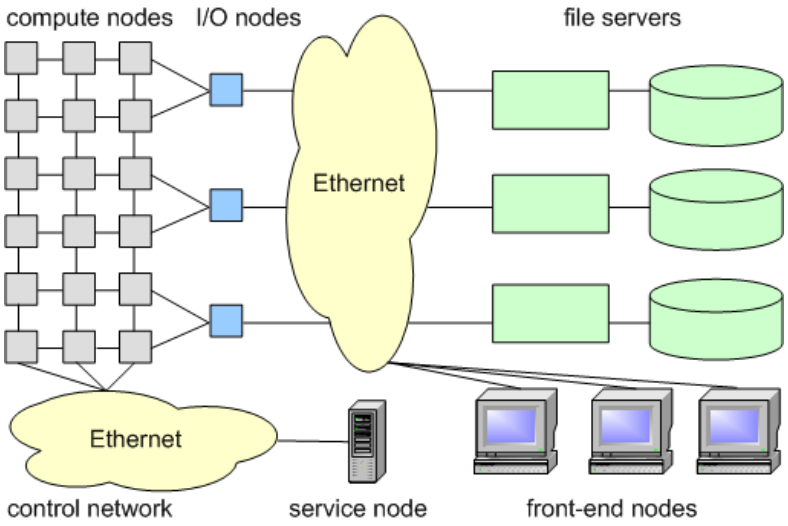

Figure 3.6: Modern High Performance Operating System Architecture

Due to the principle that the structure of the software should reflect the structure of the hardware [10], compute nodes run an adopted lightweight operating system. Certain parts of the operating system as well as other system software services are moved to I/O and service nodes in order to increase performance and scalability.

This architecture modification increases the scalability of such big systems. The functional separation avoids interferences caused by system functions. Interferences are the main problem for scalability.

However, such systems are accessed by the user via a front-end node. Depending on the concrete realization of the system, the user can compile its application direct on the front-end or the compilation task is submitted to a specialized service node. Furthermore from the front-end node the user can initialize the execution of an application. The application code is submitted to a number of compute nodes, where the code is executed.

As illustrated in Figures 3.7 and 3.6 the application runs in a Runtime Environment (RTE) [11] which is responsible for the communication between compute nodes,for in-

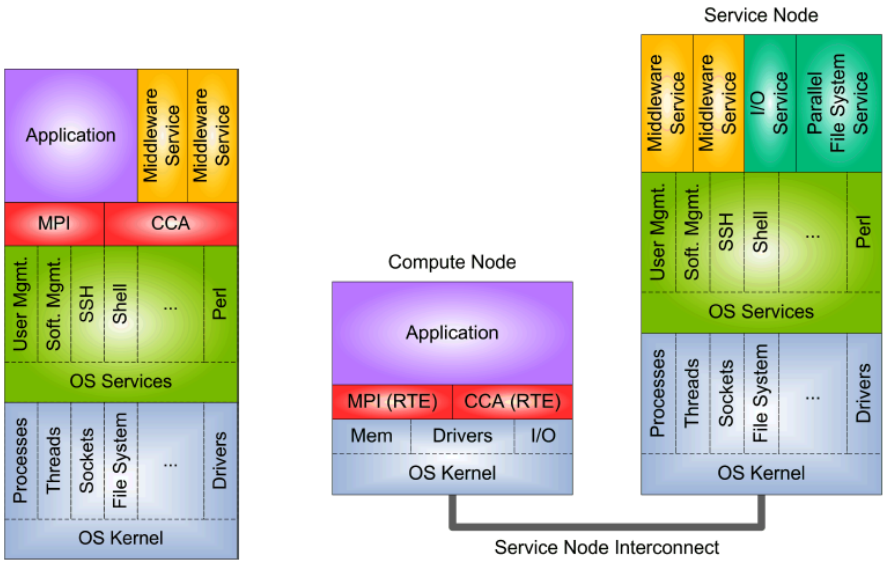

Figure 3.7: Traditional(a) and Modern Compute Node Software Architecture(b) [9]

stance via MPI [12]. While front-end and mostly service nodes too, run a full Unix-type operating system, the kernel on the compute node has not to be, and mostly is not, an Unix-type kernel.

Thus, compilation on such systems means cross-compilation of application code written for Linux to the kernel of the compute nodes. According to Figure 3.8, a Linux system call library has to be available to support Linux system calls. For instance, if a running application on a compute node invokes the I/O system call read which requires the infrastructure of the file system, the compute node kernel ships it to its assigned I/O node and waits for reply. The I/O node reissues the system call and connects to a file server node to get access of the specific file and sends the data to the compute node.

The described architecture modifications must have an influence to the upper software stack particularly on compute nodes. The middleware concept of adding missing operating system features between operating system and application as well the role and architecture of middleware in modern high performance computing systems needs to be revisited. The result is a paradigm shift in high performance computing middleware design, where single middleware services are moved to service nodes, while runtime environments continue to reside on compute nodes [9].

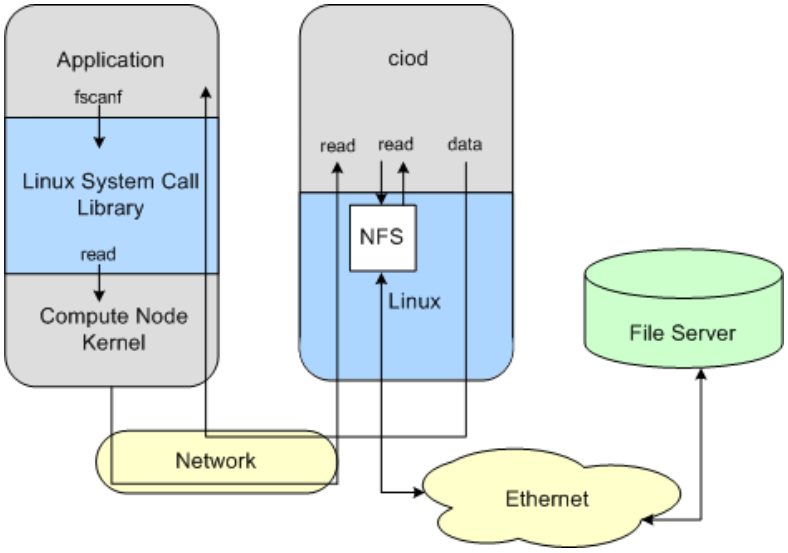

Figure 3.8: Function Shipping

3.3.2 Emulation versus Simulation

In literature is seems to be not clear what the difference is between emulation and simulation. Often, both terms are used misleadingly as synonyms.

Emulation refers to the ability of a system to imitate another system [13]. Emulation done by the emulating system returns or leads to the same result as the emulated system. Both systems produce the same results via different ways.

For the term simulation [14] there is no clear definition. There are several none-complete definitions but there is commonness in all of them. Simulation can be described as an abstract imitation of a system. The simulating system uses mostly a model of the simulated system to return or produce results closely to the original results. A model is an abstract description of a (usually more complex) system. Simulations are often used to estimate, assume or predict complex systems. Mostly these systems are too complex for understanding, for instance weather forecast.

Depending on the point of view a system imitating another can be considered as both simulation and emulation. For instance, a software system which imitates another one can calculate exactly the same results. Therefore it can be considered as emulation. But it can also be seen as a simulation if the different amount of time is involved in the consideration which is taken by the emulation.
Therefore, software emulation can be seen as a simulation if the emulation is slower or faster than the original and time constraints are a focused issue. This is not the case in the context of this dissertation project.

Furthermore, emulation can be seen as a special kind of simulation. Each system which imitates another system in a way which has to be considered as perfect under the specific conditions is an emulator.

This dissertation project concentrates on portability and functionality of software systems. If a virtual environment enables an application to run and produce the same results as on their original environment, it can be seen as an emulation of the original environment.

Therefore, the term emulation instead of simulation is used in the following.

3.3.3 Virtualisation

Virtualisation [15] is a broad term that refers to the abstraction of computer resources. It is a technique for hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. This includes making a single physical resource such as a server, an operating system, an application, or storage device appear to function as multiple logical resources; or it can include making multiple physical resources such as storage devices or servers appear as a single logical resource.

The common theme of all virtualisation technologies is the hiding of technical detail through encapsulation. Virtualisation creates an external interface that hides an underlying implementation, e.g., by multiplexing access, by combining resources at different physical locations, or by simplifying a control system.

Virtualisation can be realized from hardware level, from operating system level and from application level. Along these virtualisation levels the following terms can be classified. Depending on what should be virtualised two terms can be noticed: platform virtualisation and resource virtualisation. Platform virtualisation involves the emulation of a complete computer environment (virtual machine) created by a host software running on a given hardware platform, while resource virtualisation involves only the emulation of combined, fragmented, or simplified resources.

The following terms related to platform virtualisation can be listed. The terms listed below, are related to several approaches to perform platform virtualisation. They differ in the completeness of the environment they emulate and on the virtualisation level being built on.

Emulation

Emulation in this context means a virtual machine, which emulates the complete hardware. Hence unmodified guest operating system can be installed on it. The emulated

hardware can totally differ from the physical hardware of the machine. That is why it is possible to test the cooperation between a guest operating system and a new not-yet-built CPU during the construction phase. Examples for such full emulators are the PPC version of Virtual PC [16], QEMU [17] without acceleration.

Native Virtualisation and Full Virtualisation

Native virtualisation [18] and full virtualisation [19] can be realized by a virtual machine which emulates enough hardware to allow an unmodified guest operating system to be run in it. This approach differs from the above in the fact that the CPU which is emulated is the same type like in the physical machine. Typically, many instances can be run at once and each is isolated from the other ones. Examples are Virtual Iron [20], VMware Workstation and Server [21], Mac-on-Linux [22], and Win4Lin Pro [23].

Partial Virtualisation

A virtual machine [24] emulates multiple instances of a lot but not all of the underlying hardware. Such an environment supports resource sharing and process isolation, but does not allow separate guest operating system instances [25].

Paravirtualisation

Paravirtualisation [26] means that the virtual machine does not necessarily emulate hardware, but instead or in addition offers a special API that can only be used by modified guest operating systems. The guest operating system is modified to invoke instead of system calls the same named functions of a special API. These API functions itself invoke specific systems calls, named hypercalls, of the underlying virtual machine, and often named hypervisor. Examples are Xen [27], Parallels Workstation [28], Enomalism [29] and VMware Server, as well Win4Lin 9x [23].

Operating System-Level Virtualisation [30]

In contrast to all the above terms this approach does not emulate a guest operating system which runs isolated. The guest operating system environments share the same operating system as the host system, i.e., the same operating system kernel is used to implement guest environments. This term addresses to the virtualisation of multiple isolated and secure virtualised servers to run on a single physical server. An application runs on the host operating system but has another view on it, so it works as if it runs in an another guest operating system. The application cannot notice that it runs on the host operating system. Examples are Linux-VServer [31], Virtuozzo [32], and FreeBSD Jails [34].

Application Virtualisation

Application Virtualisation [33] is an approach which runs a desktop or server application locally, using local resources, within an appropriate virtual machine. This is in contrast with running the application as conventional local software, i.e., software that has been "installed" on the system. Such a virtualised application runs in a small virtual environment containing the components which are needed to execute, such as registry entries, files, environment variables, user interface elements, and global objects. This virtual environment acts as a layer between the application and the operating system, and eliminates application conflicts and application-OS conflicts. Examples include the Sun Java Virtual Machine [35], Thinstall [36], Altiris [37], and Trigence [38].

In practice, apart form these term definitions, the core of each virtualisation approach can be seen very abstractly in form of function mapping.

System or function calls are invoked from an upper tier in the software stack and will be mapped to another system or function call in the underlying tier. For instance, the usage of included C library for I/O operations is the easiest way of function mapping and can be considered as static. This is also valid for paravirtualisations when a modified operating system uses a library offering the functionality of a specific kernel.

Runtime environments can be seen as a layer which works as a configurable function mapping approach. Even the interpretation of Java byte code in a Java runtime environment is a function mapping.

4 Preliminary System Design

4.1 Detailed Project Strategy

As shown in Figure 3.4, two applications are implemented. The veCreator, for the creation of the virtualised environment and the veStarter, for starting applications in virtualised environments. These applications are implemented as shell scripts. The shell scripts are very convenient for prototype implementations. The program code can be quickly adapted and a powerful set of shell functions can be used very easily. Shell script implementations fit also to the portability requirements. A shell script application can be very portable of over very different operating system if it is written for a widespread used Unix-type shell and if the code uses standard functions.

According to the mentioned specificatoins the environment description concept has to be implemented as an XML dialect. This requires the use of an XML parser for reading, validating and interpreting the concept.

This causes two problems. First, if XML parsers are needed to read the description file, what would happen if an operating system is not equipped with it. Second, the description file can be seen as a big set of parameters for the shell scripts applications.

There are two possibilities to read the input parameters from a file. The first possibility is writing an own shell-script-based parser. This has to be rejected because it is a big burden and it is very inefficient to implement an own parser during a changeful prototype phase. The second possibility is more convenient. Unix-type shells provide an own parser with the source command. This source command is able to read and parse config files. As shown below, it can be easily managed in two ways by using the source command or a dot before the file.

```
source myEnvironment.conf
. myEnvironment.conf
```

Therefore, an XML parser is needed, which, according to an XSLT description, transforms the concept from XML format into a shell-script-processable or config file format. XSLT is an extensible Stylesheet language developed to describe transformations from arbitrary XML dialects into other XML dialects as well into none XML dialects.

This intermediate step is the easiest way to enable a fast shell script prototyping. It also provides a solution for the first above mentioned problem. On lightweight operating sys-

tems providing no XML parsers the user is still able to create virtualised environments by using the config file version of an environment description. As a precondition the specific environment description file has to be transformed on another capable operating system. On such lightweight operation systems the user is able to create or modify environment descriptions, if he is familiar with the use of this config file format. Therefore, the config file format has to be human readable like the XML format.

It is assumed that in future, such lightweight operating systems providing only a narrow set of functionality will be used more often, especially in face to operating systems created and virtualised by Xen.

4.2 Basic Virtualisation Approach

This section begins with a short abstract description of what should be virtualised, the environment of software. It describes the virtualisation of file structures and environment variables and critically discusses the outlined approach especially in face of the requirements of modern high performance computing systems.

4.2.1 Application Environments

The term environment has to be defined from the point of the applications. Software may require a file structure for I/O operations, the existence of appropriate compilers and libraries for compiling and running, environment variables to get information about the environment, a shell, devices, and installed services and network connections.

The application environment on the same operating system can differ depending on from where an application is started. An application specially written for a standard runtime environment like Java can only be started in such a runtime environment. In this case, the applications environment is the runtime environment and all the functionality of the underlying operating system is mainly used indirectly via its interfaces, see 4.2.

The traditional way is starting an application inside a shell. In this case, the shell is the direct application environment, as illustrated in Figure 4.1. The application uses the operating system properties directly and in addition it can use the environment variables and basic service programs of the shell.

On some modern high performance computing systems applications are started in a shell-less environment. There is no runtime environment and no shell, see Figure 4.3. The application is started remote from a service node and runs in an environment without shell environment variables and shell programs.

The last two cases are addressed by this dissertation and the following virtualisation approach.

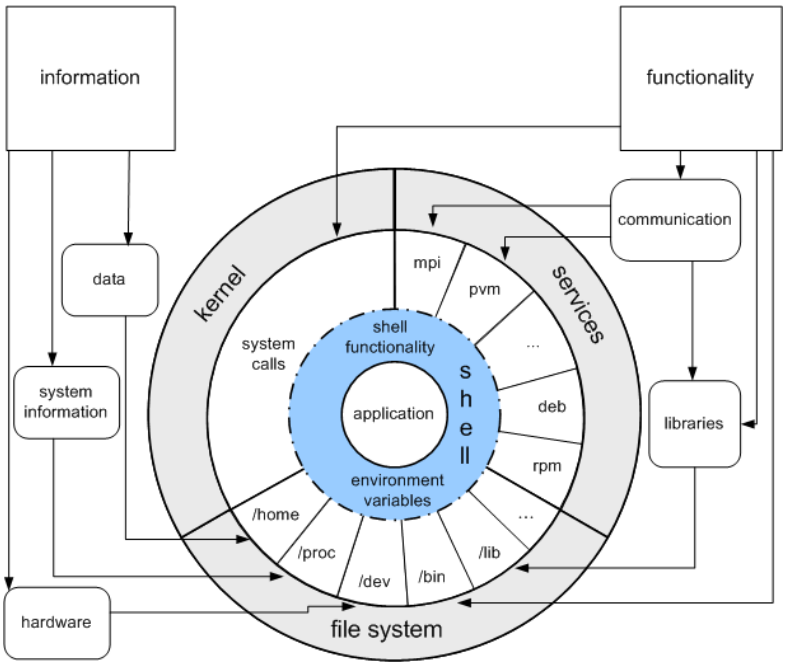

Figure 4.1: Application Environment with Shell

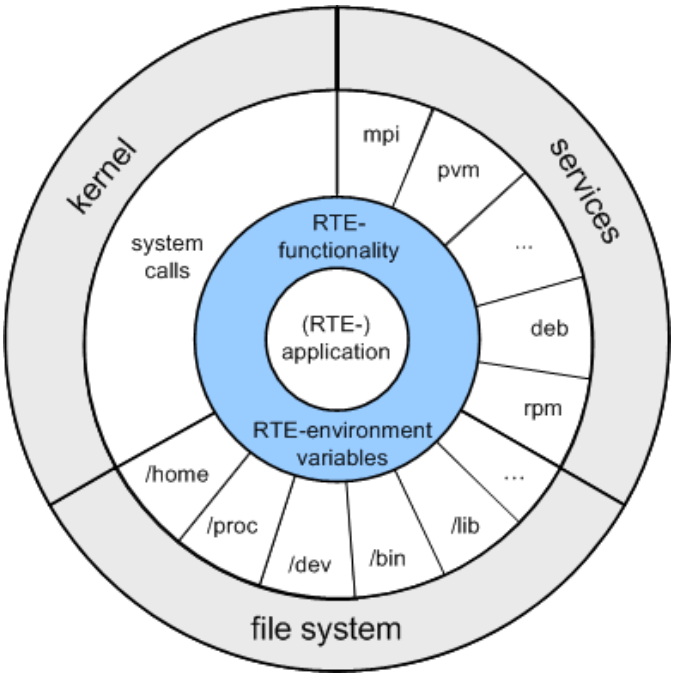

Figure 4.2: Application Environment with RTE

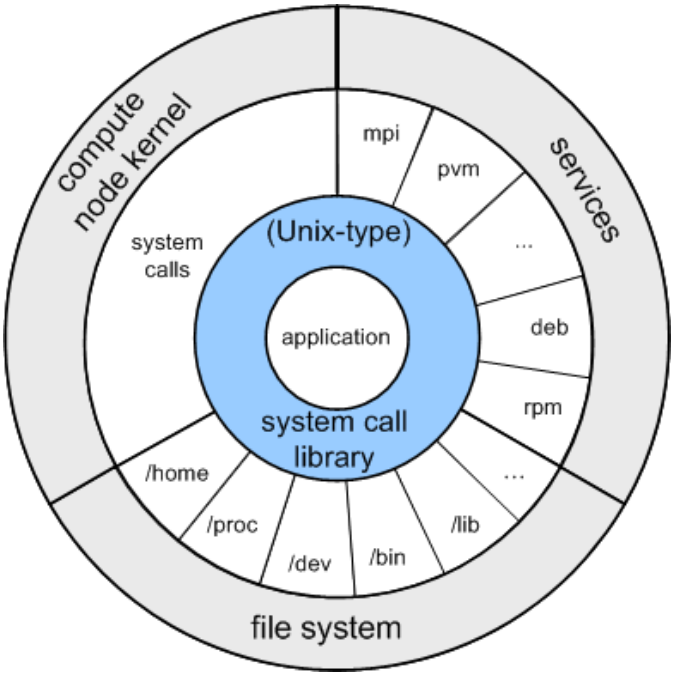

Figure 4.3: Application Environment on a Shell-less OS

4.2.2 Abstract Virtualisation of Environment Variables

The virtualisation of environment variables is required to emulate the software environment of applications which are started in a shell. In this dissertation the emulation of environment variables is achieved by starting a new shell and modifying the environment variables of this shell until it fits to the requirements of a specific application. Than the scientific application is executed in the new modified shell.

The use and data structure of environment variables is quite similar on all widespread Unix-type shells. The mostly used and widespread Unix-type shells [39] are the bourne shell(sh), the bourne again shell (bash), the korn shell (ksh), and the C-shell (csh).

For a shell, each variable is not more than a sequence of characters, a string data type. The command lines below show in line 1 a common way to initialize a variable and in line 2 how to use or print a variable's content.

```
export PATH=/bin
echo $PATH
```

It is possible to develop a portable approach, which creates environment variables in the mentioned shells.

Most scientific applications running in one of the mentioned shells are using only the environment variables and some service programs provided by the shell. The basic set of shell-provided service programs is the same for the mentioned shells.

The most scientific applications can be started in a manipulated shell on all Unix-type operating system using shells. There is still one of the mentioned shells available.

A shell script application cannot be started in a shell if both are not compatible. Then it is required a compatible shell on the operating system or create a virtual file structure, which includes the required shell. After that the required shell can be executed and manipulated and the shell script application can be started.

However, this case can be disregarded because shell script applications are to slow and therefore they are not often used for computational core tasks.

The emulation of specific shells cannot be achieved on some modern shell-less operating system. For instance, the Catamount operating systems used on the Jaguar Cray XT3 [40] at the Oak Ridge National Laboratory do not provide basic system calls, like `fork`, which is used by a process to start a child process [41]. A shell is doing the same if an application is started in them. It uses `fork` to create a child process. This child process is an exact copy of the shell and includes all the environment variables. Then the child process loads the instruction code of the application starts it.

The approach described in this section is implemented as a shell script application which is called veStarter in the following.

4.2.3 Abstract Virtualisation of File Structures

The virtualisation approach is based on the use of the `chroot` Unix command [42] to emulate a virtual file structure. The `chroot` command changes the root directory of a given process into a different given directory. After this, a process can be called "chrooted" process. Other processes are not influenced by this operation.

Only child processes of a "chrooted" process will be influenced. Figure 4.4 shows the narrowed system level scope of a "chrooted" process. Files inside the new root directory and its subdirectories cannot be named by its original path. A "chrooted" process cannot name files located outside its new root directory.

The file test.conf illustrated in Figure 4.4, is originally located in /home/peter/test.conf. If this file should be opened by a text editor which was chrooted into the directory /home/peter the editor has to name the new path /test.conf. The original file path does not exist from the editor's point of view.

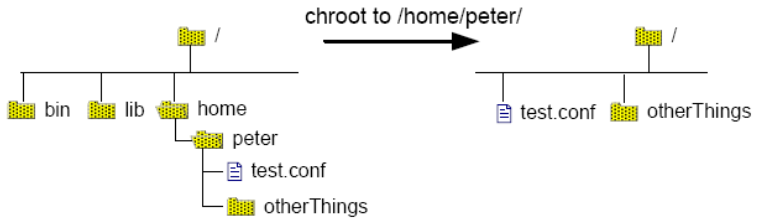

Figure 4.4: Original View and View of a chrooted Application

For this reason many chrooted applications cannot work as usual without modifications. Figure 4.5 illustrates the solving idea. The new root directory of the chrooted application includes an emulated file structure equivalent to the original one. It comprises all files required by the "chrooted" application.

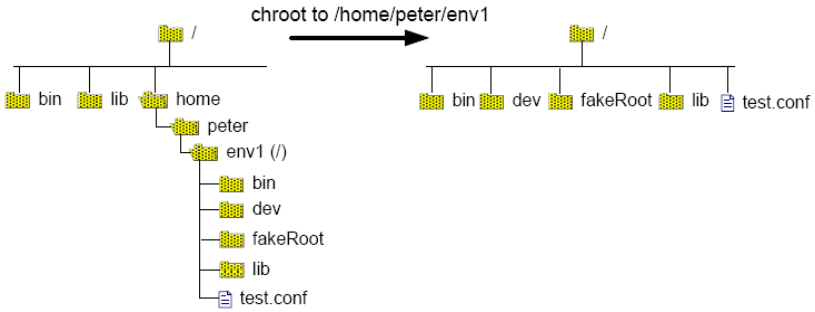

Figure 4.5: chrooted Application View in a Appropriate Virtual File Structure

`chroot` is not mentioned by the IEEE [43] as a standard system call in the POSIX [44] standard for system interfaces [45] but it is available as command or system call on many Unix-type operating systems even on modern lightweight high performance operating system like Catamount. Harness will be one of the important tools in high performance computing. It will influence other lightweight operating systems to support `chroot`.

Furthermore, the `chroot` mechanism can be used by C libraries. Therefore, the `chroot` mechanism can be used on both shell and shell-less operating systems if `chroot` is provided.

Security Issues

`chroot` can be used to trick security mechanisms of the operating system. Therefore, only a super user can perform `chroot`. If a normal user would be allowed to perform `chroot`, he could get super user rights. For instance, a normal user would be able to

reconfigure setuid programs vested with super user rights by chrooting such a program in a directory primed with manipulated configuration files.

This example shows why `chroot` can only be used per default by super users and why this configuration should not be changed.

Protected Usage of the chroot Mechanism

It is required to enable normal users to use `chroot` for the virtualisation approach. Otherwise no normal user could start specific applications in its user-space-based virtualised environments.

A protected way to enable normal users to use `chroot` can be achieved with `sudo`. `sudo` is a program that allows users to start programs with security privileges of another user. `sudo` is an abbreviation for substitute user and is available on most Unix-type operating systems.

Mostly, `sudo` is used by system administrators logged in as normal users, helping them to configure the system which requires super user privileges. As shown below, a normal user can run privileged programs with `sudo` if he knows the password of the super user. As shown in the following command lines, `sudo` is used as a prefix of the command or program which should be started. Per default `sudo` will prompt for the super user's password.

```
sudo vi /etc/shadow
Password:
```

`sudo` gives only privileges for the execution of the given program. All privileges are detracted after the execution, in this example after closing the vi editor. Per default `sudo` cannot be used to enable normal users in a protected way to run `chroot`. But the configuration of `sudo` offers the possibility to examine which program can be used by which users. The standard configuration file for `sudo` is located in /etc/sudoers. The required syntax of the sudoers configuration is given below:

```
peter  notebook = myCalc
```

This line gives `sudo` the information that user peter logged in from the host notebook is allowed to run the program myCalc. A more useful configuration line is illustrated in the following command line. This line allows all users logged in from anywhere to run the `chroot` command. `NOPASSWD` means that there is no password required.

```
ALL   ALL = NOPASSWD:/bin/chroot
```

This configuration line is still not a secure solution. It enables all users to use `chroot`. A configuration is required which enables all users to use `chroot` only for the specific task

to start their computational applications in their virtual environments. Such a limitation can be achieved by writing a shell script application which executes the following command line.

```
sudo chroot $1 $2
```

This application uses `chroot` to change the root directory of a given program $2 into a given directory $1. This application, named veStarter, should have super user rights to perform `chroot`. Therefore, the permission and ownership of the veStarter program needs to be modified, as illustrated by the following command lines.

```
chown root veStarter
chgrp root veStarter
chmod 744 veStarter
```

The sudo configuration in etc/suders includes the following line to enable all users to execute the veStarter shell scipt instead chroot, providing veStarter is located in `/bin`.

```
ALL   ALL = NOPASSWD:/bin/veStarter
```

So far, there is still the same security lack because veStarter works like the original `chroot` command. But the replacement of `chroot` by the veStarter application provides the implementation of a security mechanism into it. For instance, the veStarter application can check if the user is allowed to start a given program and it can analyze the given directory before it performs `chroot`. Since veStarter is owned by the super user, the application has no permission limits and can process all instructions, and it cannot be primed by other users.

This configuration provides a secure solution to enable normal users to use `chroot` to run their applications in a virtualised environment under secure conditions by replacing `chroot` with the veStarter application.

The result of this configuration is an encapsulation, implemented by limiting the application's view of the file structure via `chroot` mechanism.

4.3 Detailed System Workflow

Following this description of the basic virtualisation approach the entire application workflow will be outlined. Figure 4.6 illustrated the single steps in the right order.

First, the workflow starts with the transformation of the environment description XML file into a config file. This is can be done with an XSLT engine which transform the formats according to the defined transformation rules. After that, the user has to start

the veCreator application with its required parameter, the config file.

Then, the veCreator application creates the file structure of the new virtual environment in a given location according to the environment config file. Next, the user has to start the veStarter application. It uses the `chroot` mechanism to start a new shell. The root directory of this new shell is defined in the given the environment config file. After that, according to the environment config file, the veStarter application creates and manipulates the environment variables of the new shell. Finally in step eight, after the mentioned security checks, the given application is executed by the veStarter in the new shell.

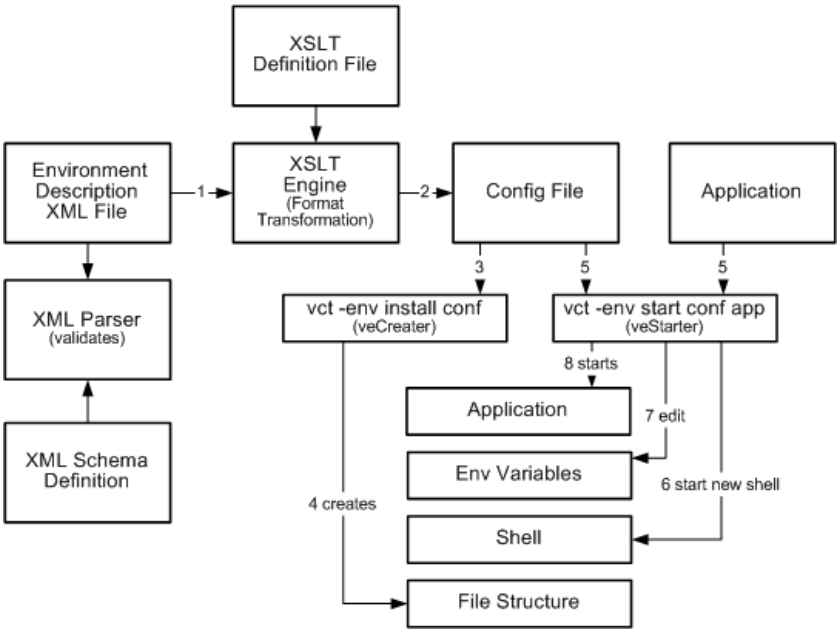

Figure 4.6: Detailed Workflow

The application is a child process of the new shell and runs as a "chrooted" process. It is required to start the real application into a "chrooted" shell to ensure that the changes of the environment variables will have no influence to the original shell.

Both applications veCreator and veStarter will be used via the virtual command toolkit as illustrated and already mentioned in Section 3.2.1. Environment variables are only temporary valid until the shell is closed. Therefore, the veStarter requires the environment config file to read and process the information about the environment variables. At a later time the first two steps will be integrated in the virtual command toolkit. It will analyze if an environment file is in XML format and needs to be transformed or not.

4.3.1 Evaluation

As already mentioned, currently this virtualisation approach concentrates only on file structures and environment variables.

Currently, some required environment properties which are not available on the operating system cannot be virtualised. For instance, running a specific service in the virtual environment could require an installation. The emulation of not provided devices requires the installation of services as well. Package management solutions are required for such complex installations. In addition, network interfaces and data streams cannot be emulated too.

Furthermore, other virtualisation tools provide resource quoting. The `chroot` mechanism in itself does not restrict the use of resources, like I/O bandwidth, disk space, or CPU time.

In the future, the `chroot` mechanism can be extended, for instance, to enable network isolation and disk quoting. On the other hand, such extensions may lead to portability problems of the virtualisation approach itself, and may decrease the performance because of running controlling processes during runtime.

In conclusion, the goal of this dissertation project is the portability of a virtualisation approach which itself ensures the portability of computational software.

The described approach fulfills this portability requirement. Especially the environment description concept can be seen as one stage of this approach, which is independent from operating systems and reusable by other approaches. The entire approach is not self-contained. Furthermore, the performance of this approach is not influenced by runtime environment software because there is none in use.

Classification

Finally, this virtualisation approach can be seen as an encapsulation realized by changing the root directory of a process and thereby cutting the system level scope of this process.

According to the term definitions in Section 3.3.3, the approach can be classified as an application virtualisation, because it does not emulate an guest operating system.

The virtualisation of a file structure and a shell environment can not be seen as an operating system environment. The approach concentrates on the runability, operability and executability of applications. Similar to the description of application virtualisation applications and libraries have not be installed. That avoids conflicts to already installed applications and libraries.

4.4 Related Research

Some related projects are described more in detail to get an overview of existing approaches developed to ensure portability of code. The rejection reasons, reusable aspects and the relation of these projects to this dissertation project are outlined. The following section involves the GNU Build System [46], the Oscar Project [47], the Xen Project [27], and the Netbuild Project [48].

4.4.1 GNU Build System

The GNU build system [49], also known as the Autotools, is a suite of open source tools produced by the GNU project. These tools are designed to assist in making various source code packages portable to many Unix-like systems.

The GNU Autotools can be seen as a collection of automated approaches to detect required system properties for compiling and installing of applications. For instance[50], it analyses specific compiler characteristics (`CC`,`CFlags`), the behavior of important utilities like `awk` (which can be `gawk`, `mawk`, `nawk`) or `grep` (`ggrep`) and it searches for required library functions, which can differ from system to system (BSD, POSIX or System V). GNU Autotools try to avoid well-known faults associated with these problems. For instance[51], some C functions like `exit`, `malloc`, and `free` having different return values and behavior, and, for instance, `strnlen` which is broken on AIX 4.3.

The GNU Autotool suite comprises the three main components autoconf [52], automake [53] and the libtool [54]. With the help of the GNU Autotool it is relatively easy for software developers to manage C and C+ projects and to create portable source code, which can be easily compiled by normal users on various Unix-like platforms. GNU Autotools are bourne shell compatible to be run able on Unix-type platforms.

Autoconf [52] produces a configuration shell script, named configure, which probes the installer platform for portability related information which is required to customize makefiles, configuration header files, and other application specific files. Then it proceeds to generate customized versions of these files from generic templates.

Automake [53] produces makefile templates, "Makefile.in" to be used by Autoconf, from a very high level specification stored in a file called "Makefile.am". Automake produces makefiles that conform to the GNU makefile standards, taking away the extraordinary effort required to produce them by hand. Automake requires Autoconf in order to be used properly.

Libtool [54] makes it possible to compile position indepedent code and build shared libraries in a portable manner. It does not require either Autoconf, or Automake and can

be used indepedently. Automake however supports libtool and interoperates with it in a seamless manner.

The developer has only to write two files called configure.in and makefile.ac [55]. The configure.in which includes macros for analyzing system properties required for the build process, and the makefile.ac file which includes information about the source code itself. The example below shows a typical configure.in file. It includes macros for checking the existence of the source code (`AC_INIT`), indicating that the source code is written in C and C++ (`AC_PROG_CC, AC_PROG_CXX`), and for generating the installation target (`AC_PROG_INSTALL`). As well it is determine that libtool and automake is used (`AC_PROG_LIBTOOL, AM_INIT_AUTOMAKE`) and how the make file, generated by automake, will be named (`AM_OUTPUT`).

```
AC_INIT(hello.cc)

AM_INIT_AUTOMAKE(hello,0.1)

AC_PROG_CC

AC_PROG_CXX

AC_PROG_INSTALL

AC_PROG_LIBTOOL

AC_OUTPUT(Makefile)
```

The second file the developer has to write is the makefile.ac. It contains information about the source project. For instance, binary directory, source code and object files can be defined as illustrated below.

```
bin_PROGRAMS = hello

hello_SOURCES = hello.h hello.cc main.cc
```

After writing these files the project is able to be build by entering the following commands:

```
1. aclocal
2. autoconf
3. automake
4. ./configure
5. make
6. make install
```

As shown in Figure 4.7 the aclocal analyses the configure.ac, creates specific analyzing macros, which will be used be autoconf and adds them in the project directory. Then

autoconf reads the configure.ac again and creates a configuration shell script, called configure. Next, automake reads the makefile.ac to create the makefile.in which is a template of the future makefile. After these steps the project is technically ready to be distributed as a portable source which can be easily build on various platforms.

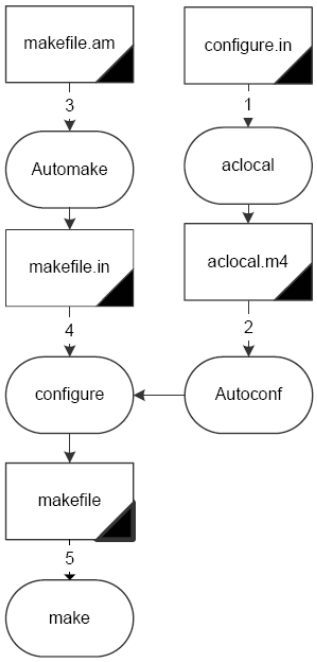

Figure 4.7: GNU Autotool Overview

Step 4 to 6 can be continued by a normal user, who has already downloaded the source and wants to install it. The configure script is doing the crucial step. It analyzes the system to see what kind of programs and libraries you have, so it knows how to build the program best and generates the final make file. At the end make can be executed to compile the project and with make install it can be installed. For these last 3 steps the GNU Autotools are not required to be installed on the system, only a bourne compatible shell and basic Unix utilities like awk and sed are needed.

After this short description it can be summarized that GNU Autotools simplifies the development of portable programs by automatic generation of the configure shell script and the make files, and distributing them together with the source code of the program.

However, it shows as well that the detection approach of GNU Autotools is not constructed to deal with high performance computer platforms where compilation and execution environment are different.

As mentioned in Section 3.3.1 modern high performance computing systems use cross-

compilation due to dedicated nodes. Abstractly, a program can be cross-compiled on a service node, and then it will be executed on a computing node. While running on a computing node the program can use the services offered by the service nodes.

GNU Autotools support cross-compiling but not on high performance computing systems. If GNU Autotool would be used on a supercomputer with such an order it would fail. The detection approach of the GNU Autotools analyses only properties of the compilation (service) node but not the execution environment on the compute node. Cross-compiling, is possible with the GNU Autotools but the entire analyzing approach, which is responsible for the portability cannot be used for the target compute node.

In theory, it seems to be a good idea to rewrite some parts of the GNU Autotools to enable them to be executable on the execution environment of a compute node and to consider these results for the compilation on a compiling (service) node.

A shell is required to execute them on a compute node. Modern high performance computing systems run lightweight operating systems working without any kind of shell on compute nodes. Thus, it is required to re-implement the entire GNU Autotool suite in C to enable it for execution on such compute nodes. The associated amount of work needs inadmissible time and will excess the limits of this project.

Especially in relation to this, it should be underlined again that the philosophy for the crucial functionality of the Harness Workbench Project is based on including users and system administrators. An automated process would require permanent maintenance because of the growing diversity of systems leading to a vast number of exceptions, as exemplified by the GNU Autotools. Otherwise the detection approach would fail if a new unknown problem occurs. The maintenance of C-implemented GNU Autotools in such a fast changing area is very inefficient.

For these reasons, the GNU Autotools cannot be considered as an alternative solution to the dissertation's virtualisation approach in face of modern high performance computing systems. In practice, as follows from the experience of my institutional supervisor, only a small percentage of scientific applications use GNU Autotools instead of proprietary build environments as current high performance applications have been developed over the last five to fifteen years.

4.4.2 Oscar

OSCAR [47] is a collection of the best known methods for building, programming, and using clusters based upon PVM [56]/MPI. OSCAR stands for Open Source Cluster Application Resources and is the primary project of the Open Cluster Group, from which the Oak Ridge National Laboratory is a part of. It helps to simplify the complex tasks of installing and configuring clusters. OSCAR consists of a full integrated and easy to

install software bundle designed for high performance computing clusters. Everything needed to install, build, maintain, and use a Linux cluster is included in the suite.

Oscar evolves a GUI-based packages manager, a resource manager for job submission and controlling, the configurable job scheduler, the Cluster Command Control (C3) suit for cluster-wide command executions and the MPI programming environment LAM. Furthermore it evolves a system installation suite (SIS) which is used to install client nodes. SIS provides a database from which OSCAR obtains its cluster configuration information. The main concept to understand the SIS as an image based install tool. An image is basically a copy of all files, that get installed on a client. This image is stored on the server, and can be accessed for customizations or updates. It is even possible to `chroot` into the image and to perform builds. Therefore SIS is interesting for this dissertation project, because it uses as well the `chroot` mechanism.

Another interesting tool of OSCAR is the switcher package, provides a convenient command-line interface to manipulate the inclusion of packages in a user's environment. Users are neither required to manually edit their .dot files, nor they are required to know, what the inclusion of a given package in the environment entails. For example, if a user specifies that they want LAM/MPI [57] in their environment, switcher will automatically add the appropriate entries to the specific environment variables like `$PATH` and `$MANPATH`.

Like this dissertation project OSCAR switcher manipulates shell environment variables to adapt an appropriate environment.

4.4.3 XEN

Xen [27] is virtual machine monitor, also called hypervisor, which allows running several guest operating systems on a host operating system at the same time on the same machine. Xen is available for different hardware architectures like for PowerPC, x86-32 and x86-64, and IA64. Xen started as a research project of the University of Cambridge and is now available as open source software and as commercial enterprise version [58] [27].

As guest operating system, it supports several Unix-like systems, unmodified versions of Windows, and, with limitations, some other operating systems. Operating systems like FreeBSD [60], NetBSD [61] and Plan 9 [62] may be also employed as guest systems.

In context of this dissertation, Xen provides a starting point as a system-level virtualisation solution for virtualised system environments (VSE)in high performance computing environments [63].

The developed description concept of this dissertation project may be reused and extended to describe VSE. System-level virtualisation tools, like Xen, may use the description concept to modify operating system image files before starting them as guest OS. Hence, the description concept should be developed with regards to such tools.

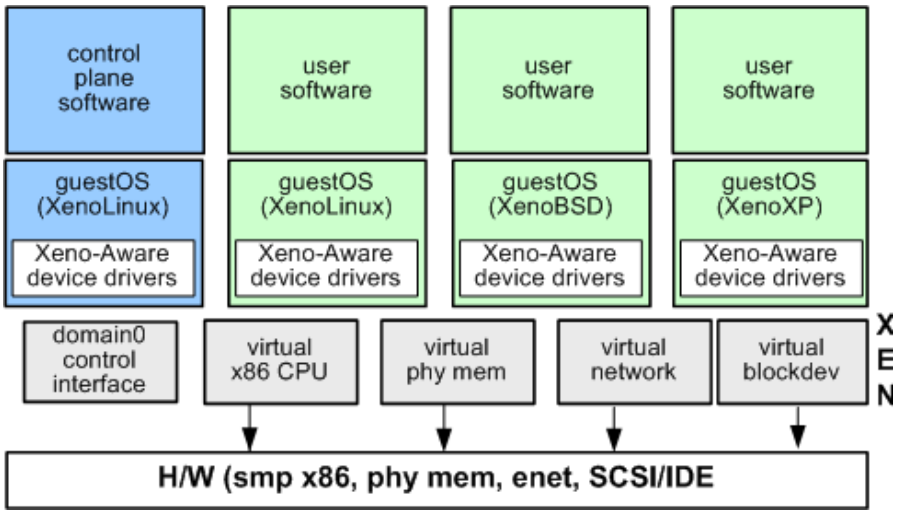

Figure 4.8: Xen Architecture

4.4.4 Netbuild

NetBuild [48] is a suite of tools designed to aid users in making use of computational software libraries that are stored on the network, without needing to have those libraries preinstalled on each user's computer. Instead, NetBuild will determine which libraries are not installed, identify suitable versions of those libraries that are accessible from the network, download those libraries, and link them into the user's program. NetBuild is developed by the ICL Team at the University of Tennessee and actually runs on most UNIX platforms (including MacOS X) and also on Windows (using Cygwin).

In practical, netbuild can be used by its client program nb2 as follows: The nb2 program is used to run the user's compiler, or make, or a compilation script, or any other program that would be used to compile the user's program.
For instance, instead of typing: `f77 program.f -llapack -lblas`

the user would type: `nb2 f77 program.f -llapack -lblas`,

or instead of typing: `make program`

the user would type: `nb2 make program`.

`nb2` then runs the supplied command in an altered environment, which has its `PATH` variable modified to have a directory prepend to it. That directory contains shims, which have the same name as compilers and linkers that `nb2` needs to intercept.

Whenever one of those compilers or linkers is invoked - either directly from `nb2`, by make, or by some other compilation tool, the shim is run instead of the real compiler. The shim then parses the compiler's arguments looking for names of libraries that need to be linked in. If those libraries are not installed on the system, the shim then downloads them, verifies their signatures, and extracts them into an empty directory. Finally the real compiler or linker is then run with a modified argument list that causes the newly-downloaded libraries to be linked in along with the user's program and any native libraries that are used.

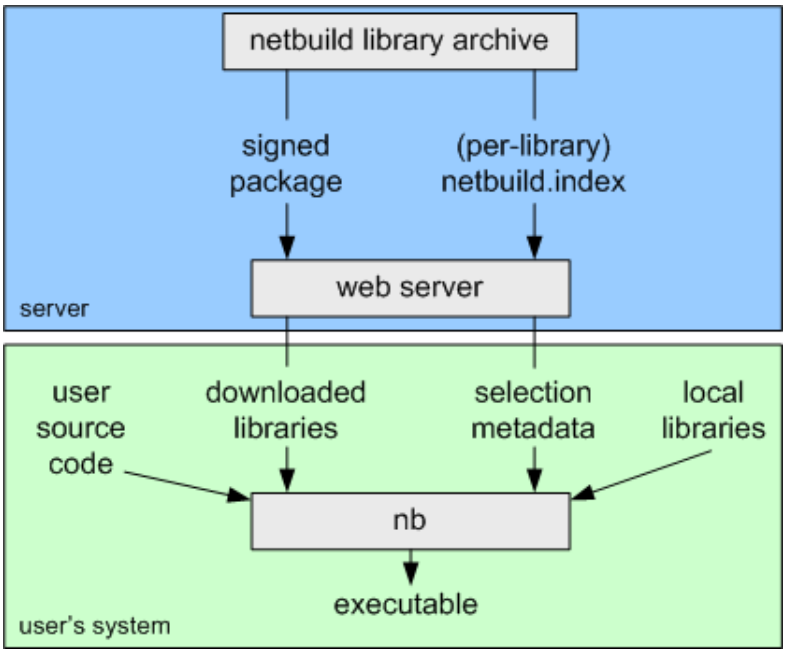

Figure 4.9: Netbuild Architecture

This short description shows that netbuild can be used to avoid library installations on the compiling platform, but using them for build processes at the same time. This library usage feature fits to one of mentioned project requirements.

In comparison to the chosen approach for using libraries accomplished by this dissertation project, it is more complex and involves external library servers deciding which library suits best for what platform. The virtualisation approach of this dissertation achieves this feature with user-space-based virtualised environments. This is an easier way, which supports not only libraries but arbitrary applications and files.

Netbuild is still in development and a data model needs to be defined for each new processor type for the identification of the best library version for a platform. Such a

library version decision approach requires that the library server holds one version of a library for each combination of operating system and processor. If this approach works perfect there is still the question if it is really needed for supercomputers, where site administrators are available, knowing specifics to take decisions about library versions. Netbuild seems to be convenient to support the development of general purposes but it is not strongly needed for specific purposes in the high performance computing area.

5 Implementation Strategy

5.1 File System and Structure Virtualisation

Description Specification

Developing a description concept for virtual file structures requires an analysis of what is needed in practice and what combinations could be possible. Based on those results, the concept can be enhanced to fulfill all-embracing demands. This section outlines, how file structures can be described, and discusses practical approaches of file integration.

Abstract Elements for a File Structure Description

At first, the concept should be able to describe a static view of a required file structure. Therefore, the concept includes definitions of terms or elements, which can represent file structure elements.

Basic elements of a file structure are files and directories. A directory is a container, which includes a number of files and subdirectories. A directory has attributes like permissions, ownerships, a name and a location. The concept should offer a data structure, which includes all these attributes.

A file is not a container. It can be seen as a leaf in the file structure tree. Technically, in Unix-type file systems a directory is not more than a file. This detail can be ignored, because the logical view is important. For this reason, a file element has the same attributes like a directory. A file has as permissions, ownerships, a name, and a location. With these two elements, the tree structure of a file system can be described. The XML dialect of the concept implements directory elements, which can include any number of file and subdirectory elements.

From this abstract view, subdirectory elements are the same like directory elements, having the same attributes and can also include any file and subdirectory elements.

The described elements above can be used to form a static tree structure. This tree structure describes the file structure of a virtualised environment. This section outlines, how to describe where the required files are originally located and how they can be integrated in the virtual file structure. According to the specifications in Section 3.2.1, it should be possible to merge both, new files and local files, already used by the operating system. This saves disk space and allows arbitrary combinations.

Therefore, the application, which creates the virtual file structure requires information from the description concept where the files are located and how they should be integrated.

File Integration

File integration is the process, which includes a file in a virtual environment, in a way that the file data is accessable for applications running in a virtual environment. As described in the following, this does not necessarily mean that integrated files will be stored in the file structure of the virtual environment.

Depending on the use of each file in the virtualised environment, different methods for file integration are convenient. This dissertation project is focused on three integration methods. The copy, link and Unionfs method.

5.1.1 Copy Method

The copy method represents the easiest way to integrate a file in the virtual file structure. A file is copied from its original location to the described location in the virtual file structure. The copy method is useful, e.g., if an original file should be protected by operations inside the virtual environment. In theory, most use cases can be satisfied with this method but it needs too much resources. Running and compiling of a complex application in a virtual environment can require copying of big parts of the operating system file structure. For instance, the library, configuration and binary directory (`/lib`, `/etc/` and `/bin`) include thousands of files. In practice, a user needs more than one of such big virtualised environments. The required disk space can exceed the system or a user's disk space quota. In addition, the creation of such virtual file structure via copy method can take unacceptable time.

5.1.2 Link Method

The link method is the ideal extension to the copy method to solve these problems. The link method creates only a symbolic link in the virtual file structure instead of a file. Links can be used to reuse existing files in the virtual file structure. It saves disk space and the creation of thousands of links instead of copying files take less time. Especially, a symbolic link to an entire directory is very efficient because only one symbolic link needs to be created to integrate all its included files and subdirectories.

This integration method has the disadvantage that the properties of an integrated file cannot be changed for the usage in the virtual environment. It is not possible to change write or read permissions for the link of a file. Furthermore, the original file is not protected by operations from inside the virtual environment. A file is not influenced if it is integrated via link and an application running inside the virtual environment deletes this. This operation deletes only the link of the file in the virtual file structure. This is in contrast to a situation, when an application deletes or changes the data of such a

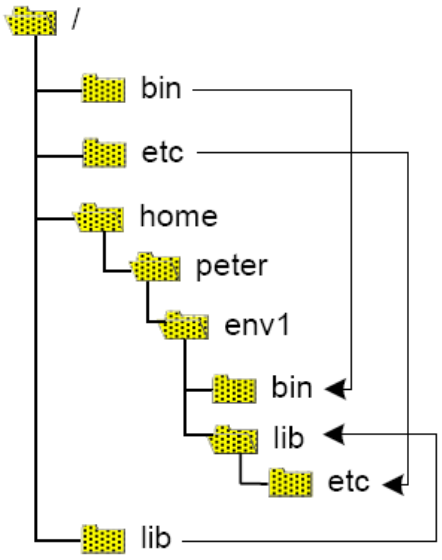

Figure 5.1: File Integration via Copy Method

linked file. Then the original file is irrecoverable changed.

The link method can be used instead of the copy method to integrate files which are required in the virtual environment without changes. Typically the user and therewith the application of the virtual environment has no write rights to such files.

Due to the fact, that the basic virtualisation approach is based on the use of the `chroot` mechanism, files located outside the new root directory are not visible and not accessable for applications running inside the virtual environment. For instance, the integration of the file test.conf in `/home/peter/` to the virtual file structure location `/etc` fails. The link is located in `/home/peter/env1/etc/` and is named test.conf like the original file and links to `/home/peter/test.conf`. For an application which was "chrooted" into `/home/peter/env1` this link is dead. `/home/peter` is outside the system level-scope of this application, see Figure 5.2.

A solution is needed to provide symbolic links, which link to a file outside the scope of a "chrooted" application. Therefore, the entire file structure of the operating system can be mounted into a given directory, which is part of the virtualised file structure.

This directory is named `.osroot` and includes the entire file structure and can be mounted as follows.

```
sudo mount --bind / home/peter/env1/.osroot
```

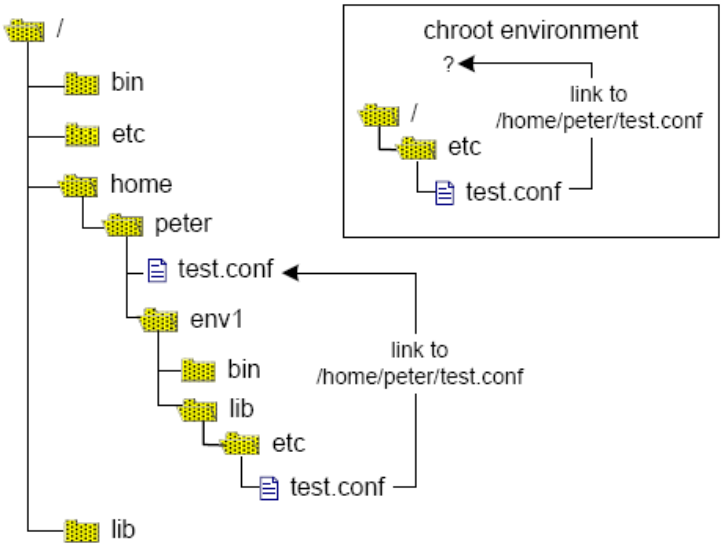

Figure 5.2: Failed Attempt of the Link Method

Even an application which is "chrooted" into `/home/peter/env1` can see the file structure which is included in `.osroot`. So, in order to integrate the file test.conf, a symbolic link has to be created, which is located in `/home/peter/env1/etc` and is named test.conf. This symbolic link links to `/.osroot/etc/test.conf`. This link does not link to `/home/peter/env1/.osroot/test.conf`, because this path is not visible for the "chrooted" application, see Figure 5.3.

In conclusion, such a symbolic link links only to the correct file if it is used by programs "chrooted" in the specific virtual file structure. For other applications this symbolic link seems to link to a not existing file.

5.1.3 Unionfs

The third approach deals with Unionfs [7]. It is a very promising stackable unification file system. Unionfs allows the specification of series of directories, which are presented to users as one virtual directory. In this context, such a virtual directory is called a union, which comprises of the specified directories, named branches. Unionfs creates a union by unifying the content of different branches.

The content of a union are the merged files and subdirectories of its branches. For a user a union looks like a normal directory. Even branches from different file systems can be merged to one union. This is commonly referred to as namespace unification.

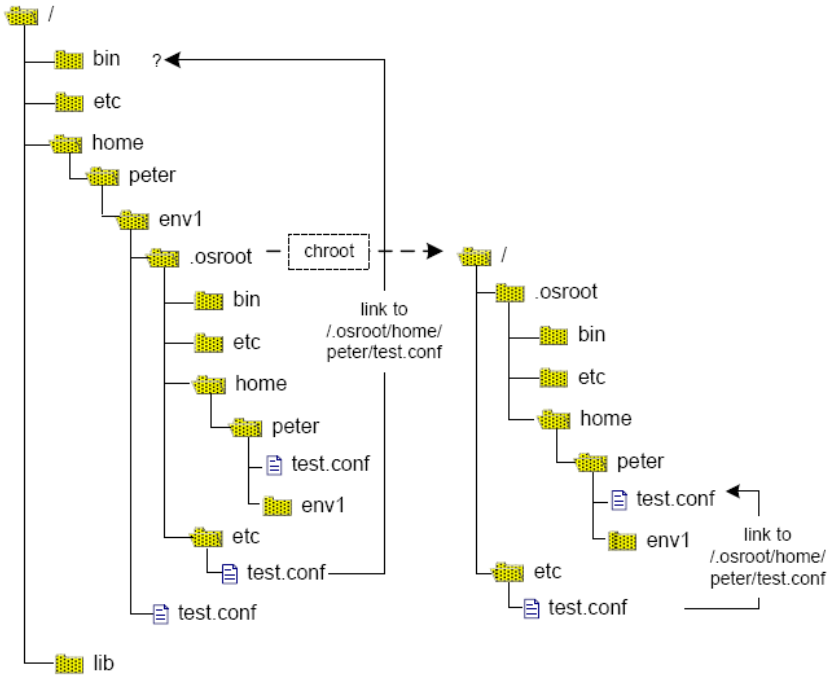

Figure 5.3: File Integration via Link Method and Mounting

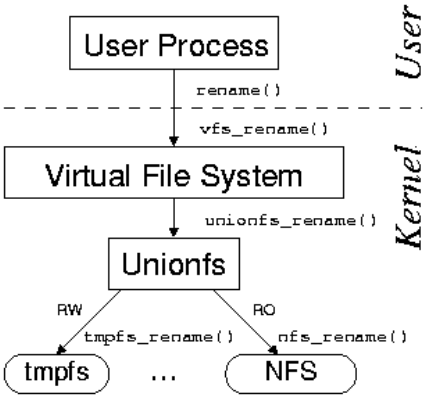

Figure 5.4: Unionfs Layers Over Multiple Directories [64]

Unionfs is based on the stackable file system templates (FiST) [65], which provides support for layering over a single directory. A normal operating system includes the kernel's virtual file system (VFS) [66] and a file system. As shown in Figure 5.4, the kernel's virtual file system is responsible for dispatching file-system-related system calls to the appropriate file system.

Unionfs, as a stackable file system, works between VFS and the file system. To the VFS, Unionfs looks like a standard file system, but instead of storing or retrieving data, a stackable file system passes calls down to lower-level file systems, which are responsible for data storage and retrieval. In this Figure 5.4, NFS is used as the lower-level file system.

To the lower-level file systems, Unionfs looks like the kernel's VFS. Unionfs can be used to layer over any file system (e.g. Ext2, Ext3, Reiserfs, SQUASHFS, isofs, and tmpfs) because it adheres to the conventions both of file systems for processing VFS calls and of the VFS for making VFS calls.

Features

A union can be created as easy as mounting devices or directories. As shown in the following command line, a union can be created and mounted via the `mount` command and appropriate options.

```
sudo mount -t unionfs -o dirs=/home/peter/mylib:/lib unionfs /home/peter/env1/lib
```

The branches are separated with ":" in the command line. According to Figure 5.5, the libraries and files of these two branches `/home/peter/mylib` and `/lib` are merged to a union, which is located in `/home/peter/env1/lib`. The example demonstrates how a system library directory and a user's directory can be merged to an new library directory which can be integrated in a virtulized environment.

To avoid file or subdirectory name conflicts, Unionfs uses a simple priority system, which gives each branch a unique priority. Each branch has a priority according to the order in the command line. If a file exists in multiple branches, the user recognizes only the file of the higher-priority branch. Hence, the branch `/home/peter/mylib` in Figure 5.5 has a higher priority than `/lib`. In this example the individual libraries override the standard libraries.

The case described above creates a new union in a new location. The original content of the two branches is still accessable and visible by its paths.
Furthermore, Unionfs provides the creation of a union with the name of an existing directory. The following example shows the creation of a union, which is located in `/lib`.

```
sudo mount -t unionfs -o dirs=/home/peter/mylib:/lib unionfs /lib
```

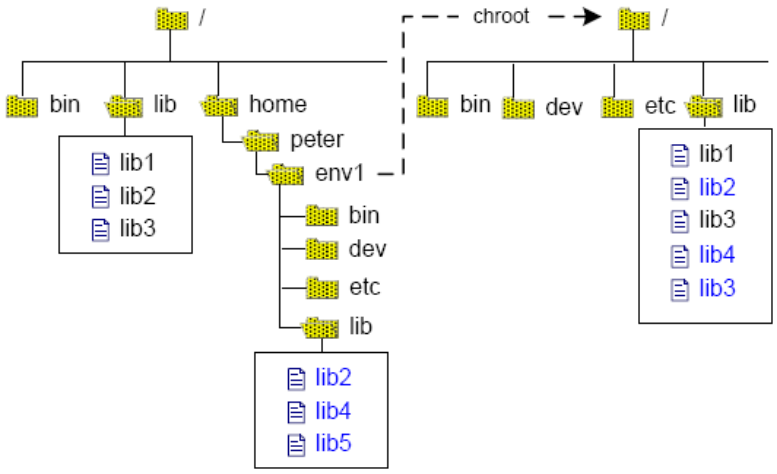

Figure 5.5: Merging Directories with Unionfs

The original content of `/lib` is not accessable until the union is unmounted again, see Figure 5.6.

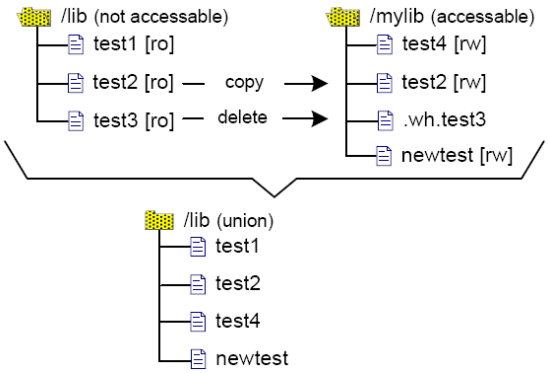

Figure 5.6: Overlying Union with Copy-On-Write Function

A union can be easily unmounted with the `umount` command as shown in the following:

```
sudo umount  /lib
```

Furthermore, Unionfs provides the determination of different permissions for each branch. A branch can have the permission readonly (ro) or read-write (rw). As long there are no permissions specified each branch is read and writable per default. The permission for each branch can be determine in the following way:

```
sudo mount -t unionfs -o dirs=/home/matti/mylib=rw:/lib=ro unionfs /lib
```

A branch, which is writable for a user via its original path, can be integrated in a union as readonly. Thus, Unionfs provides the limitation of write rights without changing the original file permissions.

At last as shown below, with the `unionctl` command an existing union can be manipulated. Branches can be added and removed, their order can be changed and the read and write permissions can be manipulated.

```
unionctl  /home/peter/env1/lib -add  /usr/lib
```

The most useful feature of Unionfs for this project is its copy-on-write function. Unionfs intercepts all accesses to merged files in a union. Unionfs controls if a process tries to write in a file, which should be protected from manipulations. It copies the specific file in another higher-priority branch of the union if a process tries to write in a file.

This copied file is then writable and replaces the write-protected original file in the union, because it is located in a higher-priority branch. This background process is not recognized by the writing processes. Thus, Unionfs requires still a higher-priority branch in a union to provide the copy-on-write function. It uses the copy-on-write semantics to simulate that all branches are writable, independent from the original file rights. In conjunction with this deleting a protected file triggers Unionfs to create a whiteout file in a higher-priority branch instead of deleting the original one.

The name of this whiteout file consists of a prefix `.wh.` and the name of the file, which should be deleted. Whiteout files are used by Unionfs like a flag for each specific file. A file is not contained in a union if such a whiteout file exists for the specific file, see Figure 5.6.

Unionfs can provide the manipulation of writeable files without changing the original one by integrating them in a union as readonly. In addition, it makes readonly branches, like cdrom-devices, virtually writable. This feature allows Live-CD developers to give their users a writable system based on read-only media. In the example above, libraries in the `/lib` branch are not writable for a normal user but on demand the copy-on-write function copies the specific library into `/home/peter/mylib`.

Limitations of Unionfs

- **mmap**
  The functionality of Unionfs is based on controlling the activities between the VFS and the underlying file system. Unionfs can intercept all activities except activities of programs using the `mmap` [67] system calls. The `mmap` system call cannot be intercepted, because it allows programs to map portions of files into a process's address space. Is a file opend with mmapp, a process can modify it by writing

to the correct location in memory. This leads to the drawback that Unionfs does not receive notification of readpage or writepage calls, so it cannot perform the copy-on-write function, when a program tries to write in a protected file [64].

- **Recursive Unification**
  Another limitation of Unionfs is that it does not provide merging of unions to a new union. It is not possible to specify a directory, which is already a union as a branch of another union. This leads to inflexibility and limits the use of Unionfs for the creation of virtual file structures.

- **Modification of Lower-Level Branches**
  The current design of Unionfs and other stackable file systems on Linux results in double caching of data and meta-data. This is an unfortunate side-effect of the way the Linux VFS is implemented. Therefore, Unionfs maintains a list of lower VFS objects for each upper object. For instance, a Unionfs inode contains an array of pointers to all the corresponding inodes on the underlying branches. Unionfs has to copy certain information, like file size and access permissions, from the underlying inode for a file to the Unionfs inode. Since Unionfs expects the underlying inode to have certain properties about the files it is possible for inconsistencies to appear if a process modifies the lower inode directly without going through Unionfs [64].

- **Performance**
  The low performance of Unionfs is the biggest problem for the requirements of this dissertation project. Performance is not a mentioned requirement in Section 3.2.1 but it is obviously an important issue for computational applications running on high performance computing systems.

- **File System Behaviour**
  The priority system of Unionfs leads to a noticeable different behavior. If a multiple file is deleted in the union its second copy from a branch with a lower priority is displayed in the union. This can be irritating for users if a file was deleted and there is still a file with the same name in the union. This behavior is not so seriously considered but it underlines that Unionfs cannot completely emulate a real file system.

- **Scalability**
  Currently, Unionfs is able to manage and merge 1,024 branches. Independent from the question if this number of branches will be exceeded, when an entire virtual file structure needs to be emulated, the performance of Unionfs is very slow. The overhead of using Unionfs becomes high with just 200 branches, even for simple operations. The problem is, that Unionfs needs to iterate through all the branches. For each branch it needs to determine, whether or not it is a duplicate, whiteout, and so on. Currently, such stacking information is stored in a simple linear array, which has a search complexity of O(n).

In addition, some of the Unionfs functions are not stable on the tested system. The limitation of access rights leads to an empty file if a user tries to change its permissions. The mount command as described in the manual is not able to mount branches as readonly. This only can be achieved by modifying an union with the `unionctl` command.

Recapitulating, Unionfs is a new project and still in progress. The mentioned disadvantages will be improved or solved in the future. Since Unionfs is used for diskless systems and Live-CDs like Knoppix [68], it becomes more popular and thereby will experience stronger efforts. It can be estimated that a convenient solution for some of the problems will be implemented in the future. Therefore, Unionfs is considered for the file integration as a useful alternative or extension to the aforementioned copy and link methods.

Especially the scalability and the performance can be improved. The mentioned search complexity of linear array structures can be replaced by logarithmic search algorithms. Currently, a partial implementation, which supports `mmap` system calls is released. Especially the `mmap` problem is lapsed for the use on many of the modern supercomputers, because their compute node kernels do not supported this system call [69].

File Integration with Unionfs

This section outlines, how Unionfs can be used as an approach to integrate files in a virtual file structure. It describes, what properties are required and how Unionfs can extend the above mentioned methods.

- **File Usage Combinations**
  In the following, thinkable combinations for file usage in conjunction with virtual environments are described. The combinations, which can be realized with the copy and link method are described first. This affords an easier analyze of Unionfs as a file integration method.

  Figure 5.7 illustrates the file system scopes of two programs. The left part shows the scope a program X, which runs on an operating system environment. Program Y, on the right, runs in a virtual environment. Both programs have access to `file.z`, which is originally located in the file structure of the operating system, and is also integrated in the file structure of a virtual environment.

  Different file integration methods can be used for `file.z` depending on the requirements of the use case. In the following, two use cases are described. The first requires information interchange, the second not.

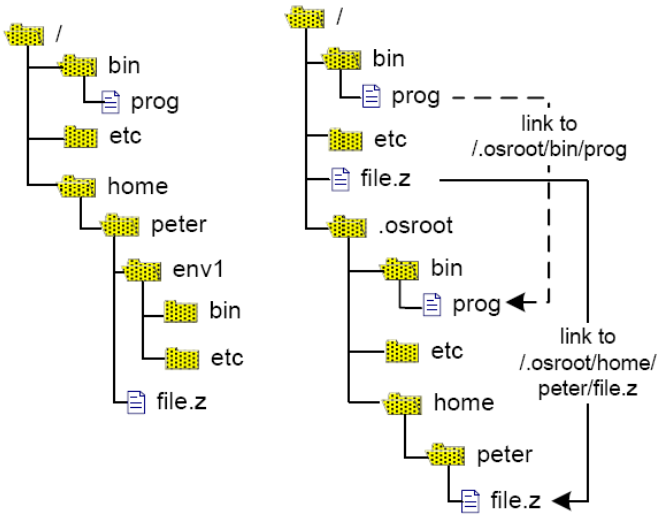

Figure 5.7: Two File System Scopes, Original(left) and Chrooted(right)

In this context, information interchange means, that program X outside the virtual environment writes data in `file.z`, and program Y inside the virtual environment is able to receive this new data. In this case a two way interchange is possible and program Y can also write in `file.z` and program X reads this data.

No information interchange means, that both programs cannot notice the other's manipulations in `file.z`, but are able to manipulate the file for their own and other programs running in the same environment.

The first situation can only realized with the link method. The veCreator application creates a link in the file's location in the virtual file structure. Both programs have the same access to the same file.

The second situation can be solved with the copy method. The veCreator application copies the file during the creation of the virtual file structure so that both programs work with different files.

Figure 5.7 illustrates that both programs are instances of the same binary file `/bin/prog`. It is required that a program's binary file is located in the virtual file structure in which it should be started with the `chroot` command. `file.z` is integrated via symbolic link to allow information interchange.

However, these situations can be split in more detailed combinations.

Table 5.1 illustrates combinations, which considers different file permissions be-

	OS	Transfer	ENV	Solution
1	rw	static	ro	Copy
2	rw	static	rw	Copy; UnionfsCow
3	ro	static	rw	Copy; UnionfsCow
4	ro	static	ro	Copy
5	rw	dynamic	rw	Link
6	ro	dynamic	ro	impossible
7	ro	dynamic	rw	impossible
8	rw	dynamic	ro	UnionfsRo

Table 5.1: Combinations

tween the environments. This table is focused to a user's point of view for one specific file.

The first column shows a user's read and write permissions for a file on the operating system's file system. This user is the one who starts applications in its own virtual environment. The third column shows the (virtual) read and write permissions of the same user for the same integrated file in the user's virtual environments.

The second column contains the intended information interchange via the specific file between programs in the operating system environment and programs running in virtual environment or between two programs running in two different virtual environments. The arrows illustrate the possible directions of the interchange.

The first four combinations match to situation 2 and can be solved with the copy method. For instance, the first combination represents a situation in which the user has read and write rights (rw in first column) for the file outside the virtual environment. A program X, started by this user outside the virtual environment, has full access to this file. Inside the virtual environment program Y, which is started as well from this user, should have no write rights for this file, only read rights. Furthermore, information interchange between this both programs should not be possible. Short, for program Y this file has not changed and looks as it was, when the virtual environment was created.

As mentioned, this situation can be solved with the copy method. The veCreator application copies the file during the creation of the virtual file structure, and then it revokes the write permissions of the user for this file. The other three combinations can be realized in the same way, only the required file permissions are different.

Similar to situation 2, there are four combinations for situation 1 but only the combination in row 5 can be realized with the link method. Combination 6 cannot

be realized because the write permission is not granted in both environments. The file can be integrated via link method, but with no information interchange. Combination 7 is as well not realizable. It is not possible to distinguish different file permissions depending on the access via link or directly. The copy method would enable information interchange. The only way to grant more access rights to a program in a virtual environment than the normal user has, is starting the program in the virtual environment in the name of a more privileged user, for instance with `sudo`. But such possibilities are not considered as a solution, because they involve security relevant issues which are discussed in Section 6.6.

The last combination is similar to the combination 7 only the file permissions are opposite. As well the link method cannot be used because of the different file permissions.

Combination 4 can also be realized with the link method, because the write permission is not granted in both environments. The file cannot be manipulated and information interchange is not possible. In this case a link is sufficient, but this table shows only the read and write permissions of user programs. Super user programs and system daemons can be able to manipulate the data of the file. The file could be replaced because of system installations and configurations. A link cannot protect a file from such activities. Depending on what is required, this can be an advantage or disadvantage.

The above outlined table shows the integration possibilities of the copy and link method. Three of eight combinations cannot be realized with these methods. In the following it is analyzed, how Unionfs can be used to extend these two methods.

Unionfs provides two interesting functions, which can be helpful: the copy-on-write function and the limitation of write permissions. Required directories can be merged in a union, which itself is located in the virtual file structure. Depending on the situation each directory can be integrated as readonly or as read and writeable.

The ability of Unionfs to limit write permissions without manipulations on the original file system suits perfect to combination 8. Unionfs can be used to limit the write permission of a file, in a way, that only the program inside the virtual environment is able to read it. Thus, this is a one way communication into the virtual environment.

Combination 7 results as well to a one way communication but outside the virtual environment. This combination could be realized with Unionfs, but the union which limits the write permission has to be outside the virtual environment. This cannot be implemented, because it could influence the entire operating system.

As mentioned, Unionfs is slower than a normal file system. Therefore such an implementation could slowdown the operating system or important services. Third party programs, which need to write into this file cannot work properly because Unionfs blocks their write activities. Unionfs cannot be configurated to distinguish different permission for different users.

- **Implementation of Unionfs**

With Unionfs, it is not possible to select single files to integrate them with the specific permissions. It is only possible to select and integrate entire directories, which includes as well their subdirectories. In Figure 5.8 there are two directories `/dir1` and `/dir2`, each containing three files, but only one file from each, should be merged in a new directory `/dir3` which then is integrated in a virtual file structure `/home/peter/env1`. Referring to combination 8, these files are readable and writable for the user on the file system, and are needed as readonly files in the virtual environment. Information interchange is needed too. Therefore, this case cannot be solved with the copy or link method.

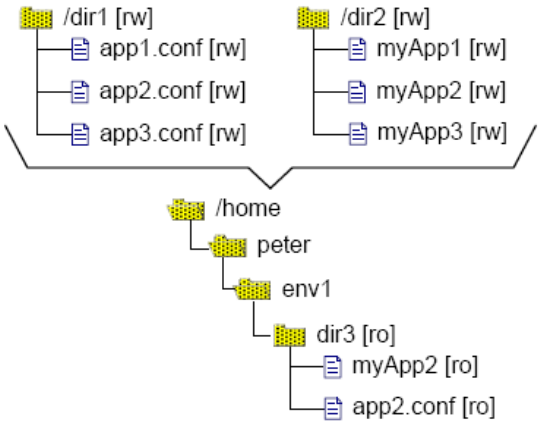

Figure 5.8: Two Directories and their Required Merge

An attempt to use Unionfs functionality for single files is to create an intermediate directory `/temp`, which contains two symbolic links to the both required files `/dir1/app2.conf` and `/dir2/myApp2`. Then this intermediate directory is merged as readonly in a union, see Figure 5.9.

The union consists only of the branch `/temp` to avoid the copy-on-write function. With the command line shown below the union `/dir3` is created and mounted in the virtual file structure.

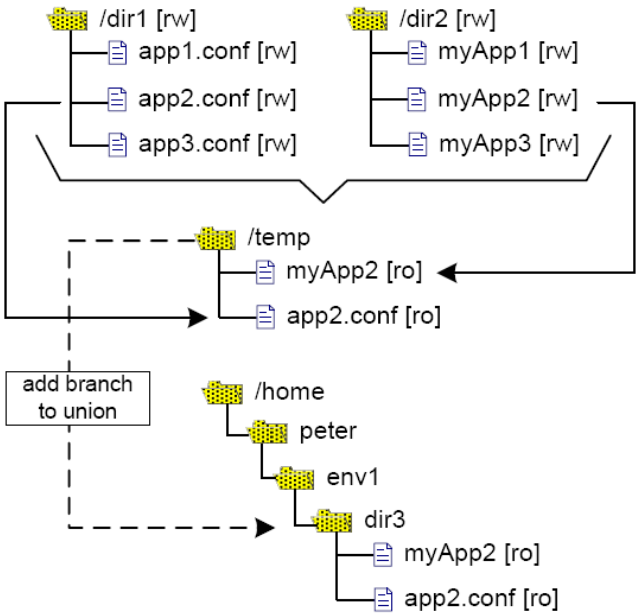

Figure 5.9: Failed Attempt

```
sudo mount -t unionfs -o dirs=/temp=ro unionfs /home/peter/env1/dir3
```

The goal of this construction is obviously but it fails. Unionfs controls only files which are contained in the union. The union contains only the two symbolic links but not the original files they linking to. Hence, running programs in the virtual environment can access the original files via the symbolic links in the union. These programs cannot delete the two symbolic links, but they can manipulate the data of the both files.

A solution for this problem is illustrated in Figure 5.10. The order of the above try is turned upside down. Each directory is merged as readonly in a union. These unions are integrated in the virtual file structure but in an intermediate location `/home/peter/env1/.unions`. The required directory `/home/peter/env1/dir3`is created like a normal directory in the virtual file structure. At next, two symbolic links are created in this new directory. These links link to the required files in the two intermediate unions. This construction fulfills the requirements. The virtual file structure contains the required directory which includes only two symbolic links instead of the original files and Unionfs is able to control the file permissions because the files are part of the unions.

```
sudo mount -t unionfs -o dirs=/dir1=ro unionfs /home/peter/env1/.unions/dir1
sudo mount -t unionfs -o dirs=/dir2=ro unionfs /home/peter/env1/.unions/dir2
```

```
mkdir /home/peter/env1/dir3
ln -s ./unions/dir1/app2.conf /home/peter/env1/dir3/app2.conf
ln -s ./unions/dir2/myApp /home/peter/env1/dir3/myApp
```

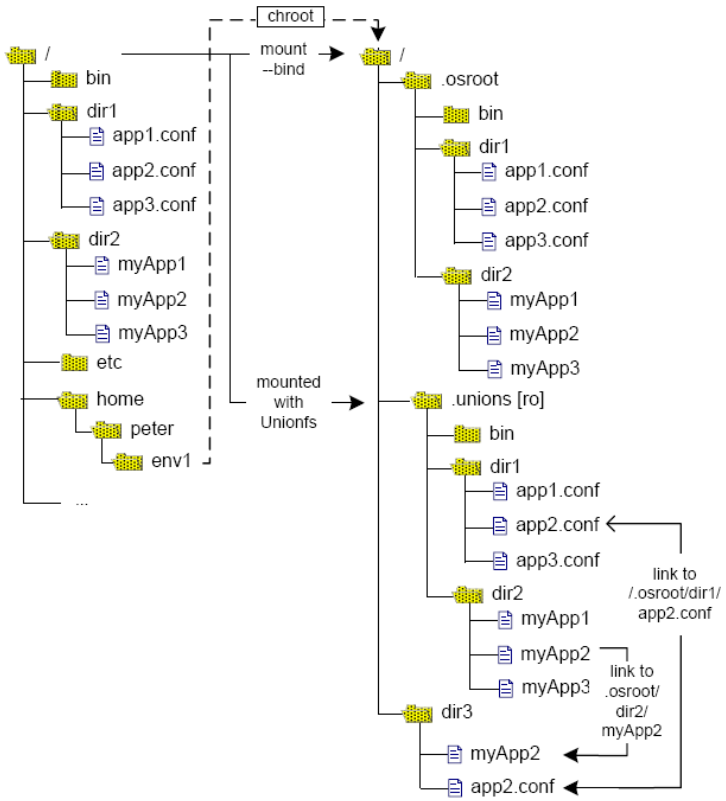

Figure 5.10: Unionfs Implementation Solution

Thus, Unionfs can be used selectively for single files. The described approach suits to the requirements but it leads to the creation of many unions, for each directory, which contains a file, which should be integrated. The following solution avoids big amounts of unions for each virtual environment.

As mentioned each virtual file structure mounts the entire file structure of the operating system in an intermediate directory called `.osroot`. Like the `.osroot` directory, a union named `.unions` is integrated in the virtual file structure. This union contains the entire file system again as readonly. Therefore, every file of the file system can be integrated as readonly file via Unionfs.

This solution demonstrates that Unionfs can be used as a helpful addition to the copy and link methods. Unionfs is the only integration method which can im-

plement combination 8 but it can also be used as a good alternative to the copy method for combination 2 and 3. In these combinations the copy-on-write function can be used if a file will be manipulated.

In addition to this readonly `.union`, a second union `.unionCow` is required which provides the copy-on-write function for the entire file structure. A union with copy-on-write function consists of two branches. The first branch is an empty intermediate directory, `.branchWrite`. It has the highest priority in this union and is read and write able. The second branch is the root directory, named `.osrootRo` . When a file of this union is manipulated for the first time, it is copied first from the `.unionRo` branch in the `.branchWrite` and replaces the original file in the union.

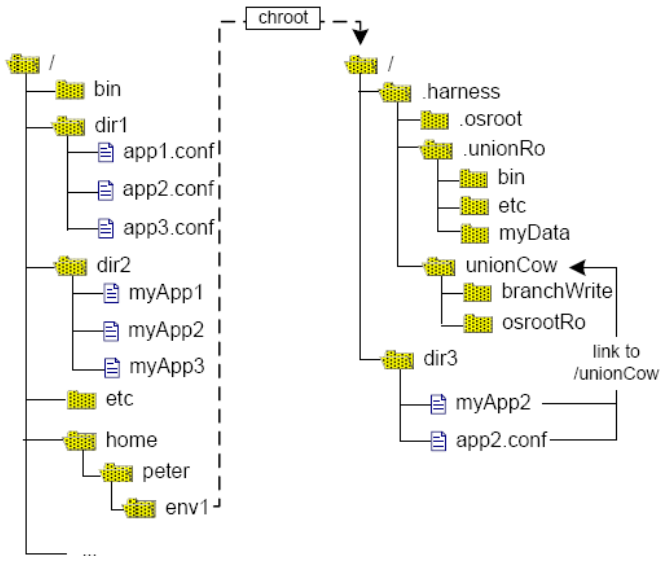

Figure 5.11: File Integration via Copy-On-Write Union

Figure 5.11 shows the file integration of `app2.conf` and `myApp` via Unionfs with copy-on-write function. The root directory is three times mounted in the virtual file structure, in `.osroot`, in `.unionRo` and in `.unionCow`.

This construction makes it possible to integrate each file of the entire file system with the appropriate permissions multiple times on different locations by creating a symbolic link to `.osroot`, to the readonly `.unionRo` or to the copy-on-write `.unionCow` and as well by copying a file.

The Unionfs copy-on-write method offers the advantage that a file will be copied first, when it really needs to be manipulated. Furthermore, the creation of a virtual

file structure takes less time with the Unionfs method. The creation of a Union and the symbolic links for each file needs less time than copying all the required files. On the other hand I/O activities of programs in the virtual environment during runtime are processed with a lower performance if Unionfs is used.

However, the use of the Unionfs method instead of the copy method for combination 2 and 3 offers another possibility. The copy method can be used if a user or administration knows exactly, which files and directories are required for a program. The Unionfs method can be preffered if it is not exactly known, which files should be modifiable.

Furthermore, Unionfs can be used to analyze what files are manipulated by a program. For this purpose the entire virtual file structure is integrated via the symbolic links to the readable and writable union in the virtual file structure. Then the program is executed in this virtual environment and processes a test to induce I/O operations. After this test the user can see all the manipulated files in the readable and writable branch of the union.

In conclusion, the copy method is required to block information interchange and to enrich the "sandbox" characteristic of a virtual environment. The link method makes a virtual environment more flexible and saves disk space. New system installations and updates are directly adopted in the virtual environment. Unionfs can be used to limit the write permissions to realize a one way information interchange and can be used instead of the copy method if it is not clear which integrated files will be manipulated by the programs running inside the virtual environment.

The use of the copy and link method is very portable and the operating system does not need to be reconfigured. In contrast to Unionfs, which needs to be installed as a kernel patch, but the development of an out-of-the-kernel module is planned.

- **Technical Addition**
  The described Unionfs method cannot be implemented without adaptations. The mentioned unions have to be split, because Unionfs is not able to create a union, which consists other unions. A union which contains the root directory cannot be part of a union because the union itself is located somewhere in the root directory. A union cannot contain a union even not itself. Therefore, every union is split into a small number of unions, one for every main directory of the root directory, shown in Figure 5.12.

  Furthermore, each of the copy-on-write unions require also its own intermediate extra branch for the copy-on-write function. According to Figure 5.12, the entire structure is located under the directory `.harness`.

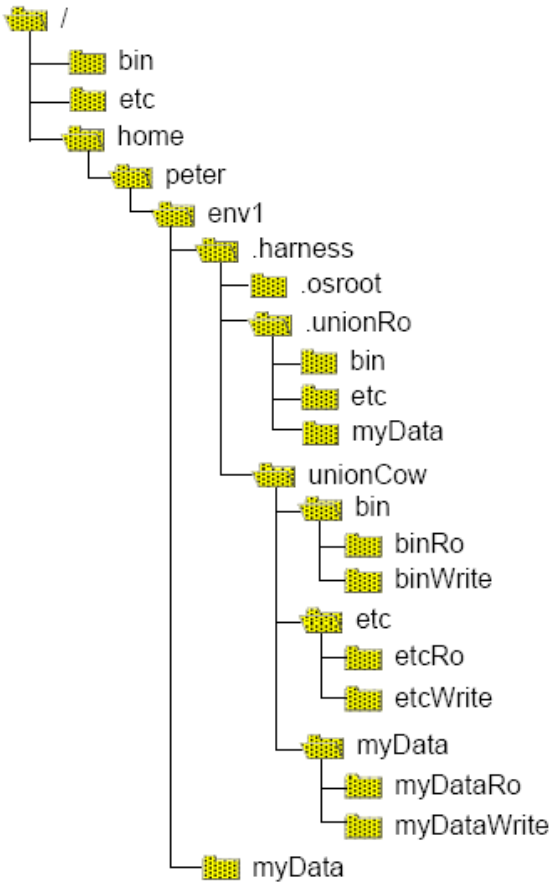

Figure 5.12: Split Unions

5.2 Environment Variables

As mentioned, beside file structures this dissertation project is also focused on the virtualisation of environment variables or rather shell environment variables [70]. Environment variables are used by programs and the shell itself to get information about the environment and user specific settings. A lot of programs need to know, what kind of terminal is used. For instance, some Unix tools need to know the favorite editor. The start parameters for such a standard editor can be read from environment variables even the user's preferred text color can be find if specified.

In contrast to "normal" shell variables, environment variables are bequeath by every process, which is started from a shell. Nearly all programs on Unix-type operating system are started from shells. Therefore, environment variables are important for environment virtualisation.

Environment variables are easy implemented. The data type is always string. The data structure is as easy as possible. Complex data structures are not known. For a shell a variable has a name and a string value. The interpretation of these variables depends on the programs using them. For instance, the predefined `PATH` variable contains a string, which contains a list of search paths, each separated with a ":".

For the programs this string is an array or list of strings and it has to know, what character is used as the separator. For the shell it is only a simple string. This is common to all widespread Unix/Linux shells.

It is a convention that environment variable name consists only of capital letters, but there is no special restriction about variable names. They can be created with any arbitrary characters. Environment variable names are case-sensitive. The mentioned predefined variable `PATH` is not overwritten by a new variable named path.

For these reasons, the environment description concept offers no possibility to describe data types or data structures for environment variables. The description concept supports the determination of variable `name` and `value`.

Furthermore the concept provides the description of manipulations of existing environment variables. This is useful, if an environment description inherits another one. For instance, adding a new path in the mentioned environment variable `PATH` can require the determination of the position on which the new path should be added. This could be at the beginning, at the end or somewhere between. For this, the determination of the separator character is required as well. The manipulation includes the possibility to eliminate existing environment variables, which is not equivalent to a variable with an empty value.

5.3 Performance Tests

Tests are required to analyze the performance of the virtualisation approach, and to estimate its usability for scientific high performance computing. The performed tests are used to measure the creation time of virtual file structures and the runtime performance of programs running in virtual environments. For the comparison of the results, all tests are performed on the same hardware with same installed software. The test platform is a dedicated desktop system, which is typically used for software development.

- **Hardware**
  - CPU: Dual Intel Pentium D CPU with 3.40GHz
  - Harddisk: Western Digital wd2500JS with 250 GByte, 8 Mbyte Cache, SATA II, and 7200 U/min (wd2500JS), formatted with Ext3
  - RAM: 4 GByte
  - SWAP: inexistent and not required
- **Software**
  - Operating system: Kubuntu6.06 LTS (2.6.15)
  - C compiler: GNU gcc version 4.0.3-1ubuntu5
  - File system: Ext3 and Unionfs 1.3

5.3.1 File Structure Creation

The veCreator shell script creates virtual file structures. The time the veCreator takes depends on the number of files, which should be integrated, and the used file integration methods. The test involves the integration of 32,935 files out of the directories `/bin /lib` and `/sbin`. Typically, more than one file integration method is used to integrate this amount of files.

The creation time can vary depending on the distribution of the used file integration methods. The following tests are performed with extreme distributions to measure the speed of each file integration method in extreme cases.

Therefore, the first tests integrates all files via the copy method, the second one uses the link method. The last test uses Unionfs with copy-on-write function.

Obviously the copy method is the slowest method. Figure 5.13 shows that it takes 65 seconds to copy all the files, while the other two methods take only 5 to 6 seconds.
Link and Unionfs integration take the same time, because in these cases it is required to create links for integrated files, only the link source differs.
The creation time answers the question, how virtual environments can be used in practice. There are two possibilities. The first one creates the file structure with every start of

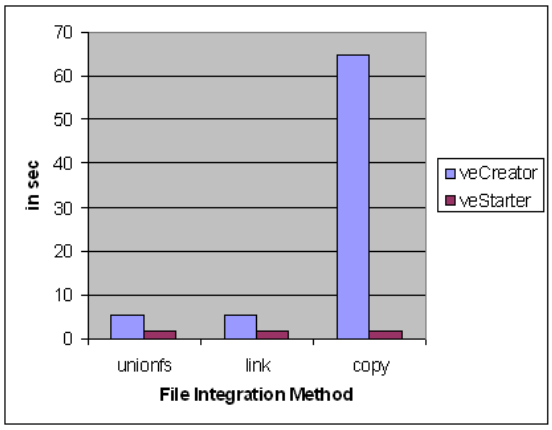

Figure 5.13: Performance of veCreator and veStarter

a program in a virtual environment. Each time, the veStarter executes first the veCreator.

The second one has session character. The virtual file structure is created once, at the beginning of the session. Starting applications with the veStarter do not involve the veCreator.

The results show, that the second use is more effective for bigger environments. The first one could take to much time, if large files are integrated via copy method. On the other hand, it is an extreme case to copy every file and not using the link method. For the most cases, it is more realistic to assume, that most files are integrated via link or Unionfs.

5.3.2 veStarter Performance

Starting an application in the virtual environment requires some preparations. Before an application is started and "chrooted", the veStarter program manipulates environment variables, and has to mount the root directory and the unions.

The test involves the manipulation of 10 to 20 environment variables. The number of mounted directories do not vary. The current implementation of the veStarter takes always 1.8 seconds, indepent from the number of environment variables, see Figure 5.13.

5.3.3 Runtime Performance

Runtime performance tests are required to measure the performance lack of a virtual environment, and to compare the three file integration methods.

Obviously, a lower runtime performance of a virtual environment, which emulates an original environment can only be caused by I/O activities. More CPU-intensive programs cause a less loss of runtime performance.

An I/O intensive program "chrooted" in a virtual file structure, which includes only copied files, has the same performance than running without being "chrooted". A virtual file structure, which contains only symbolic links has nearly the same performance. A small benchmark script, which opens, in a loop, files or links causes that symbolic links are slower, but less than 1 %, see Figure 5.14.

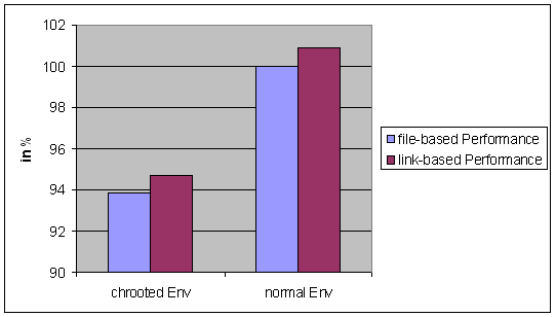

Figure 5.14: File and Link-based Environments With and Without Chroot

Each benchmark was performed 30 times for files and links inside and outside a virtual environment. Surprisingly, the performance inside the virtual environment is fast (6 to 7 %). Both, the copy and link method can be used without notable performance loss. Therefore, all three benchmarks are focused on the performance of Unionfs to compare it with Ext3.

Three benchmarks were used. The first two benchmarks, Iozone [71] and Postmark [72], are special I/O benchmarks. The third benchmark is a compilation of kernel source code, which is more CPU-intensive.

Iozone Benchmark

Iozone is a powerful I/O benchmark that generates and measures a variety of file operations. The Iozone can be used to generate the following file operations: read, write, re-read, re-write, random read and random write, random mix, backwards read, recond re-write, strided read, Fwrite and Frewrite, Fread and Freread.

The benchmark is configured to generate all this file operations for file sizes of 64 Kbyte to 4 Gbyte and transfers for each file record sizes of 4 Kbyte to 16 Mbyte.

Figure 5.15 shows the read performance diagrams for Ext3 and Figure 5.16 of Unionfs. The diagrams show the read Kbytes per second, depending from record size and file size. Not measured combinations are display with zero values.

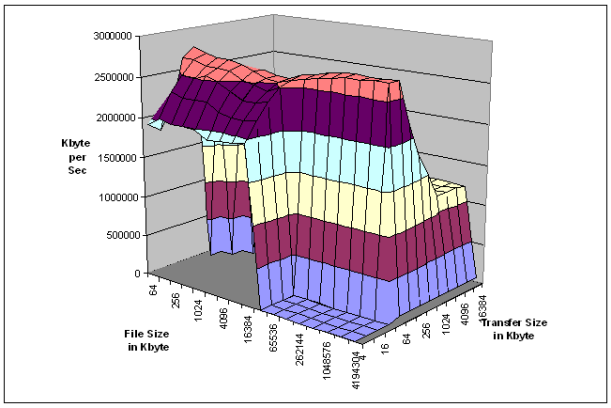

Figure 5.15: Read Performance of Ext3

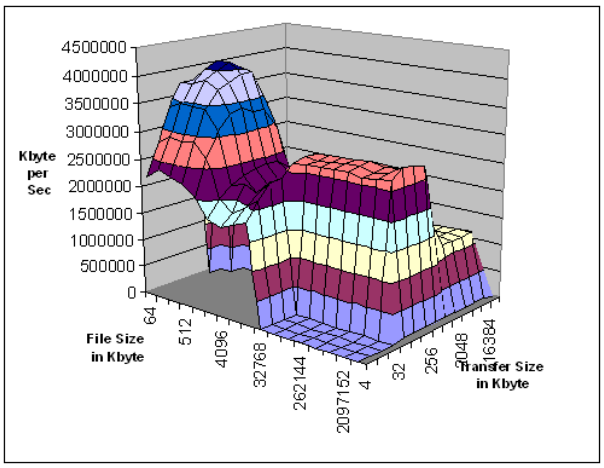

Figure 5.16: Read Performance of Unionfs

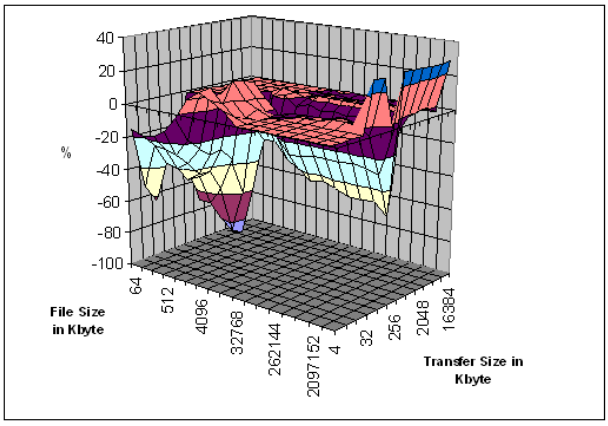

Figure 5.17: Relative Read Performance of Unionfs to Ext3

Figure 5.17 shows the percental difference diagram for Unionfs. Every value over zero shows that Unionfs is faster than Ext3 for the respective combination. As assumed, Unionfs is mostly slower than the Ext3 especially for smaller files. Due to different, caching approaches Unionfs is faster in some combinations. The average percent of all measured values of this diagram results that Unionfs is 12.17 % slower than Ext3. Figure 5.18 shows the average percent of all four test. Surprisingly, Unionfs is 1.83 % faster for write operations and that in series. The concrete reason for is is not known, but it could be caused by buffer caching.

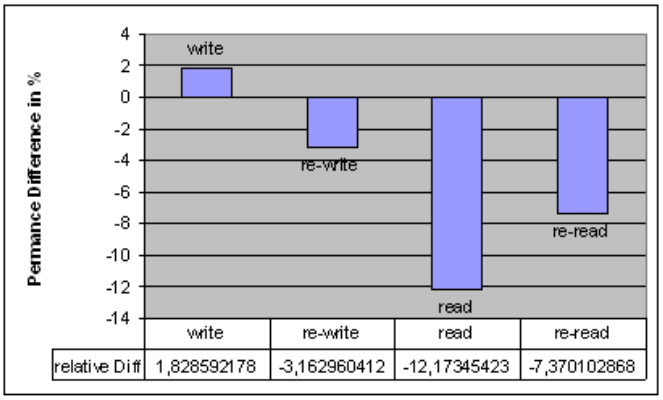

Figure 5.18: Relative Performance Differences of Unionfs to Ext3

Figure 5.19 shows the percental difference diagrams for the file operatings re-read, write and rewrite.

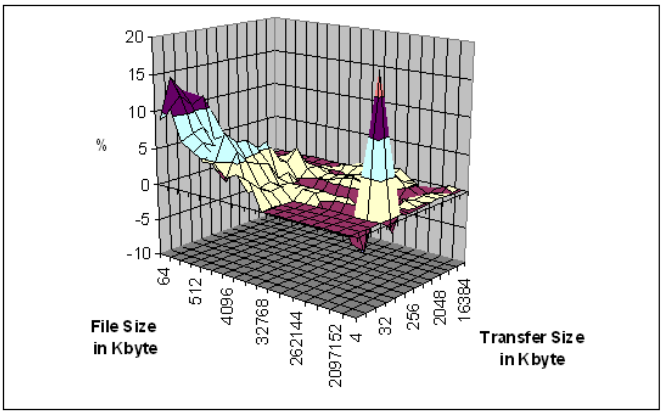

Figure 5.19: Relative Read Performance of Unionfs to Ext3

Postmark Benchmark

Another I/O benchmark is postmark. Postmark v1.5 simulates the operation of electronic mail servers. It performs a series of file system operations, like appends, file reads, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. The benchmark was configured to create 20,000 files, between a size of 512-10,240 Kbytes, and perform 200,000 transactions. 200 subdirectories were used to prevent linear directory look ups from dominating the results. All of the branches were read-write, to distribute the load evenly across branches.

The test results were published [64] by the Stony Brook University. Figure 5.20 shows the performance results for Unionfs working with unions comprising of 1 to 8 branches. This is different to all other benchmarks and demonstrated that the performance of Unionfs decreases with the number of branches included in a branch. With an overhead of 64 to 71.7 % Unionfs is much slower than in the other benchmarks, but this cannot be compared because the benchmark was performed on another platform (e.g. using Ext2 instead of Ext3).

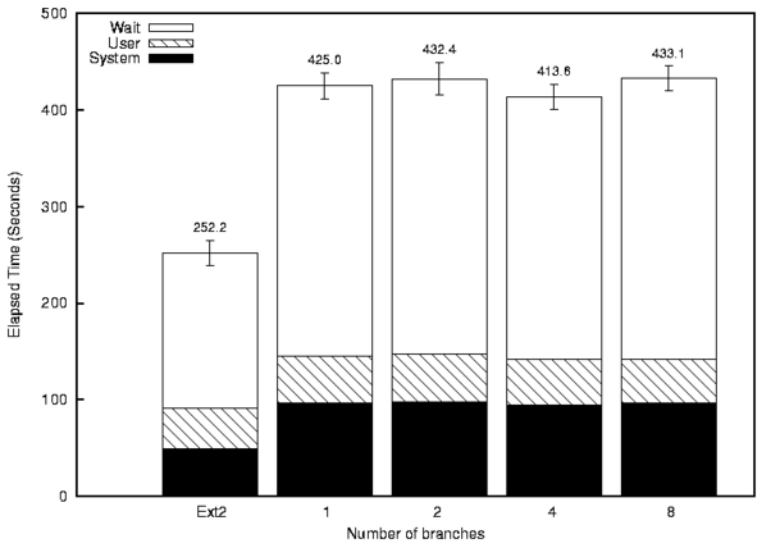

Figure 5.20: Postmark Performance Results with 20,000 Files and 200,000 Transactions [64]

However, these results demonstrate that Unionfs has the best performance with unions consisting of 1 or 2 branches. The file integration method of this dissertation project uses Unionfs with unions comprising 1 (readonly) or 2 (copy-on-write) branches. The developed Unionfs file integration method offers the highest runtime performance, which is possible with Unionfs.

Kernel Source Compilation

Compiling is a convenient benchmark because it is more CPU than I/O-intensive, and represents the workload characteristic of a software developer better than Postmark and Iozone. Therefore, the results of this benchmark can be seen as a better estimation for the all day usability of the Unionfs file integration method.

The benchmark was performed with the shell script:

```
cd /usr/src/linux-source-2.6.15
cp /home/user/.config .config
make oldconfig
sudo make -j 2
```

The benchmark was tested on Ext3, and with Unionfs overlying on Ext3. The test results are very stable and there is no result, which differs more than a second from the average. Figure 5.21 shows that Unionfs is only 0.55 % (6 sec) slower than the Ext3. This result shows that the Unionfs approach is definitely useable for non I/O-intensive programs.

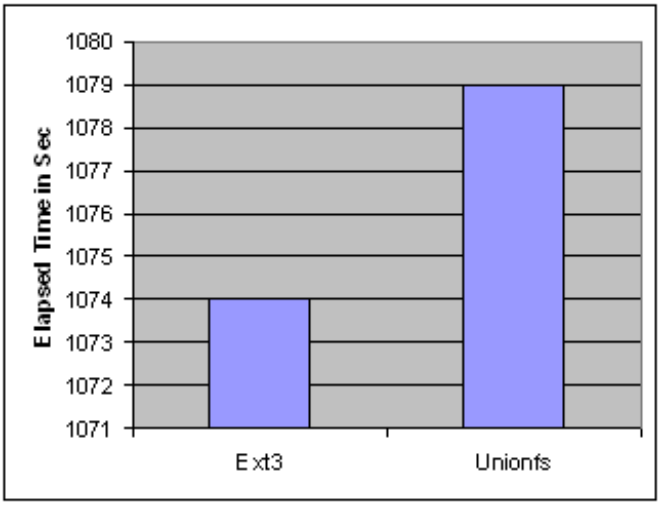

Figure 5.21: Kernel Compiliation with Unionfs and Ext3

6 Detailed Software Design

This chapter describes the detailed XML implementation of the concept for virtualised file structures and file integration as well for environment variables. It follows a description of an equivalent syntax for config files, which is able to represent the same information. Furthermore, it is described, how the shell script applications veCreator and veStarter read and process environment description files, and what security issues and implementation challenges are involved.

6.1 XML Concept for Virtualised File Structures and File Integration

There are two criteria for the concept: extensibility, and usability. Extensibility means the core structure of the concept has to be constructed to allow inclusions of additional elements at a later point. Usability deals with the question, how convenient the concept can be used in practice. Therefore the concept is balanced between extensibility and usability.

According to Section 5.1, XML elements are required, whose structures contain all the properties of files and directories. Therefore, the XML elements file, directory and branch are described in the following. The determination of the file integration method follows afterwards.

6.1.1 File Element

A file element describes a single file. Depending on the use, a file element requires different attributes. The XML implementation of this element is shown in the following example and contains minimum the both attributes name and permission.

```
<file name="/etc/test.conf" permission="755">
</file>
```

The permission attribute contains three digits to describe the directory permission in a Unix-like way. This assumes, that only Unix-type permissions are used and required. New The content of the name attribute depends on the use of the file element.

However, the way how attribute values are stored in the above example needs to be changed for the following reasons.

An attribute is a variable, which contains meta data about an element. The use of attributes leads to disadvantages in XML. Attributes cannot contain multiple values and they are not easily to expand for future changes, because an attribute cannot describe a structure only simple data.

Furthermore, attributes cannot be easily validated against Document Type Definition (DTD). DTD is used to define legal elements to check if an XML document conforms to or breaks the defined document rules. At last, it is difficult to manipulate attributes by program code. Therefore, child elements are used instead of attributes as illustrated below. However, these meta data is named attributes to underline their meta data character, but in XML it is implemented as child element.

```
<file>
      <name>/etc/test.conf</name>
      <permission> 755 </permission>
</file>
```

A file element can be a standalone file element, or a child element of a directory or branch element.

A standalone file element can be used to select a single file from a source to integrate it in a directory of a virtual file structure. The standalone file element has the additionally attribute source.

```
<file>
<source>/home/paul/test.conf</source>
      <name>etc/test.conf</name>
      <permission> 755 </permission>
</file>
```

The name attribute defines the target location and contains path and name of the file in the virtual environment. As shown in Figure 6.1, this path is related to the root directory of the virtual environment and can be named the virtual absolute path. The source attribute contains the source path including the original name of the source file. This source path is the absolute path according to the root directory of the operating system. Missing directories will be created implicitly if the target directory does not exist in the virtual file structure.

File name in the name attribute can differ from the original name of the file in the source attribute. Therefore, a file can appear with a different name in the virtual file structure. A file is integrated with the name of the source file, if the name attribute is missing.

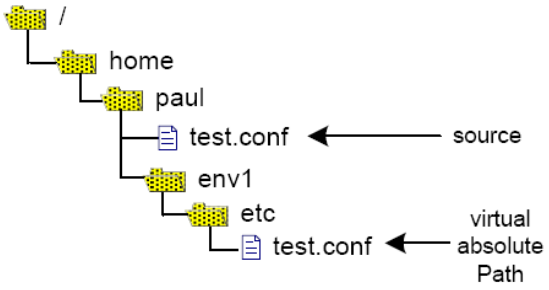

Figure 6.1: Source and Target of a "Standalone" File Element

Finally, it is required to determine, which file integration method should be used for a file. The integration attribute can set to `copy` for the copy method, to `link` for the link method, to `unionfsCow` for the integration via a union with copy-on-write function, and `unionfsRo` for the integration via a readonly union. These attribute values are very specific because Unionfs would not stay the only proper stackable file system.

However, the use of the permission attribute depends on the used file integration method. Only the copy method allows to change the permission of the integrated file. It can be skipped, if one of the other file integration methods is used.

The use of the file element as part of a directory and branch element is described in the following sections.

6.1.2 Directory Element

A directory element describes a directory, which has to be created in the file structure of the virtual environment. Like a file element, a directory contains the attributes name and permission, as shown in the following command lines.

A directory element does not describe a source of files, which can be integrated in the file structure of a virtual environment. Therefore, the name attribute contains the virtual absolute path of the target location in the virtual file structure. The permission attribute contains three digits and is used the same way like the file's permission attribute.

```
<directory>
          <name>etc/myDir</name>
          <permission> 755 </permission>
</directory>
```

It is possible to describe an entire virtual file structure only with the aforementioned standalone file elements, but this would be a burden, because every single file needs to be described, and the description file would not be easy to overlook. Therefore, a directory element is a container, which can include any child elements, like file elements,

branch elements and other subdirectory elements.

File elements can be placed into a directory element to put several single files with different sources together in one directory. The target location of these files is determined in the name attribute of the directory element, in which they are contained. In this case, the name attribute of the file element contains only the file name without the path. As mentioned, the name attribute is only required, if the file should be integrated with a different name.

Below a directory element is shown, which contains two file elements. The first file will be copied, and the second one will be integrated via link with a different name.

```
<directory>
          <name>lib/myDir</name>
          <permission> 755 </permission>
          <file>
                <source>lib/test.conf</source>
                <permission> 755 </permission>
                <integration>copy </integration>
          </file>
          <file>
                <source>lib/test2.conf</source>
                <name>newName.conf</name>
                <integration>link </integration>
          </file>
</directory>
```

It is possible to define default settings in a directory element valid for its child elements. This makes the description text shorter and clearer. The directory attribute `umask` can be used to determine the default file permission. This `umask` attribute is valid for file elements with a missing permission attribute. It is possible to distinguish default permissions for files and subdirectories. Similar to the `umask`, the `dirUmask` attribute can be used to determine the default permission for the subdirectories of a directory.

Furthermore, directory element provides an own integration attribute to determine the default file integration method.

```
<directory>
          <name>lib/myDir</name>
          <permission> 755 </permission>
          <umask>755</umask>
          <integration>copy </integration>

          <file>
               <source>lib/test.conf</source>
          </file>
          <file>
```

```
            <source>lib/test2.conf</source>
            <name>newName.conf</name>
            <integration>copy </integration>
        </file>
</directory>
```

6.1.3 Subdirectory Element

A directory element can contain subdirectory elements. A subdirectory element is a directory element, but it is called subdirectory, if it is included in a directory or another subdirectory element. As well it is provided to determine all the default values for the included files and subdirectories. These subdirectory defaults overwrite the parent directory defaults. A subdirectory element can also include file, subdirectory and branch elements.

The only difference is, that the name attribute of a subdirectory element is interpreted as a relative path in relation to its parent (sub)directory element. Normally, the subdirectory's name attribute contains only its name, but relative paths are possible. This makes it easier to move entire directory structures by editing only the name attribute of the highest directory element of the respective directory structure.

```
<directory>
        <name>lib</name>
        <permission> 755 </permission>
        <umask>755</umask>
        <integration>copy </integration>

        <file>
            <source>lib/test.conf</source>
        </file>

        <subdir>
          <name>app1/source</name>
          <file>
              <source>lib/test2.conf</source>
              <name>newName.conf</name>
              <integration>copy </integration>
          </file>
          <subdir>
           <name>version</name>
          </subdir>
        </subdir>
</directory>
```

Figure 6.2 illustrates the result of this code above.

6.1.4 Branch Element

If it is required to include many files from the same source into a directory, it is bothersome to retype the whole source path in the source attributes for each file element. To

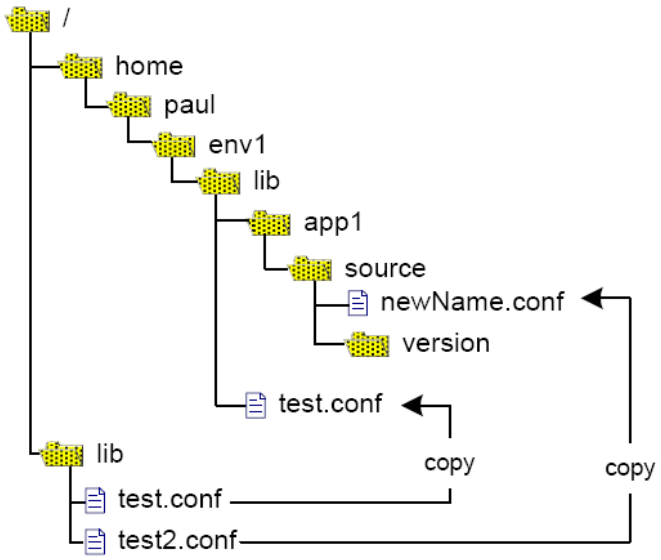

Figure 6.2: Subdirectories in Virtual File Structure

avoid this, the concept supports with the use of branch elements an easy way to merge one, two or more different directories into one directory in the virtual file structure.

The source attribute of the branch element contains a path to the source directory. This source path is the absolute path according to the root directory of the operating system. With the attributes `getAll` of a branch element, it can be determine, what files should be integrated out of the branch source.

```
<directory>
          <name>project</name>
          <integration> link </integration>
          <branch>
                 <source>dir1/</source>
                 <integration> copy </integration>
                 <getAll>files</getAll>
          </branch>
</directory>
```

The result of the code above is shown in Figure 6.3.

The `getAll` attribute can be set to `files`, if it is required to integrate all files from the source directory of the branch element, as shown in Figure 6.3. This does not include the subdirectories and their files. The `getAll` attribute has to be set to recursive to integrate recursively all files and subdirectories of a branch source. The veCreator application creates all the subdirectories in the new directory and integrates its files, see Figure 6.4.

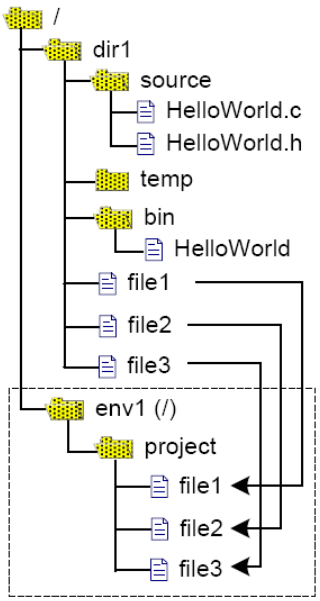

Figure 6.3: File Integration with getAll Attribute

```
<directory>
          <name>project</name>
          <integration> link </integration>
          <branch>
                 <integration> copy </integration>
                 <source>dir1/</source>
                 <getAll>recursive</getAll>
          </branch>
</directory>
```

The result of the code above is shown in Figure 6.4.

A branch element can contain an integration attribute to support as well the determination of the file integration method. As shown in the following example, the files of the first branch are integrated with the file integration method, which is defined for its branch. The files of the second branch are integrated according to the defaults of the directory element, because it does not contain an integration attribute.

Like in directory elements, it is possible to define default file and subdirectory permission. A branch element can contain an `umask` and a `dirUmask` attribute. Attributes of a branch element are default values for included file elements. Default values can be overwritten by adding the respective attribute in the file element.

In a second step, the amount of files defined by the `getAll` attribute can adjust with file elements. A branch element can contain any file elements. Each file element in a branch

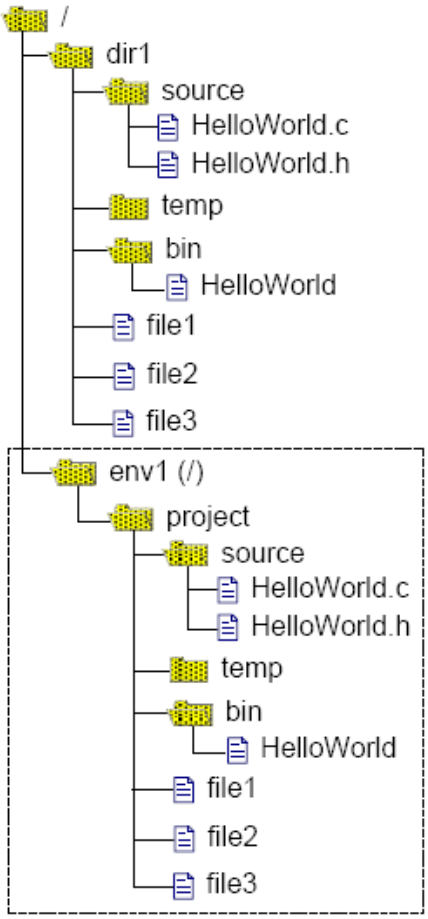

Figure 6.4: Recursive File Integration with getAll Attribute

can redefine the properties of a specific file, which is included in the `getAll`-amount.

This third kind of file element has no source attribute, it is not needed. The source attribute determines, for which file out of the defined set the file element is valid for. The source attribute contains only a name, but no path. For instance, it is possible to redefine the integration method for an important file to copy, when all other files of the branch are integrated via link method.

It is also possible to delete one or more files from the defined set of files. This can be realized by including a file element for the respective file and setting its integration attributes to block. Such a file will be blocked and not integrated.

```
<directory>
          <name>project</name>
          <permission> 751 </permission>
   <!--------------------branch section------------------------>
          <branch>
                 <source> dir1 </source>
                 <integration>link</integration>
                 <getAll> files </getAll>
                 <file>
                      <source> secret.conf </source>
                      <permission> 700 </permission>
                      <integration> copy </integration>
                 </file>
          </branch>
          <branch>
                 <source> dir2 </source>
                 <integration> link </integration>
          </branch>
   <!--------------------file section--------------------->
          <file>
                 <source>lib/test.conf</source>
                 <permission> 755 </permission>
                 <integration> copy </integration>
          </file>
          <file>
                 <source>lib/test2.conf</source>
                 <name>newName.conf</name>
                 <integration> link </integration>
          </file>
</directory>
<file>
<source>/home/paul/test.conf</source>
      <name>etc/test.conf</name>
      <permission> 755 </permission>
</file>
```

Merging more than one branch in a directory can cause file name conflicts between files with the same name of different branches. Similar to Unionfs, this can be avoided with

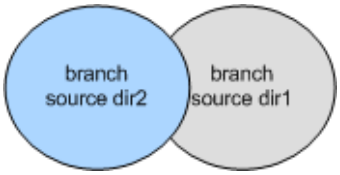

Figure 6.5: Branch Priority System

a simple priority system. Each branch gets a priority. The priority of each branch is determined implicitly via its position in its directory element. The first branch gets the lowest priority. In case of a name conflict, the file from the last positioned branch in the directory element will be integrated. According to the above code see Figure 6.5

6.1.5 Usability Aspects

The description concept allows flexible redefinitions of already described directories or files. A directory `/etc` can be described by a directory element, and can be manipulated by another directory element. Furthermore, there is no restriction to place and describe subdirectories like `lib` only with subdirectory elements. It is possible to describe this subdirectory as well with a directory element. Only the name attribute has to contain `usr/lib` instead of `lib`.

Directory elements, all kinds of file elements, and even branch elements can be used to modify or delete existing instances, for example by setting the integration attribute to delete.

In addition, it should be underlined, that a directory can be integrated recursively as well with a file element. This is the only way to integrate an entire directory via a symbolic link.

```
<directory>
<name>usr</name>
<file>
<name>lib</name>
<integration>link</integration>
</file>
</directory>
```

A Uniform Resource Locator (URL) can be used to determine, for instance, a file's source if it is located ouside the local file structure.

```
<file>
      <source>ftp://paul:<Passwort>@server/home/paul/test.conf</source>
      <name>etc/test.conf</name>
      <permission> 755 </permission>
</file>
```

6.1.6 Inheritance

According to the concept specifications in Section 3.2.2, the concept provides environment descriptions to inherit from one another. Therefore, the mentioned env element can contain any `include` attribute. Such an `include` attribute contains the path to another environment description file. The content of this file will be bequeath. The environment desciption of last positioned include attribute is bequeath at last.

```
<env>
<name>~/env2</name>
<include>/harness/basic.conf</include>
<include>/harness/mpi.conf</include>
</env>
```

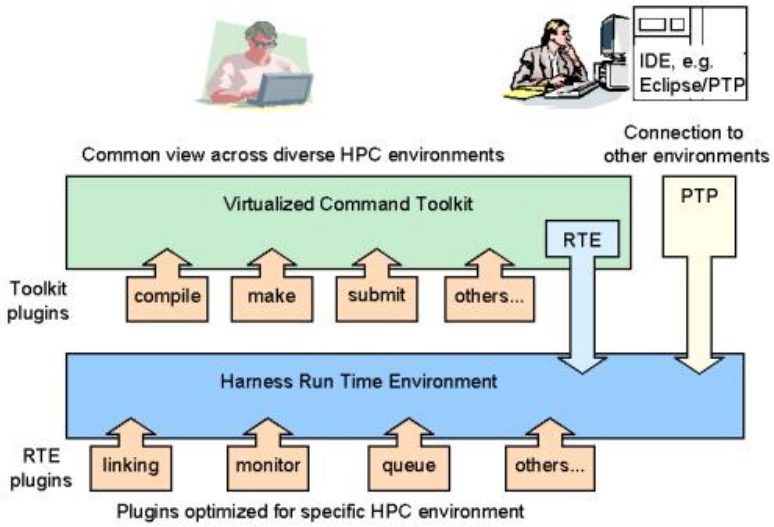

Figure 6.6: Inheritance of Environment Descriptions

Inheritance improves the usability and gives system administrators the influence to the user's virtual environments.
The inheritance can be defined more detailed for every element to ensure secury issues. Environment, file, directory and branch elements can contain an inheritance attribute which can be used to determine if an element can be modified by the environment which inherits them. The inheritance attribute can be set to static or modify. static allows no changes of the respective element. New files cannot be added to a static directory. The permissions and ownership of the directory cannot be changed as well. modify allow to change all properies even deleting. The default setting is modify.

```
<env>
<name>~/env2</name>
<inheritance></inherit>
```

```
</env>

<directory>
<name>etc</name>
<inheritance>static</inherit>
</directory>
```

6.1.7 Creation Order of the File Structure Elements

The creation process creates the elements in the following order: directory, branch, file elements in branches, file elements in directories, and at last standalone file elements.

No matter, if standalone file elements are positioned at first or mixed between directory elements, they are processed at last. The creation order of elements included in the same parent element, is based on the document position of these elements.

According to this, the program starts with the creation of the first positioned directory element. Then it integrates all the files of the first positioned branch element included in this directory element, if there is one. After that, if existing, the file elements included in this branch element are integrated, as well in the order of their positions to each other. Then, in position order all the other branches are processed in the same way. The files of the last positioned branch element can overwrite the file integrated before. This is an easy way to follow the branch priorities without checking for file name conflicts.

At next, all file elements, included in the directory element, are processed in document position order. At last, each included subdirectory element of this directory element is processed recursively in the same way like a directory element.

After processing the first directory element and its subdirectory elements recursively, the second positioned directory element is processed and so on. The first positioned standalone file element will be created after processing all directory elements.

6.1.8 Environment Element

The environment element env can be used to determine the absolute location of the virtual root directory of the virtual file structure. More precisely, the name attribute of the environment element contains the absolute path to the virtual root directory. The entire environment location can be moved to another location by changing this path.

The env element can be also used to determine default values, like umask, dirUmask, integration method. Furthermore, the env element includes configuration attributes. The aforementioned directory .harness which contains, for instance, the .osroot directory can be modified if required.

```
<environment>
<env>~/env1</env>
<umask> 755 </umask>
<dirUmask> 755 </dirUmask>
<integration> link </integration>
<mountPoint> .harness <mountPoint>
</environment>
```

The environment element offers `|integratAlt|`attributes to ensure that an environment description is still processable on operating system using no Unionfs. If the integration method of a file is determine with `unionfsRo` or `unionfsCow` it cannot processed. It is required to define an alternative, for instance `copy`. Therefore, the `|integrateAlt|`attribute can be used follows.

```
<environment>
<env>~/env1</env>
<integratAlt>unionfsCow copy</integratAlt>
<integratAlt>copy unionfsRo</integratAlt>
</environment>
```

It is possible to define several alternatives. If Unionfs is not availalbe each file or directory which should be integrated via unionfsCow will be copied.
For a more detailed determination for each file is also possible. As shown below the integration attribute can contain a second value, even in directory and branch elements.

```
<directory>
<name>bin</name>
<integration>unionfsCow link</integration>
<file>
<name>su</name>
<integration>unionfsCow copy</integration>
</file>
</directory>
```

6.2 XML Concept for Environment Variables

An environment variable can be defined with a `var` element. A `var` element can contain the following attributes and child elements name, value, action, and insertPosition. Name contains the name and value, the value of the environment variable. Below the definition of the `PATH` variable is shown.

```
<var>
    <name>PATH</name>
    <value>/home/user/apps</value>
</var>
```

This example leads to a definition of an environment variable and can overwrite already existing variable of the same name. The `action` attribute can be used to determine, whether a `var` element should be interpreted as a manipulation or a definition of a variable. The `action` attribute can be set to `unset`, `new` or `modify`.

Setting the action attribute to `new` leads to the declaration and initialization of the variable with the value of the `var` element and ensures that already existing variables with the same name are overwritten. The default value of the `action` attribute is `modify`.

If the `action` attribute is set to `modify`, the value of the `var` element will be added to the value of an existing variable with the same name. In this case, the `insertPosition` attribute can be used to determine, at which position the `value` attribute should be inserted. For instance, this feature is useful to change the order of search paths. Setting the `insertPosition` attribute to `prepend` adds the value at the first position, and setting it to `append` adds the value at the end. The position can also be determined more detailed with position numbers. Number 1 is equal to `prepend`. The `insertPosition` attribute is optional. A value will be insert per default at the end if the attribute is missing.

```
<var>
    <name>PATH</name>
    <value>/home/user/apps</value>
    <action>modify</action>
    <insertPosition>append</insertPosition>
</var>
```

The string of the `value` element needs to be separated into fields to identify the given insert position, except the positions prepend and append. The separator attribute can be used to determine the separation character. The default separation character is a colon.

The manipulation of existing variable values requires some knowledge about the current content. In some cases the exact insert position is not known or variable, depending on the current content of the `value` element. The `insertPattern` attribute contains a search pattern, and can be used to define the insert position, depending on the position of a matched field in the string.

As well, it is possible to define the insert position relative to the matched position of the `insertPattern` attribute. The insert position of the `insertPosition` attribute is related to the matched position of the `insertPattern` attribute if it is used. The follwing example shows, how to add a new value two positions before the matched pattern. The minus and plus characters can be used to determine, whether the position is before or after the matched position.

```
<var>
    <name>PATH</name>
```

```
    <value>/home/user/apps</value>
    <action>modify</action>
    <insertPattern>/bin</insertPattern>
    <insertPosition>-2</insertPosition>
    <separator>:</separator>
</var>
```

The `insertPosition` attribute could be irritating, because it seems to be the insert position of the entire environment variable, or rather the order of the variable creation. The correct structure of the `var` element is illustrated next. The `insertPosition` attribute and as well the `insertPattern` and `separator` attributes are included in the `value` element, because they are related to the `value` and not to the `var` element.

```
<var>
    <name>PATH</name>
    <value>/home/user/apps
          <insertPattern>/bin</insertPattern>
          <insertPosition>-2</insertPosition>
          <separator>:</separator>
    </value>
    <action>modify</action>

</var>
```

The order, in which variables are created is determined by the position of the `var` element in the environment description file. First positioned `var` element is created at first.
A `var` element will be declared and initialized as a new variable, if the `action` attribute is set to `modify` and no variable with the same name exists.

If the `action` attribute is set to `unset`, the variable will be eliminated, if it exists. A variable with an empty value is not unset, it still exists.

```
<var>
    <name>PATH</name>
    <action>unset</action>
</var>
```

In addition, it is possible to define the value of a new environment variable with the value of an existing one. The example below describes the `MYPATH` variable, which describes the working directory of a user application. `MYPATH` consists of the predefined environment variable `USERPATH`, which contains a user's home directory, and appends path to the working directory. The $ character identifies a word as a variable. The appended text needs to be encapsulated with single quotes to be separated from the name of the variable.

```
<var>
    <name>MYPATH</name>
```

```
    <value>$USERPATH/myapps</value>
    <action>new</action>
</var>
```

At the moment these manipulations are powerful enough to satisfy the most needs to create the environment variables of a virtual environment. If it is required, the description possibilities for such manipulations can be extended and processed, for instance with Unix tools like `sed` and `awk`.

6.2.1 Config File Syntax

The two prototype applications veCreator and veStarter are implemented as shell scripts. To enable them reading an environment description from an XML file, it is required to program an XML parser in shell script code. This is a bothersome task, and can be avoided with the shell command source [73].

The source command is used to execute another shell script B from a shell script A. Such an executed shell script B can contain code of variable definitions. The benefit of the source command is, that shell script B is executed by the current shell, and not as normal by a child process of the current shell. Therefore, the variables exist still after the execution of shell script B, and can be used by shell script A. Thus, a shell config file is a shell script file, which contains only variable definitions.

Using the source command is a good way to avoid writing an XML parser. A shell-script-conformable syntax is required, which is able to represent all information from the described XML elements.

There are three problems to realizes this:

1. in XML, attributes and child elements are assigned to a parent element, because they are encapsulated. The document position of an attribute is used to assign it. By using the source command, all information about the attribute and element positions is lost.

2. a variable cannot be used, if the variable name is not known to the shell script (A). The source command does not result a list of the new variables or something else, from which variable name or even its order can be derived.

3. in XML code the names of attributes and elements are not unique. For instance, each directory element has the same name: "directory". This is in opposite to the config file syntax. Each name of a variable has to be unique, otherwise the value of equal named variables would be overwritten by each other.

Therefore, the meta information, which assign attribute values to their parent elements has to be attached to or in the variable names. The attached meta information should ensure unique variable names. The rule, which describes the attachment of the meta information can be used to derived the variable names from a given start information, otherwise the shell script is not able to use them.

In the following example shows XML code and the equivalent config code. The variable `DIRS` contains a list of words or rather paths separated by spaces. The name of the `DIRS` variable is the given start information the shell script has to know. The shell script can derived the variable names of the following variables from the content of the `DIRS` variable. The first path of the `DIRS` list is the prefix of the name of the name attribute from this directory element.

——————————XML Code——————————————

```
<directory>
          <name>/etc/myDir</name>
          <file>
               <source>/etc/test.conf</source>
               <permission> 755 </permission>
               <integration>copy</integration>
          </file>
          <file>
               <source>/home/paul/test2.conf</source>
               <name>newName.conf</name>
               <integration>link</integration>
           </file>
</directory>
<directory>
          <name>/usr/bin</name>
          <file>
               <source>/home/paul/myCalc </source>
               <integration>link</integration>
          </file>
</directory>
```

——————————Config Code——————————————

```
DIRS="myLib etc"
myLib_NAME="/etc/myDir"

myDir_FILES="test.conf test2.conf"
test.conf_SOURCE="/etc/test.conf"
test.conf_PERMISSION="755"
test.conf_INTEGRATION="copy"

test2.conf_SOURCE="/home/paul/test2.conf"
test2.conf_NAME="newName.conf"
test2.conf_INTEGRATION="link"
```

```
etc_NAME="/usr/bin"
etc_FILES="myCalc"
test.conf_SOURCE="/home/paul/myCalc"
test.conf_NAME="calc"
test.conf_INTEGRATION="link"
```

All file elements of the first directory are listed in the `*_FILES` variable. Such a suffix exists for every attribute and child elements, and are required to be known by the processing shell script application. This name rule is similar and, for instance, is used by the GNU Autotools, but it is not sufficient for this purpose. The variable names are not unique. The file variable test.conf exists two times.

This could be changed by adding the entire path in the `DIRS` or `*_FILES` variable, see the example above, but this would lead to messy long strings. The special characters, which can be included in a path are the second problem of this name rule. For instance, it is not possible to declare a variable, which has a dot in its name (test.conf). At first it seems to be possible to avoid this with escaping or character replacing rules, but this leads to a bad usability. The config code would be hardly to read and to manipulate per hand.

This name rule has to be modified as well in face to usability reasons. It is planned, that these config files will be used like XML files. System administrators can create and use XML description files, parse them to config files, and use them for Harness. If there is no XML parser available, modifications can be done direct in the config file. Therefore, the usablilty should provide the creation of an entire environment description directly in config file syntax.

For instance, the `DIRS` variable in the above example should not contain each entire path or file name. As shown in the example below, the `DIRS` variable contains the number of all directories. The variable name of these directories comprises `DIR`, its number according to the position in the XML document and separated by underscores the name of the specific attribute. The shell script has to know the prefixes and suffixes for each element, e.g. `DIR` for directory, `_NAME` for its name attribute, `FILE` for files, and `BRANCH` for branch elements.

For instance, the shell script application reads in the `DIR1_FILES` variable how much files are contained in the directory. According to this number, it derives the variable names of the files. A variable name like `DIR1_FILE5` cannot be derived, if the number of files in `DIR1_FILES` is smaller 5. The application has to know all possible attributes of a file and has to check their existence. Not determined attributes will be initialized with default values.

```
DIRS="2"
DIR1_NAME="/etc/myDir"
DIR1_FILES="2"
```

```
DIR1_FILE1_SOURCE="/etc/test.conf"
DIR1_FILE1_PERMISSION="755"
DIR1_FILE1_INTEGRATION="copy"

DIR1_FILE2_SOURCE="/home/paul/test2.conf "
DIR1_FILE2_INTEGRATION="link"

DIR2_NAME="/usr/bin"
DIR2_FILES="2"
DIR2_FILE1_SOURCE="/home/paul/myCalc"
DIR2_FILE1_NAME="calc"
DIR2_FILE1_INTEGRATION="link"
```

The principle of this rule is appending the names of attributes and child elements at the end of its parent element's name, except subdirectory elements. In XML it is possible to encapsulate subdirectories elements into subdirectory elements in endless levels. This principle cannot be continued for subdirectory elements, because the maximal length of a variable name would be exceeded. Therefore, the variable names for subdirectory elements are created like directory elements. Only the relative path is replaced by the absolute path.

——————————XML Code——————————

```
<directory>
          <name>/home</name>
          <directory>
                    <name>paul/name>
                    <directory>
                              <name>apps</name>
                              <directory>
                                        <name>bin</name>
                              </directory>
                    </directory>
          </directory>
</directory>
```

——————————Config Code v1——————————

```
DIRS="1"
DIR1_NAME="/home"
DIR1_SUBDIRS="1"
DIR1_SUBDIR1_NAME="paul"
DIR1_SUBDIR1_ SUBDIRS="1"
DIR1_SUBDIR1_ SUBDIR1_NAME="apps"
DIR1_SUBDIR1_ SUBDIR1_SUBDIRS="1"
DIR1_SUBDIR1_ SUBDIR1_SUBDIR1_NAME="bin"
```

——————————Config Code v2——————————

```
DIRS="4"
DIR1_NAME="/home"
DIR2_NAME="/home/paul"
DIR2_NAME="/home/paul/apps"
DIR2_NAME="/home/paul/apps/bin"
```

In conclusion, the above described config file code is easy to read and manipulate, and can represent all information included in the XML code. The creation rules for the variable names are simple to be implemented with transformation languages like XSLT.

6.3 Implementation challenges

The development of the two shell script applications veCreator and veStarter is associated with writing portable shell code, especially, in the face of older Unix systems. Therefore, a shell is required which is mostly available on all systems, and offers a unified and powerful set of commands.

There are four traditional and widespread shells for Unix-type operating systems, the korn shell (ksh), bourne shell (sh), bourne again shell (bash), and the C shell (csh), and some newer developments like the perl shell.

The bourne shell is the only one, which is available on all systems apart from shell-less operating systems. Bourne shell code is portable to most systems, and is supported by the bash, which is the de facto standard shell on many Linux-type operating systems. The disadvantage of this shell is a smaller set of commands, which leads to an uncomfortable way of programming. Many of the familiar commands, which are offered by newer shells, are not available in the bourne shell.

For instance, it is not possible to declare associative arrays. Local functions are provided, but using them with parameters leads to an overwriting of the shell script parameters. Therefore, all shell script parameters have to be saved in an extra string before using a function. After finishing the function, all saved parameters have to be reset with the `set --` command. Tilde and brace extensions and also local variables in functions are not available. And finally, it is not possible to declare variables with the `declare` or `typeset` command.

```
globalArgs=$@
function this is a test
set -- $globalArgs
```

For other commands, a different syntax is required. The `test` command cannot be used in bracket form and has no operators, like "<", ">", "<=", or ">=".

6.4 veCreator

The veCreator reads the environment description and creates all file structure elements according to a given environment description. The veCreator application expects one

parameter from the user, the file of the environment description.

```
veCreator /home/marie/myEnv.conf
```

The path, where the virtual environment should be located, the environment location path, is determined in the environment description. The veCreator is able to process this path as the second parameter. Currently, this is needed for the veCreator program to start another instance of it, which processes bequeathed environment descriptions. The veCreator cannot execute a local function to do this, because local variables are not available. The second parameter overwrites the environment location path of this bequeathed environment descriptions.

First, some relative paths for mounting the file structure and Uniofs are initialized, as well some global default values, like the `umask`. Later, these settings will be read from configuration files and are not part of the program code. Then the program reads the environment description via the source command. At next, it checks, if the description should inherit other environment descriptions, if so, another instance of the veCreator is started, but as mentioned with two parameters.

After that, the program creates some temporary directories. In these directories the unions and the original file structure will be mounted. For performance reasons the current implementation of this prototype needs to mount some directories. This requires careful unmounting at the end of the application flow.
At next, the program starts to process and create all directories and files in the described order (Section 6.1.7). At last, the temporarily mounted directories will be deleted after unmounting them successfully.

6.5 veStarter

The veStarter is used to start applications in a virtual environment. The application expects two parameters. The first parameter is a path to the description file of the virtual environment. The second parameter is the application, which should be started.

At first, similar to the veCreator application, the veStarter initializes some relative paths for mounting the root directory and the unions, and sets some default values. These settings will be read from configuration files, and are not part of the program code. After that, the veStarter reads the environment description via the source command and checks, if the description should inherit other environment descriptions. For this case, the veStarter cannot start another veStarter instance, because this would be started as a child process with its own environment variables.

The child process cannot influence the environment variables of the parent process.

Therefore, the veStarter executes a local function `checkIncludes()`, which reads and creates recursively all bequeathed environment descriptions and its described environment variables. The first `checkIncludes()` function A reads a list of bequeathed environment descriptions. For this example it is assumed, that the list contains three environment descriptions. Then the `checkIncludes()` function A, running in a loop, starts a second `checkIncludes()` function B with the first description file of the list as a parameter.

The `checkIncludes()` function B reads its given environment description and checks itself, if its environment description inherits also another one. Provided that is not the case, the described environment variables are created and the function ends.

Then, the first `checkIncludes()` function A runs the second cycle of the loop and starts another `checkIncludes()` function to process the second environment description included in the list. But this list is empty, because it was overwritten with an empty list. `checkIncludes()` function B read via source command its environment description, which includes an empty list of bequeathed environment descriptions.

It is not possible to store the list in a local variable, because the bourne shell does not supportevd local variables in local functions. Therefore, the `checkIncludes()` function has to read its environment description before it runs every cycle of the loop to reinitialize the variable. But the loop counter variable was also overwritten, and cannot read from the environment description.

Therefore, it is required to use a string variable as an array to enable each local `checkIncludes()` function appends its current loop counter at the end of the string, before starting another instance.

However, the `checkIncludes()` function A leaves this loop and creates also environment variables according to its given description.

After completing the variable manipulations, the root directory and the unions are mounted in the virtual file structure. Then veStarter executes the given application and chrooted it in the virtual file structure. When the given application ends, the veStarter shell script continues with unmounting the unions and the root directory. At last, the veStarter ends and the shell environment is eliminated.

6.6 Security Aspects

The developed virtualisation approach creates a virtual environment, which can be used to port programs and support sufficient performance. But such a virtual environment needs to fulfill security policy, if it should be used on high performance computing systems. Security means as well the virtual environment should have a sandbox charac-

teristic. Programs running in it should not be able to get super user rights and cannot corrupt files out of their environment. Some important security aspects are discussed in the following to outline some possible solutions.

6.6.1 Password Security and Security Environment Descriptions

As described in Section 4.2.2, the veStarter application has super user rights and starts a given application A with the `chroot` command (`chroot /home/paul/env myApp`). This causes, that application A runs as well with super user rights and is able to destroy the entire system.

This can be avoided by using the `su` command. `su` is used to assume the login shell of another user without logging out. It is commonly used to change to super user permissions for administrative work without logging off and again on. In this case it is used in the opposite, to change from a super user to a normal user.

When `su` is started, it asks for the target user's password and grants the user access to that account. A super user, who wants to access a normal user's account is not asked for a password. However, `chroot` starts the `su` command instead of application A. As shown below, the `su` command is started with a given user. The given user is the one, who started the veStarter application before.

```
chroot /home/paul/env su paul
```

After that, user paul is logged in the shell, and is able to start its programs, but the veStarter was started to start application A. Therefore, application A is attached to the above command line.

```
chroot /home/paul/env su paul myApp
```

Hence, the veStarter can be started as shown below, if a user wants to use a shell in the virtual environment.

```
sudo veStarter env.conf  sh
```

This command line is part of the current implementation of the veStarter application.

This security procedure requires a virtual file structure, which includes all files required by the `su` program. For instance, the `su` program in `/bin` and the password file shadow in `etc` are required otherwise the veStarter fails. However, this security procedure can be eluded. A virtual file structure can contain a dummy `su` program, which is named `su`, but is written by the user to start a shell. The same can be done with dummy system

libraries used by super user programs.

Another security vulnerability is related to the location, where a virtual file structure is created. Image a virtual file structure with its shadow file is created on a mounted USB stick. The user can port this USB stick to another system and can read the encoded password. The password can be decoded via `brute force` or could be replace by another encoded password, which is known.

The user could hack the system with a decoded super user password, or he could remount the USB stick on the original system to start a shell in the virtual environment with the veStarter. In this "chrooted" shell the `su` command can be used to change to the root account. `su` asks for the super user's password and the user enters the replaced password, which will be accepted by the `su` program, because it uses the corrupted shadow file. `su` cannot use the original shadow file, because it is "chrooted" and did not recognize this.

This USB stick scenario may not be valid for supercomputers, where the file system is located on extra file servers, but the Harness Workbench is developed for a wide range of computing systems maintained with less security policies, and therefore it cannot offer these security vulnerabilities.

Therefore, a dummy shadow file is required, which contains another encoded super user password. All other user data are removed apart from the user who owns the virtual environment. Such a dummy shadow file needs to be copied by the veStarter program on every run to overwrite a user-corrupted shadow file. The dummy super user password needs to be random generated on every run of the veStarter.

As mentioned the user can read and decode this dummy password, and can use it to get super user rights in the virtual environment. Alternatively, to avoid generating a new password, the super user can be removed as well from the dummy shadow file, because it is not required for starting a program with normal user rights.

The virtualisation approach requires configuration possibilities for system administrators to force the veStarter program to execute a security environment description, which overwrites user-corrupted dummy file and programs in defined locations.

6.6.2 Sandbox Characteristic

As mentioned for software tests and security reasons, a virtual environment should protect the original operating system, the user's home directory and other virtual environments against the programs running inside them.

The current implementation enables a program inside the virtual environment to delete all files of its user. The program has the user rights, and can access all files from the

osroot directory, which contains the entire file system and the user's home directory.

It would be a very conditional solution to mount each main directory of the root directory except the home directory. The integration of files from the user's home directory could only be realized via the copy method, or mounting single directories. A two way communication via a file, which is mixed with sensitive files in a directory is not possible without changing the original file structure.

A more convenient solution is the use of a second user account (paul and paul2), which is used to start applications in a virtual environment with its rights. The program has the rights of paul2, and has no private access to the paul's home directory, like every other user.

The disadvantage of this approach is it requires changes of the permission, or changes of the ownership of a file, if it should be writeable in the virtual environment. A file permission conflict appears, if a file should be writable at the same time from inside and outside the virtual environment. Write rights for both users can be granted with a user group, of which both users are members. Then the veCreator has to change the group of the respective file. For more security the veCreator can create a new user group, which consists only of paul and paul2.

Each user needs an own second user account, otherwise a program from inside a virtual environment could access virtual environments of other users, see Figure 6.7.

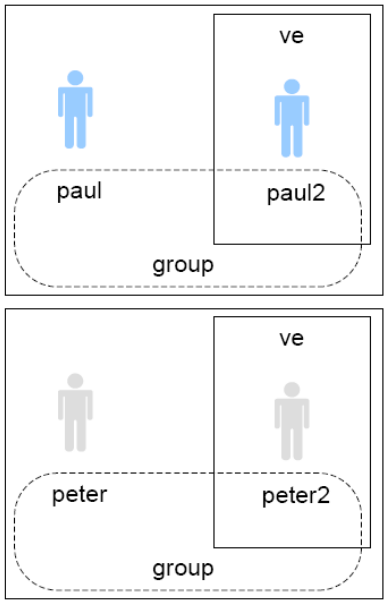

Figure 6.7: Separation via Additional Users

However, this approach is not able to isolate programs of a virtual environment from other virtual environments of the same user. Therefore, a user needs different additional user accounts, one for each virtual environment.

The creation and deletion of additional users is a critical aspect of this approach. It can interference the security policies of large-scaled computing system. The creation of virtual users instead of real system users is a more convenient way to create additional user for each virtual environment. The veStarter application can create a virtual user directly in the virtual environment.

As shown in below, a "chrooted" shell is started in the virtual environment instead the `su` program. Then a new user is created with `adduser`. At next, the permissions and ownerships of all required files are changed. After that the `su` command is executed for this new user and after it the application.

```
chroot /home/paul/env sh
adduser virtualPaul            ... [password] ... [uid] ...
...change required file permissions and ownerships to virtualPaul
su virtualPaul
./myApp
... rollback the file permissions and ownerships of all file from virtualPaul to paul ...
```

The new user does not exist on the operating system, but is able to create files and run its own programs. The password and user name are generated with appropriate approaches, or can be defined in the environment description. Furthermore, it is required to determine a non assigned userID, otherwise the virtual user could access the files of another existing user. If required, the userID determination needs to be synchronized with the operating system.

This approach supports a better sandbox characteristic and can be used without interferences for the operating system.

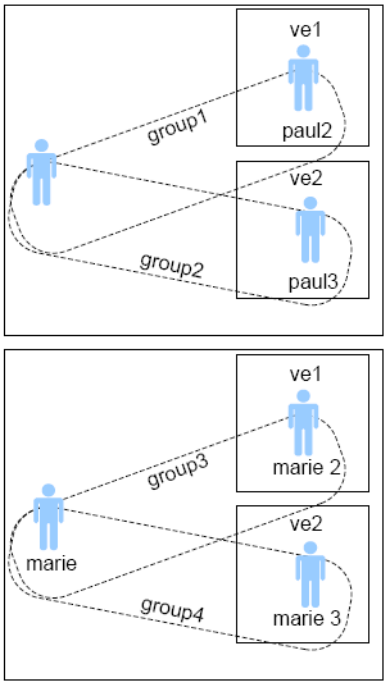

Figure 6.8: Separation via Virtual Users

7 Conclusion

7.1 Results

This dissertation project is a first step to develop a virtualisation approach for the Harness Workbench. The virtualisation approach helps to make software portable, and is itself portable on operating systems supporting the `chroot` mechanism. The approach is able to create and use virtualised environments according to environment descriptions. The creation in user space is possible, and helps to simplify the system administration of high performance computing systems.

The developed environment description concept is XML-based and provides the definition of environment variables, and file structures, and their file integration methods. Additionally description elements are evolved to specify image files and required software packages.

This description concept is platform-independent and can be integrated into other virtualisation projects, for instance Xen. The concept is constructed to be extensible for future requirements and is easy and flexible in use. System administrators can define inheritable environment descriptions to influence and simplify the description process for normal users.

The two prototype shell script application, veCreator and veStarter, were implemented as bourne compatible shell scripts, and are portable to all Unix-type operating systems offering the `chroot` mechanism and a bourne or bash shell.

Three file integration methods were developed and implemented. The combined use of the copy, link, and Unionfs method facilitates a seamless file integration, which fits to all use cases. Arbitrary file structures can be flexibly created, because files integrated via different integration methods can be merged with no constrains.

At last, different performance benchmarks were performed to proof and compare the usability of the file integration methods and to demonstrate the usability of the entire virtualisation approach. The veCreator application takes acceptable time to create a normal sized virtual file structure. The veStarter needs negligible time to manipulate the environment variables, and to prepare the virtual file structure. Furthermore, the runtime performance inside a virtual environment is slower but less than 1 %. Only, I/O-intensive workloads can cause a bigger performance lack but only if Unionfs is used.

However, it is necessary to implement the shell script applications veCreator and veStarter in C-code. C-code programs are much faster and can be portable too. Writing C-code is surprisingly easier than writing bourne-compatible shell code. Especially the nonexistence of local variables in functions and arrays leads to bothersome situations, and makes it impossible to checking functions for link-based directory integrations and more sophisticated inheritance declarations.

It will benefit application development and deployment by increasing the portability of software, especially for scientific computation on modern high performance computing systems.

As a final result of this dissertation project is achieved, the important system design tasks have been finished. An overall system design solution of the key problems of this dissertation has been created. A powerful description concept is developed and possible extensions and validation structures are outlined, and can be extended to describe virtualised system environments(VSE). The shell-script-restricted applications cannot realize all environment definitions, but enable a user to run programs in appropriate and self-described virtualised environment. Furthermore, detailed solutions are described and discussed to avoid security vulnerabilities without decreasing the functionality.

7.2 Future Work

In any event, according to the results of this dissertation project the described system design need to be implemented in C. A C library is also required to port the virtualisation approach to shell-less compute nodes.

The entire inheritance process needs to be changed from the currently used overwrite mechanism. The integration of directories via link should be carefully implemented to accelerate the file structure creation.

Furthermore, the sandbox characteristic of the approach and the security in general should be increased. Therefore, the described solutions can be implemented and tested. Especially, an appropriate method is required to avoid userID and groupID conflicts between virtual and real users.

In addition, the description concept can be sophisticated to allow a detailed description of network issues, like network identity, communication streams and required server addresses. Moreover, the concept should provide the configuration and installation of services. Package management is another issue, which should be provided by a complete virtualisation approach. Last but not least, all these need to be implemented in a portable way, without the requirement to manipulate the target system, for instance by installing new kernel modules.

Bibliography

[1] G. A. Geist, Harness - the power of network, http://www.csm.ornl.gov/harness, [Online; accessed 11-March-2007]

[2] TOP500.Org, TOP500 Statistics, http://www.top500.org/stats, [Online; accessed 11-March-2007]

[3] Wikimedia Foundation, Inc., Wikipedia.Cygwin, the free encyclopedia, http://en.wikipedia.org/wiki/Cygwin, [Online; accessed 11-March-2007]

[4] Wikimedia Foundation, Inc., XML - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/XML, [Online; accessed 11-March-2007]

[5] Wikimedia Foundation, Inc., Expat (XML) - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Expat_%28XML%29, [Online; accessed 11-March-2007]

[6] Refsnes Data, XSLT Tutorial, http://www.w3schools.com/xsl/default.asp, [Online; accessed 11-March-2007]

[7] Andrew Morton, A Stackable Unification File System, http://www.am-utils.org/project-unionfs.html, [Online; accessed 11-March-2007]

[8] Wikimedia Foundation, Inc., Beowulf (computing) - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Beowulf_(computing), [Online; accessed 11-March-2007]

[9] Christian Engelmann and Hong Ong and Stephen L. Scott, Middelware in Modern High Performance Computing System Arcitectures, Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS), Beijing, China 2007

[10] José Moreira and Michael Brutman and José Castaños and Thomas Engelsiepen and Mark Giampapa and Tom Gooding and Roger Haskin and Todd Inglett and Derek Lieber and Pat McCarthy and Mike Mundy and Jeff Parker and Brian Wallenfelt, Blue Gene system software — Designing a highly-scalable operating system: the Blue

Gene/L story, ACM Press, New York, NY, USA, SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florid, 2006

[11] Wikimedia Foundation, Inc., BRuntime-Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Runtime, [Online; accessed 11-March-2007]

[12] Wikimedia Foundation, Inc., Message Passing Interface - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Message_Passing_Interface, [Online; accessed 11-March-2007]

[13] Wikimedia Foundation, Inc., Emulator - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Emulator, [Online; accessed 11-March-2007]

[14] Wikimedia Foundation, Inc., Simulation - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Simulation, [Online; accessed 11-March-2007]

[15] Wikimedia Foundation, Inc., Virtualization - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Virtualization, [Online; accessed 11-March-2007]

[16] Wikimedia Foundation, Inc., Microsoft Virtual PC - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Microsoft_Virtual_PC, [Online; accessed 11-March-2007]

[17] Fabrice Bellard, QEMU Open Source Processor Emulator, http://www.qemu.org/, [Online; accessed 11-March-2007]

[18] Wikimedia Foundation, Inc., Native Virtualization - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Native_virtualization, [Online; accessed 11-March-2007]

[19] Wikimedia Foundation, Inc., Full Virtualization - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Full_virtualization, [Online; accessed 11-March-2007]

[20] Wikimedia Foundation, Inc., Virtual Iron - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Virtual_Iron, [Online; accessed 11-March-2007]

[21] Wikimedia Foundation, Inc., VMware - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/VMware, [Online; accessed 11-March-2007]

[22] GNU Free Documentation, FAQ-MOL, http://mac-on-linux.sourceforge.net/wiki/index.php/FAQ, [Online; accessed 11-March-2007]

[23] Wikimedia Foundation, Inc., Win4Lin - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Win4Lin, [Online; accessed 11-March-2007]

[24] Wikimedia Foundation, Inc., Virtual machine - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Virtual_machine, [Online; accessed 11-March-2007]

[25] Wikimedia Foundation, Inc., Partial virtualization - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Partial_virtualization, [Online; accessed 11-March-2007]

[26] Wikimedia Foundation, Inc., Paravirtualization - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Paravirtualization, [Online; accessed 11-March-2007]

[27] XenSource, Inc., XenFaq, http://wiki.xensource.com/xenwiki/XenFaq, [Online; accessed 11-March-2007]

[28] Wikimedia Foundation, Inc., Parallels Workstation - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Parallels_Workstation, [Online; accessed 11-March-2007]

[29] Enomaly Inc., Enomalism: XEN Virtualized Server Management Console, http://www.enomalism.com/features/, [Online; accessed 11-March-2007]

[30] Wikimedia Foundation, Inc., Operating system-level virtualization-Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Operating_system-level_virtualization, [Online; accessed 11-March-2007]

[31] GNU Free Documentation, Overview-Linux VServer, http://linux-vserver.org/Overview#The_Linux-VServer_approach, [Online; accessed 11-March-2007]

[32] Wikimedia Foundation, Inc., Virtuozzo - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Virtuozzo, [Online; accessed 11-March-2007]

[33] Wikimedia Foundation, Inc., Application Virtualization - Wikipedia, the free encyclopedia,

http://en.wikipedia.org/wiki/Application_Virtualization, [Online; accessed 11-March-2007]

[34] Wikimedia Foundation, Inc., FreeBSD jail - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/FreeBSD_Jail, [Online; accessed 11-March-2007]

[35] Wikimedia Foundation, Inc., Java Virtual Machine - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Java_Virtual_Machine, [Online; accessed 11-March-2007]

[36] Thinstall, Thinstall 3.0 Virtualization Suite, http://www.thinstall.com/products/virtualization_suite.php, [Online; accessed 11-March-2007]

[37] Wikimedia Foundation, Inc., Altiris - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Altiris, [Online; accessed 11-March-2007]

[38] Trigence Corp., Trigence AE: Product Details, http://www.trigence.com/products/productdetails.html, [Online; accessed 11-March-2007]

[39] Wikimedia Foundation, Inc., Unix shell - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Unix_shell, [Online; accessed 11-March-2007]

[40] National Center for Computational Sciences, Jaguar, http://info.nccs.gov/resources/jaguar, [Online; accessed 11-March-2007]

[41] Sarah Anderson Cray, Inc., Cray XT3 Programming Introduction, www.psc.edu/training/XT3_Aug05/lectures/Intro.ppt, [Online; accessed 11-March-2007]

[42] Simon Sheppard, Inc., chroot MAN page, http://www.ss64.com/bash/chroot.html, [Online; accessed 11-March-2007]

[43] Wikimedia Foundation, Inc., Institute of Electrical and Electronics Engineers - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/IEEE, [Online; accessed 11-March-2007]

[44] Wikimedia Foundation, Inc., POSIX - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/POSIX, [Online; accessed 11-March-2007]

[45] The IEEE and The Open Group, The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition, http://www.opengroup.org/onlinepubs/009695399/, [Online; accessed 11-March-2007]

[46] Wikimedia Foundation, Inc., GNU build system - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/GNU_build_system, [Online; accessed 11-March-2007]

[47] Open Cluster Group, OSCAR Open Source Cluster Application Group, http://oscar.openclustergroup.org/, [Online; accessed 11-March-2007]

[48] Innovative Computing Laboratory (ICL), NetBuild, http://icl.cs.utk.edu/netbuild/, [Online; accessed 12-March-2007]

[49] Wikimedia Foundation, Inc., IGNU build system - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/GNU_build_system, [Online; accessed 11-March-2007]

[50] Free Software Foundation, Inc., Inc., Existing Tests, http://www.gnu.org/software/autoconf/manual/html_node/Existing-Tests.html, [Online; accessed 11-March-2007]

[51] Free Software Foundation, Inc., Portability of C Functions, http://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.61/html_node/Function-Portability.html, [Online; accessed 11-March-2007]

[52] Free Software Foundation, Inc., Autoconf, http://www.gnu.org/software/autoconf/, [Online; accessed 11-March-2007]

[53] Free Software Foundation, Inc., GNU Automake - The GNU Project, http://www.gnu.org/software/automake/, [Online; accessed 11-March-2007]

[54] Free Software Foundation, Inc., GNU Libtool - The GNU Portable Library Tool, http://www.gnu.org/software/libtool/, [Online; accessed 11-March-2007]

[55] Murray Cumming, Using Automake and Autoconf with C++, http://www.openismus.com/documents/linux/automake/automake.shtml#automakeandautoconf/, [Online; accessed 11-March-2007]

[56] Oak Ridge National Laboratory, CSM, PVM - Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/pvm_home.html, [Online; accessed 11-March-2007]

[57] Wikimedia Foundation, Inc., LAM/MPI - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/LAM/MPI, [Online; accessed 11-March-2007]

[58] Paul Barham and Boris Dragovic and Keir Fraser and Steven Hand and Tim Harris and Alex Ho and Rolf Neugebauer and Ian Pratt and Andrew Warfield, Xen and the Art of Virtualization, University of Cambridge Computer Laboratory, Cambridge, UK, 2003, http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf, [Online; accessed 11-March-2007]

[59] Wikimedia Foundation, Inc., Institute of Electrical and Electronics Engineers - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/IEEE, [Online; accessed 11-March-2007]

[60] The FreeBSD Project, About FreeBSD, http://www.freebsd.org/about.html, [Online; accessed 11-March-2007]

[61] The NetBSD Foundation, Inc. , NetBSD Documentation, http://www.netbsd.org/Documentation/, [Online; accessed 11-March-2007]

[62] Wikimedia Foundation, Inc., Plan 9 from Bell Labs - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Plan_9_from_Bell_Labs, [Online; accessed 11-March-2007]

[63] Stephen L. Scott and Hong Ong and Geoffroy Vallé and Thomas Naughton, Configurable Virtualized System Environments for High Performance Computing, Proceedings of 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt), Lisbon, Portugal 2007

[64] David Quigley and Josef Sipek and Charles P. Wright and Erez Zadok, Unionfs: User - and Community-Oriented Development of a Unification File System, Stony Brook University, Ottawa Linux Symposium (OLS 2006), 2006, http://www.fsl.cs.sunysb.edu/docs/sipek-ols2006/index.html, [Online; accessed 12-March-2007]

[65] FiST: Stackable File System Language and Templates, http://www.filesystems.org/, [Online; accessed 12-March-2007]

[66] Neil Brown, The Linux Virtual File - system Layer, http://www.cse.unsw.edu.au/ñeilb/oss/linux-commentary/vfs.html, [Online; accessed 12-March-2007]

[67] The IEEE and The Open Group, mmap - map pages of memory, http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html, [Online; accessed 12-March-2007]

[68] Wikimedia Foundation, Inc., Knoppix - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/KNOPPIX, [Online; accessed 12-March-2007]

[69] Suzanne M. Kelly and Ron Brightwell and John Van Dyke, Catamount Software Architecture with Dual Core Extensions, http://www.cs.sandia.gov/ smkelly/SAND2006-2561C-CUG2006-CatamountDualCore.pdf, [Online; accessed 12-March-2007]

[70] Jerry Peek and Tim O'Reilly and Mike Loukides, UNIX POWERFUL TOOLS, http://www.unix.org.ua/orelly/unix/upt/ch06_05.htm, [Online; accessed 12-March-2007]

[71] Capps, IOzone Filesystem Benchmark, http://www.iozone.org/, [Online; accessed 12-March-2007]

[72] Jörg Hoh, Inc., Postmark - Benchmark, http://www.devone.org/linux/postmark.html, [Online; accessed 12-March-2007]

[73] SS64.com, . (source or dot operator), http://www.ss64.com/bash/period.html, [Online; accessed 12-March-2007

A Appendix

B program code

B.1 veCreater.sh

```
#!/bin/sh
#usage veCreator DESCRIPTION_FILE APPLICATION

#defaults

stdUmask="755"
stdDirUmask="755"
stdVia="link"

#ve-relative paths
harnessDir="/.harness"

osMountPoint=$harnessDir"/osroot"
osMountPointCow=$harnessDir"/unionfsCow"
osMountPointRo=$harnessDir"/unionfsRo"
writeDirSuffix="_write"

#set processID

if test -n "$3";then
processID=$3
else    processID=0
fi

#read description file
. $1

#absolute paths to VE location

if test -n "$2" ;then
fakeDir=$2
else  fakeDir=$ENV
fi


root="$fakeDir$osMountPoint" # absolute path to the mounted root
rootCow="$fakeDir$osMountPointCow" # absolute path to the mounted root
rootRo="$fakeDir$osMountPointRo" # absolute path to the mounted root

prepareVE()
{

#create fakedir and
```

```
mkdir -p $fakeDir

#mounting the real root directory in the virtual root directory
mkdir -p $root
sudo mount --bind / "$root"

mkdir -p "$fakeDir$osMountPointCow"
mkdir -p "$fakeDir$osMountPointRo"

#temporary mounting of the osMountPoint in rooot
sudo mkdir -p "$osMountPoint"
sudo mount --bind / "$osMountPoint"

}
#copy all files required for sudo and su
makeSecurity()
{

mkdir -p "$fakeDir/etc"
rm "$fakeDir/etc/sudoers"
cp /etc/sudoers "$fakeDir/etc/sudoers"
cp -fr /var/run "$fakeDir/var/run"
#mkdir -p "$fakeDir/var/run"

mkdir -p "$fakeDir/usr/bin"
rm "$fakeDir/usr/bin/sudo"
cp /usr/bin/sudo "$fakeDir/usr/bin/sudo"

#create su
mkdir -p "$fakeDir/bin"
rm "$fakeDir/bin/su"
cp /bin/su "$fakeDir/bin/su"
rm "$fakeDir/etc/pam.d"
cp /etc/pam.d "$fakeDir/etc/pam.d"

#create passwd
rm "$fakeDir/etc/shadow"
cp /etc/shadow "$fakeDir/etc/shadow"
rm "$fakeDir/etc/passwd"
cp /etc/passwd "$fakeDir/etc/passwd"

}

#tries to ensure unmounting
myExit()
{ sudoCount=0
echo "exit"
while test  $? -eq "0" ;do
sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$root"
done
if test $sudoCount -eq 0 -a ‘ls -al "$root" | wc -l‘ -gt 2;then

echo "WARNING: directory $root is not empty or still mounted!!"
```

```
fi

sudoCount=0
echo "  "
while test  $? -eq "0" ;do
echo "sudo2"
sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$osMountPoint"

done
fc=‘ls -al "$osMountPoint" | wc -l‘

if test $sudoCount -gt 0 -a  $fc -le 2;then
echo "rmdir "$osMountPoint""
else
echo "WARNING: temporary directory $osMointPoint is not empty!! Please unmounted it before deleting!!"
fi
## unionfsCOW

sudoCount=0
echo "  "
while test  $? -eq "0" ;do
sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$rootCow"
done
if test $sudoCount -eq 0 -a ‘ls -al "$rootCow" | wc -l‘ -gt 2;then

echo "WARNING: directory $rootCow is not empty or still mounted!!"
fi

sudoCount=0
echo "  "
while test  $? -eq "0" ;do

sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$osMountPointCow"

done
fc=‘ls -al "$osMountPointCow" | wc -l‘

if test $sudoCount -gt 0 -a  $fc -le 2;then
echo "rmdir "$osMountPointCow""
else
echo "WARNING: temporary directory $osMointPointCow is not empty!! Please unmounted it before deleting!!"
fi
##unionfsRo

sudoCount=0
echo "  "
while test  $? -eq "0" ;do
sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$rootRo"
done
if test $sudoCount -eq 0 -a ‘ls -al "$rootRo" | wc -l‘ -gt 2;then
```

```
echo "WARNING: directory $rootRo is not empty or still mounted!!"
fi

sudoCount=0
echo "  "
while test  $? -eq "0" ;do

sudoCount=‘expr $sudoCount + 1‘
sudo umount  "$osMountPointRo"

done
fc=‘ls -al "$osMountPointRo" | wc -l‘
if test $sudoCount -gt 0 -a  $fc -le 2;then
echo "rmdir "$osMountPointRo""
else
echo "WARNING: temporary directory $osMointPointRo is not empty!! Please unmounted it before deleting!!"
fi

}
#creates all directories required for unionfs
prepareunionfs()
{


echo "unionfsprepare:$1 $2 "
sudo mkdir -p "$osMountPointCow"
sudo mkdir -p "$osMountPointRo"
temp=‘pwd‘

cd /

x=‘ls -al | wc -l‘
lstext=‘ls -abm‘



while test $x -ge 1 ;do

name=‘echo $lstext | awk 'BEGIN{FS=", "}{print $'$x'}' | sed 's/\([^ \]\)/\\\\\1/g'‘

if test -n "$name";then

if eval test -d "/$name" -a "$name" != ".." -a "$name" != "." -a "/$name" != "$harnessDir"
         -a "/$name" != "/home"; then
eval mkdir -p "$fakeDir/$osMountPointCow/$name"
eval mkdir -p "$fakeDir/$osMountPointCow/$name$writeDirSuffix"
eval mkdir -p "$fakeDir/$osMountPointRo/$name"

#eval sudo  mkdir -p "$osMountPointCow/$name"
#eval sudo mkdir -p "$osMountPointCow/$name$writeDirSuffix"

#eval sudo mkdir -p "$osMountPointRo/$name"
#eval echo "$fakeDir/$osMountPointCow/$name"
```

```
#else echo nein:$name:$harnessDir
fi

fi
x=‘expr $x - 1‘
#
done

sudo mount --bind / "$osMountPointCow"
sudo mount --bind / "$osMountPointRo"
cd "$temp"

}


#usage: createFile varname stdUmask stdVia target source
#creates files defined in branch elements
#suffix:CBF
#this variable suffix ensures unique variable names--> nothing is local
createBranchFile()
{
echo "create file $1 $2 $3 $4 $5"

targetCBF="$4"
#targetCBF=‘echo $targetCBF | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
sourceCBF="$5"
#sourceCBF=‘echo $sourceCBF | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
eval fileCBF=’$’"$1"
fileCBF=‘echo "$fileCBF"| awk ’BEGIN{ FS=":"} {print $3}’ | sed  -e
                                ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
eval echo "fileCBF:"$fileCBF"’M’"
eval fileNNameCBF=’$’$1’_NNAME’
fileNNameCBF=‘echo "$fileNNameCBF" | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
echo $fileNNameCBF
eval fileViaCBF=’$’$1’_VIA’
#echo $fileViaCBF
eval filePermCBF=’$’$1’_PERMISSION’
eval fileOwnCBF=’$’$1’_OWNER’
eval fileGrpCBF=’$’$1’_GROUP’
#echo $filePermCBF

if test -z $filePermCBF; then
filePermCBF=$2
fi

if test -z $fileViaCBF; then
fileViaCBF=$3
fi
if test -z "$fileNNameCBF"; then
fileNNameCBF=$fileCBF
#eval echo "fileNNameCBF:$fileNNameCBF"
fi
if test -z "$fileCBF" ; then
echo "$1 is wrong"
```

```
myExit
fi

# if test -z $fileName; then
# fileName=‘basename $fileSource‘
# fi
#create the file
case $fileViaCBF in

"delete")
eval rm -f "$fakeDir$targetCBF/$fileNNameCBF"

;;
"link") eval cp -sf "$osMountPoint$sourceCBF/$fileCBF" "$fakeDir$targetCBF/$fileNNameCBF"
#./symlink $targetCBF’/’$fileNNameCBF $fileCBF $fakeDir $osMountPoint
;;
"copy")
eval cp -f "$sourceCBF/$fileCBF" "$fakeDir$targetCBF/$fileNNameCBF"
if test !  $? -eq "0" ;then

eval rm -f "$fakeDir$targetCBF/$fileNNameCBF"
eval cp -f "$sourceCBF/$fileCBF $fakeDir$targetCBF/$fileNNameCBF"

fi
eval chmod "$filePermCBF" "$fakeDir$targetCBF/$fileNNameCBF"
eval chown "$fileOwnCBF" "$fakeDir$targetCBF/$fileNNameCBF"
eval chgrp "$fileGrpCBF" "$fakeDir$targetCBF/$fileNNameCBF"
;;
"unionfsCow")
eval cp -sf "$osMountPointCow$sourceCBF/$fileCBF" "$fakeDir$targetCBF/$fileNNameCBF"
;;
"unionfsRo")
eval cp -sf "$osMountPointRo$sourceCBF/$fileCBF" "$fakeDir$targetCBF/$fileNNameCBF"
;;
esac



}

#usage: createDirFile varname stdUmask stdVia target
#creates files defined in directory elements
#suffix:CDF
createDirFile()
{
targetCDF="$4"
echo "createDirfile $1 $2 $3 $4 $5"
eval fileNNameCDF=’$’$1’_NNAME’
fileNNameCDF=‘echo "$fileNNameCDF" | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
echo "fileCDF:$fileNNameCDF"
eval fileSourceCDF=’$’$1
fileSourceCDF=‘echo "$fileSourceCDF" | awk ’BEGIN{ FS=":"}
    {print $3}’| sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
echo $fileSourceCDF
```

```
eval fileViaCDF='$'$1'_VIA'
echo $fileViaCDF
eval filePermCDF='$'$1'_PERMISSION'
echo $filePermCDF
eval fileOwnCDF='$'$1'_OWNER'
eval fileGrpCDF='$'$1'_GROUP'

if test -z $filePermCDF; then
filePermCDF=$2
fi

if test -z $fileViaCDF; then
fileViaCDF=$3
fi

if test -z "$fileSourceCDF";  then
echo "$1 is wrong"
myExit "missing source"
fi

if test -z "$fileNNameCDF"; then
fileNNameCDF=‘basename "$fileSourceCDF"‘
fi

#create the file
case $fileViaCDF in

"delete") eval rm -f "$fakeDir$targetCDF/$fileNNameCDF"
;;
"link") eval cp -sf "$osMountPoint$fileSourceCDF" "$fakeDir$targetCDF/$fileNNameCDF"

;;
"copy") eval cp -f "$fileSourceCDF" "$fakeDir$targetCDF/$fileNNameCDF"
if test !  $? -eq "0" ;then

eval rm -f "$fakeDir$targetCDF/$fileNNameCDF"
eval cp -f "$fileSourceCDF" "$fakeDir$targetCDF/$fileNNameCDF"

fi
eval chmod "$filePermCDF" "$fakeDir$targetCDF/$fileNNameCDF"
eval chown "$fileOwnCDF" "$fakeDir$targetCDF/$fileNNameCDF"
eval chgrp "$fileGrpCDF" "$fakeDir$targetCDF/$fileNNameCDF"
;;
"unionfsCow")
eval cp -sf "$osMountPointCow$fileSourceCDF" "$fakeDir$targetCDF/$fileNNameCDF"
;;
"unionfsRo")
eval cp -sf "$osMountPointRo$fileSourceCDF" "$fakeDir$targetCDF/$fileNNameCDF"
;;
esac

}
#usage: createStandAloneFile varname stdUmask stdVia
#suffix:CSAF
```

```
createStandAloneFile()
{
echo "createStandAlonefile $1 $2 $3"
eval fileTargetCSAF='$'$1'_TARGET'
fileTargetCSAF=`echo "$fileTargetCSAF" | sed  -e 's/^-/.\/-/g' -e 's/\(.\)/\\\\\1/g'`
echo "fileCSAF:$fileTargetCSAF"
eval fileSourceCSAF='$'$1
fileSourceCSAF=`echo "$fileSourceCSAF" | awk 'BEGIN{ FS=":"} {print $3}'|sed
            -e 's/^-/.\/-/g' -e 's/\(.\)/\\\\\1/g'`
echo $fileSourceCSAF
eval fileViaCSAF='$'$1'_VIA'
echo $fileViaCSAF
eval filePermCSAF='$'$1'_PERMISSION'
echo $filePermCSAF
eval fileOwnCSAF='$'$1'_OWNER'
eval fileGrpCSAF='$'$1'_GROUP'

if test -z $filePermCSAF; then
filePermCSAF=$2
fi

if test -z $fileViaCSAF; then
fileViaCSAF=$3
fi

if test -z "$fileSourceCSAF";  then
echo "$1 is empty"
myExit
fi

if test -z "$fileTargetCSAF"; then
fileTargetCSAF="$fileSourceCSAF"
fi

case $fileViaCSAF in

"delete") eval rm -f "$fakeDir$fileTargetCSAF"
;;
"link") mkdir -p "`eval dirname "$fakeDir$fileTargetCSAF"`"
eval cp -sf "$osMountPoint$fileSourceCSAF" "$fakeDir$fileTargetCSAF"

;;
"copy") mkdir -p "`eval dirname "$fakeDir$fileTargetCSAF"`"
eval cp -f "$fileSourceCSAF" "$fakeDir$fileTargetCSAF"
if test !  $? -eq "0" ;then

eval rm -f "$fakeDir$fileTargetCSAF"
eval cp -f "$fileSourceCSAF" "$fakeDir$fileTargetCSAF"

fi
eval chmod "$filePermCSAF" "$fakeDir$fileTargetCSAF"
eval chown "$fileOwnCSAF" "$fakeDir$fileTargetCSAF"
eval chgrp "$fileGrpCSAF" "$fakeDir$fileTargetCSAF"
;;
```

```
"unionfsCow")
mkdir -p "‘eval dirname "$fakeDir$fileTargetCSAF"‘"
eval cp -sf "$osMountPointCow$fileSourceCSAF" "$fakeDir$fileTargetCSAF"
;;
"unionfsRo")
mkdir -p "‘eval dirname "$fakeDir$fileTargetCSAF"‘"
eval cp -sf "$osMountPointRo$fileSourceCSAF" "$fakeDir$fileTargetCSAF"
;;
esac

}
#link all files of a directory
#usage linkDir $fakeDir $osMountPoint $sourceSource $dirTarget
# not in use
linkDir()
{
echo "linkdir:"$1 $2 $3 $4

#cd $1$4
count=‘ls -al $3| wc -l‘

x=1

while test "$x" -le $count ;do

name=‘ls -a $3 | awk ’ NR==’$x’ {print $0}’‘
if test ! -d "$3"’/’"$name" ; then
ln -f -s $2$3’/’"$name" $1$4’/’"$name"
#else echo "directory:$name"
fi
x=‘expr $x + 1‘

done

}
# usage: integrateBranch curDir branchnumber
# processes a given branch element
integrateBranch()
{
curBranch=$1’_BRANCH’$2

echo "integrate $curBranch"
eval branchTarget=’$’"$1"
branchTarget=‘echo "$branchTarget" | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
eval branchSource=’$’$curBranch
branchSource=‘echo "$branchSource" | awk ’BEGIN{ FS=":"} {print $3}’ | sed  -e ’s/^-/.\/-/g’
          -e ’s/\(.\)/\\\\\1/g’‘
echo "branchsource:$branchSource"

eval branchVia=’$’$curBranch’_VIA’

eval branchUmask=’$’$curBranch’_UMASK’

eval branchDirUmask=’$’$curBranch’_DIRUMASK’
```

```
eval branchGetAll='$'$curBranch'_GETALL'

#eval branchGetAllR='$'$curBranch'_GETALLRECURSIVE'


if test -z "$branchUmask"; then
branchUmask=$umask
fi
if test -z "$branchDirUmask"; then
branchDirUmask=$dirUmask
fi
if test -z "$branchVia"; then
branchVia=$dirVia
fi


if test -z "$branchSource"; then
echo "Error: $curBranch has no source!"
exit
fi
#process getAll="files"
if test -n "$branchGetAll" -a "$branchGetAll" = "files";then
echo "1"
case $branchVia in

"link") eval cp -fs "$osMountPoint$branchSource/"* "$fakeDir$branchTarget"
eval cp -fs "$osMountPoint$branchSource/".* "$fakeDir$branchTarget"
;;
"copy") echo "jetzt copy"
#files=`ls -ab | sed 's/^*$//g'| sed 's/^-/.\/-/g'`
#     / " ' * ; - & ? ( ) [ ] ~ ! $ { } > < # @
#files=`ls -ab | sed 's/\([]/"'"';"'*-&?()[~!${}<>#@]\)[^ ]/\\\\\1/g'`

temp=`pwd`
eval cd "$branchSource"
#files=`eval ls -a | sed -e 's/^-/.\/-/g'
      -e's/\(.\)/\\\\\1/g'`
# files=`ls -abm | sed -e 's/[,][ ]-/, .\/-/g' -e 's/\([^ \]\)/\\\\\1/g'
     -e 's/\\\, / . /g'`
#files=`ls -abm | sed -e 's/[,]$/, /g' -e 's/^[-]/ -/g' -e 's/[ ]-/ .\/-/g'
     -e 's/\([^ \]\)/\\\\\1/g' -e 's/\\\, / . /g'`
emptyspace=" "
files=`ls -abm | sed -e 's/[,]$/, /g' -e 's/^[-]/ -/g'
  -e 's/[ ]-/ .\/-/g' -e 's/\([^ \]\)/\\\\\1/g' -e 's/\\\, / $emptyspace /g'`

#echo  "files:$files"
#eval echo $files
#pwd
eval cd "$fakeDir$branchTarget"
#pwd
eval rm $files 2>/dev/null

eval cp -Lf "$branchSource/"* "$fakeDir$branchTarget" 2>/dev/null
```

```
eval cp -f "$branchSource/".* "$fakeDir$branchTarget" 2>/dev/null
cd "$temp"
;;

"unionfsCow")
eval cp -sf "$osMountPointCow$branchSource/"* "$fakeDir$branchTarget"
eval cp -sf "$osMountPointCow$branchSource/".* "$fakeDir$branchTarget"
;;
"unionfsRo")
eval cp -sf "$osMountPointRo$branchSource/"* "$fakeDir$branchTarget"
eval cp -sf "$osMountPointRo$branchSource/".* "$fakeDir$branchTarget"
;;
esac
fi

#process getAll="recursive"
if test -n "$branchGetAll" -a "$branchGetAll" = "recursive";then
echo "3"
case $branchVia in

"link") eval cp -fsr "$osMountPoint$branchSource/". "$fakeDir$branchTarget" 2>/dev/null
#cp -fsr "$osMountPoint$branchSource/".* "$fakeDir$branchTarget"
;;
"copy")
 ./recSynDelete "$branchSource" "$fakeDir$branchTarget"
eval cp -frL "$branchSource/". "$fakeDir$branchTarget" # 2>/dev/null
eval cp -fr "$branchSource/". "$fakeDir$branchTarget" #2>/dev/null
;;

"unionfsCow")
eval cp -fsr "$osMountPointCow$branchSource/". "$fakeDir$branchTarget" 2>/dev/null
;;
"unionfsRo")
eval cp -fsr "$osMountPointRo$branchSource/". "$fakeDir$branchTarget" 2>/dev/null
;;
esac
fi
#end: getAll


#create all single files of the branch
eval fileCountIB='$'$curBranch'_FILES'
echo $fileCountIB
if test -n "$fileCountIB"; then
fileIBX=1

while test `expr $fileIBX` -le `expr $fileCountIB` ;do
# echo "bin drin"

curFileIB=$curBranch'_FILE'$fileIBX
curFileIB=`echo "$curFileIB" | awk 'BEGIN{ FS=":"} {print $3}'|
  sed  -e 's/^-/.\/-/g' -e 's/\(.\)/\\\\\1/g'`
echo "curFileIB:$curFileIB"
# #creates the file
```

```
createBranchFile "$curFileIB" "$branchUmask" "$branchVia" "$branchTarget" "$branchSource"
fileIBX=‘expr $fileIBX + 1‘
done
fi


}

##void main() start point of functional part

#check for include instructions
includeX=0
eval includeFile=’$include’$includeX

while test -n "$includeFile" ; do

childID=‘expr $processID + 1‘

$0 $includeFile $fakeDir $childID
includeX=‘expr $includeX + 1‘

eval includeFile=’$include’$includeX
echo "drin fakedir:$fakeDir file:$includeFile X:$includeX"
done

if test $includeX -eq 0 ; then
prepareVE
prepareunionfs
fi

dirCount=$DIRS

dirX=1
#create the directory
while test $dirX -le $dirCount ;do
echo "bin in schleife"
curDir=’DIR’$dirX


eval dirName=’$’$curDir #’_DIRNAME’
dirName=‘echo "$dirName" | sed  -e ’s/^-/.\/-/g’ -e ’s/\(.\)/\\\\\1/g’‘
eval umask=’$’$curDir’_UMASK’
echo $umask
eval dirUmask=’$’$curDir’_DIRUMASK’
echo $dirUmask
eval dirVia=’$’$curDir’_VIA’
echo $dirVia
eval dirOwn=’$’$curDir’_PERMISSION’
eval dirOwn=’$’$curDir’_OWNER’
eval dirGrp=’$’$curDir’_GROUP’

if test -z "$dirName"; then

#echo "$dirName $dirX $dirCount"
```

```
dirX=‘expr $dirX + 1‘
continue
fi

if test -z "$umask"; then
umask=$stdUmask
fi

if test -z  "$dirUmask"; then
dirUmask=$stdDirUmask
fi


if test -z "$dirVia"; then
dirVia=$stdVia
fi
if test -z "$dirPermission"; then
$dirPermission=$stdDirUmask
fi
eval mkdir -p "$fakeDir$dirName"
chmod "$dirPerm" "$fakeDir$dirName"
chown "$dirOwn" "$fakeDir$dirName"
chgrp "$dirGrp" "$fakeDir$dirName"
#integrate branch
eval branchCount=’$’$curDir’_BRANCHES’

if test -n "$branchCount"; then
branchX=1

while test ‘expr $branchX‘ -le ‘expr $branchCount‘ ;do
echo "bin drin"
#integrate creates the file
integrateBranch $curDir $branchX

branchX=‘expr $branchX + 1‘
done
fi



#create all single listed files in the directory
eval fileCount=’$’$curDir’_FILES’

if test -n "$fileCount"; then
fileX=1

while test ‘expr $fileX‘ -le ‘expr $fileCount‘ ;do
curFile=$curDir’_FILE’$fileX
echo "curFile:$curFile"
#creates the file
createDirFile "$curFile" "$umask" "$dirVia" "$dirName"
fileX=‘expr $fileX + 1‘
done
fi
```

```
#create all subdirectories
eval subDirCount='$'$curDir'_SUBDIRS'
echo "subdir creation"
if test -n "$subDirCount"; then
subX=1
echo "in subdircreation"
while test `expr $subX` -le `expr $subDirCount` ;do
curSubDir=$curDir'_SUBDIR'$subX
echo "curSubDir:$curSubDir"
#creates the file pathToDescriptionFile currentDir parentDirPath stdUmask stdDirUmask stdVia
./createSubDir $1 $curSubDir $dirName $umask $dirUmask $dirVia
subX=`expr $subX + 1`
done
fi

dirX=`expr $dirX + 1`
done

#create standalone files
fileCount=$FILES
echo "standalone beginn:$fileCount"
if test -n "$fileCount"; then
fileX=1

while test `expr $fileX` -le `expr $fileCount` ;do
curFile='FILE'$fileX
echo "curFile:$curFile"
#creates the file
createStandAloneFile "$curFile" "$umask" "$dirVia"
fileX=`expr $fileX + 1`
done

fi

if test $processID -eq 0 ;then

#makeSecurity
myExit normal

fi
```

B.2 veStarter.sh

```
#!/bin/sh
#usage veStarter DESCRIPTION_FILE APPLICATION


#determine standard values for....
# the fild separator; DEFAULT is ":"
stdSep=":"
# the insert position for new values in an existing variable; DEFAULT is "append"
stdPos="append"
# the action; you can choose between "new","modify", and "unset" ; DEFAULT is "modify"
stdAction="modify"

#currently not defined in the DESCRIPTION_FILE
stdHarnessDir="/.harness"
harnessDir=$stdHarnessDir
stdOsMountPoint=$harnessDir"/osfs"
stdOsMountPointCow=$harnessDir"/unionfsCow"
stdOsMountPointRo=$harnessDir"/unionfsRo"
stdWriteDirSuffix="_write"


#check user name for secure use of the su command
if test -n "$SUDO_USER";then
veuser=$SUDO_USER
else  veuser=$LOGNAME
fi
if test -z "$veuser" ;then
echo "Abort: Could not find a User!"
exit
fi

osMountPoint=$stdOsMountPoint
osMountPointCow=$stdOsMountPointCow
osMountPointRo=$stdOsMountPointRo
writeDirSuffix=$stdWriteDirSuffix

#read description file
. $1

#absolute paths

#if test -n "$3" ;then
# fakeDir=$3
#else  fakeDir=$ENV
#fi
fakeDir=$ENV

## start here
createVars()
{
```

```
#check user name
if test -n "$SUDO_USER";then
veuser=$SUDO_USER
else  veuser=$LOGNAME
fi

if test -z "$veuser" ;then
echo "Abort: Could not find a User!"
exit
fi

varCount="$VARS"
if test -z $varCount ;then
varCount=0;
fi
varX=1

while test $varX -le "$varCount" ;do

#read description file
. $1
echo "bin in Varschleife"
curVar='VAR'$varX

#create the variables
eval varName='$'$curVar
echo $varName
eval varValue='$'$curVar'_VALUE'
#eval varValue=$varValue
echo "varhallo:$LOGNAME"
echo $varValue
eval varAction='$'$curVar'_ACTION'
echo $varAction
eval varPos='$'$curVar'_INSERTPOS'
echo $varPos
eval varSep='$'$curVar'_SEPARATOR'
echo $varSep
eval varPattern='$'$curVar'_INSERTPATTERN'
echo $varPattern

if test -z $varSep; then
varSep=$stdSep;
fi
if test -z $varPos; then
varPos=$stdPos;
fi
if test -z $varAction; then
varAction=$stdAction;
fi

if test -z "$varName"; then
echo "VAR$varX has no name!"
else
eval varTemp='$'$varName
```

```
echo "test:$varTemp"

case $varAction in

"new") export "$varName=$varValue"
echo "export:"$varName="'"'$varValue'"'""
;;

"modify")
#NF: number of field elements in the variable
NF=‘echo $varTemp | awk 'BEGIN{FS="'$varSep'"} {print NF}'‘
append=0
if test "$varPos" = "append" ; then
varPos=‘expr $NF + 1‘
append=1
fi
prepend=0
if test "$varPos" = "prepend" ; then
varPos=1
prepend=1
echo "1"
fi

if test -n "$varPattern";then  #search pattern in $varTemp
echo "2"
x=1
part=""
while test $x -le $NF ;do

field=‘echo $varTemp | awk 'BEGIN{FS="'$varSep'"}{print $'$x'}'‘
if test "$field" = $varPattern ; then
echo "pattern"
# prepend
if test $varPos -eq 1; then
if test $prepend = 1 ; then
varPos=$x
else
varPos=‘expr $x + 1‘
fi
echo "5"
else #append
echo "append $x $varPos"
if test "$varPos" -eq ‘expr $NF + 1‘ -a $append = 1; then
varPos=‘expr $x + 1‘

else
if test "$varPos" -ge 0;then
varPos=‘expr $x + $varPos‘
echo "6:$varPos"
fi

if test "$varPos" -lt 0;then
varPos=‘expr $x + $varPos + 1‘
echo "3"
```

```
fi
fi
fi
#varPos=‘expr $x +$varPos‘

#break while loop
x=‘expr $NF + 1‘
fi

x=‘expr $x + 1‘
done
fi

#include at the end
echo "4 und $varPos und $NF"
if test "$varPos" -gt $NF ; then
export "$varName=$varTemp$varSep$varValue"

else # inlcude at beginning
if test "$varPos" -le 1 ; then
export "$varName=$varValue$varSep$varTemp"

else
x=1
part=""
while test $x -le $NF ;do

field=‘echo $varTemp | awk ’BEGIN{FS="’$varSep’"}{print $’$x’}’‘
if test $x -eq $varPos ; then
part=$part$varSep$varValue
fi
if test $x -eq 1;then
part=$field
else
part=$part$varSep$field
fi
x=‘expr $x + 1‘
done

export "$varName=$part"
fi


fi
;;

"unset") unset "$varName"

;;

esac
echo "zz:$varTemp"

fi
```

```
varX=‘expr $varX + 1‘

done

}

#check for include instructions this function needs an array to store includeX
#the function works only properly if one environment is bequeathed
checkIncludes()
{ echo 4 $1

includes=0

#read config file
. $1

includeCount=$includes
includeX=1


while test $includeX -le $includeCount ; do

eval includeFile=’$include’$includeX

checkIncludes $includeFile

includeX=‘expr $includeX + 1‘
#read config file again
. $1

done

createVars $1
}


checkIncludes $1

#mount dirs

temp=‘pwd‘
cd /

y=‘ls -al | wc -l‘
lstext=‘ls -abm‘

x=$y

while test $x -ge 1 ;do

name=‘echo $lstext | awk ’BEGIN{FS=", "}{print $’$x’}’ | sed ’s/\([^ \]\)/\\\\\1/g’‘

if test -n "$name";then
#echo NAme:$name
```

```
if eval test -d "/$name" -a "$name" != "." -a "$name" != ".."
          -a "/$name" != "$harnessDir" -a "/$name" != "/home" ; then

eval sudo mount -t unionfs -o dirs=/$name unionfs "$fakeDir/$osMountPointCow/$name"
#eval sudo unionctl "$fakeDir/$osMountPointCow/$name" --add --mode
rw "$fakeDir/$osMountPointCow/$name$writeDirSuffix"
#eval sudo unionctl "$fakeDir/$osMountPointCow/$name" --mode /$name ro

eval sudo mount -t unionfs -o dirs=/$name unionfs "$fakeDir/$osMountPointRo/$name"
#eval sudo unionctl "$fakeDir/$osMountPointCow/$name" --mode /$name ro

fi

fi
x=‘expr $x - 1‘

done

mount --bind / "$fakeDir/$osMountPoint"

chroot $fakeDir su "$veuser" $2


echo "shutdown environment"

x=$y

while test $x -ge 1 ;do

name=‘echo $lstext | awk ’BEGIN{FS=", "}{print $’$x’}’ | sed ’s/\([^ \]\)/\\\\\1/g’‘

if test -n "$name";then

if eval test -d "/$name" -a "$name" != "." -a "$name" != ".."
   -a "/$name" != "$harnessDir" -a "/$name" != "/home" ; then

echo umount:"$fakeDir/$osMountPointCow/$name"
eval sudo umount -t unionfs "$fakeDir/$osMountPointCow/$name"
eval sudo umount -t unionfs "$fakeDir/$osMountPointRo/$name"

fi

fi
x=‘expr $x - 1‘

done
sudo umount "$fakeDir/$osMountPoint"
```

B.3 recSynDelete.sh

```
#!/bin/sh
#delete all files and links
#recSynDelete: source target
source=$1
target=$2
echo "synDele:$1 $2 "
   temp=‘pwd‘

eval cd "$source"

#files=‘ls -abm | sed -e ’s/[,][ ]-/, .\/-/g’ -e ’s/\([^ \]\)/\\\\\1/g’ -e files=‘ls -abm | sed
            -e ’s/[,]$/, /g’ -e ’s/^[-]/ -/g’ -e ’s/[ ]-/ .\/-/g’ -e ’s/\([^ \]\)/\\\\\1/g’
            -e ’s/\\\, / . /g’‘  ’s/\\\, / . /g’‘
#files=‘ls -abm | sed -e ’s/[,]$/, /g’ -e ’s/^[-]/ -/g’ -e ’s/[ ]-/ .\/-/g’ -e ’s/\([^ \]\)/\\\\\1/g’
            -e ’s/\\\, / . /g’‘
emptyspace=" "
files=‘ls -abm | sed -e ’s/[,]$/, /g’ -e ’s/^[-]/ -/g’ -e ’s/[ ]-/ .\/-/g’ -e ’s/\([^ \]\)/\\\\\1/g’
             -e ’s/\\\, / $emptyspace /g’‘

#echo "files:"$files
eval cd "$target"
eval rm $files 2>/dev/null

eval cd "$source"
x=‘ls -al | wc -l‘
lstext=‘ls -abm‘
#echo "lstext:$lstext"

cd "$temp"

while test $x -ge 1 ;do
#name=‘echo $lstext | awk ’BEGIN{FS=", "}{print $’$x’}’ | sed -e ’s/^[.]$//g’ -e ’s/^[.][.]$//g’
         -e ’s/^-/.\/-/g’ -e ’s/\([^ \]\)/\\\\\1/g’‘
name=‘echo $lstext | awk ’BEGIN{FS=", "}{print $’$x’}’ | sed ’s/\([^ \]\)/\\\\\1/g’‘
#echo "name:$name"

if test -n "$name";then
#pwd
if eval test "$name" != ".." -a "$name" != "." -a -d "$target/$name"; then
#echo "yes:$name"

./recSynDelete "$source/$name" "$target/$name"

#else echo "no:$name"
fi
fi
x=‘expr $x - 1‘
#
done
#
#echo "synDel end"
```

B.4 XML Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.csm.ornl.gov"
xmlns="http://www.csm.ornl.gov"
elementFormDefault="qualified">

<!-- simple type definitions -->

<xs:simpleType type="sysString">
<xs:restriction base="xs:string">
<xs:minLength value="1"/>
<xs:restriction>
</xs:simpleType>



<xs:complexType mixed="true" name="userType" type="sysString">
<xs:sequence>
<xs:element name="id" type="xs:integer" minOccurs="0"/>
<xs:element name="firstname" type=:xs:string minOccurs="0"/>
<xs:element name="group" type="groupType" minOccurs="1" minOccurs="unbounded"</>
</xs:sequence>
</xs:complexType>

<xs:complexType mixed="true" name="groupType" type="sysString">
<xs:sequence>
<xs:element name="id" type="xs:integer" minOccurs="0"/>
<xs:element name="user" type="userType" minOccurs="0" minOccurs="unbounded"</>
</xs:sequence>
</xs:complexType>


<xs:complexType name="permissionType1">
<xs:restriction base="xs:string">
<xs:pattern value="(([r-][w-][x-]){3})|([0-7]{3})"/>
</xs:restriction>
</xs:complexType

<xs:complexType name="permissionType2">
<xs:sequence>
<xs:element name="owner" type="sysString"/>
<xs:element name="group" type="sysString"/>
<xs:restriction base="xs:string">
<xs:pattern value="(([r-][w-][x-]){3})|([0-7]{3})"/>
</xs:restriction>
</xs:sequence>
</xs:complexType

<xs:complexType name="permissionType3">
<xs:sequence>
<xs:element name="name" type="sysString" default="others">
<xs:element name="type" default="user"/>
```

```
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="user" />
          <xs:enumeration value="group" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>
<xs:element name="permission">
<xs:restriction base="xs:string">
       <xs:pattern value="([r-][w-][x-])|[0-7]"/>
  </xs:restriction>
</xs:element>
</xs:sequence>
</xs:complexType>

<xs:complexType name="permissionType">
<xs:choice>
<xs:element name="permission" type="permissionType1" />
<xs:element name="permission" type="permissionType2" />
<xs:element name="permission" type="permissionType3" maxOccurs="unbounded"/>
</xs:choice>
</xs:complexType>

<xs:complexType name="fileSourceType">
<xs:choice>
<xs:element name="uri" type="sysString"/>
<xs:any />
<!--xs:element name="ftpSource" type="ftpConnectionType/-->
<!--xs:element name="partitionSource" type="paritionSourceType/-->
<!--xs:element name="dev/hda2" type="deviceType/-->
  </xs:choice>
</xs:complexType>

<xs:complexType name="listSourceType">
<xs:element name="source" type="fileSourceType" />
<xs:element name="umask" type="permissionType"/>
<xs:element name="directory_umask" type="permissionType"/>
<xs:element name="getAllFiles" type="xs:boolean"/>
<xs:element name="getAllRecursive" type="xs:boolean"/>

<xs:element name="via" default="link"/>
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="copy" />
          <xs:enumeration value="link" />
<xs:enumeration value="create" />
<xs:enumeration value="mount" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>



<xs:element name="file"  type="fileType3" minOccurs="0" maxOccurs="unbounded" />
```

```
</xs:complexType>


<xs:complexType name="file" type="fileType1">
<xs:sequence>
<xs:element name="source" type="fileSourceType" />
<xs:element name="via" default="link"/>
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="copy" />
          <xs:enumeration value="link" />
<xs:enumeration value="create" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>
<xs:element name="target" type="sysString" />
<xs:element name="permission" type="permissionType" />
<xs:element name="type" default="regular">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="regular" />
     <xs:enumeration value="symLink" />
          <xs:enumeration value="hardLink" />
<xs:enumeration value="device" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>

<xs:sequence>
<xs:complexType

<xs:complexType name="file" type="fileType2">
<xs:sequence>
<xs:element name="source" type="fileSourceType" />
<xs:element name="via" default="link"/>
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="copy" />
          <xs:enumeration value="link" />
<xs:enumeration value="create" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>
<xs:element name="permission" type="permissionType" />
<xs:element name="type" default="regular">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="regular" />
     <xs:enumeration value="symLink" />
          <xs:enumeration value="hardLink" />
<xs:enumeration value="device" />
</xs:restriction base="xs:string">
</xs:simpleType>
```

```
</xs:element>
<xs:sequence>
<xs:complexType

<xs:complexType name="file" type="fileType3">
<xs:sequence>
<xs:element name="via" minOccurs="0"/>
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="copy" />
          <xs:enumeration value="link" />
<xs:enumeration value="create" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>
<xs:element name="permission" type="permissionType" minOccurs="0" />
<xs:element name="type" default="regular" minOccurs="0">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="regular" />
     <xs:enumeration value="symLink" />
          <xs:enumeration value="hardLink" />
<xs:enumeration value="device" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>

<!-- minSize maxSize setID stickybit namedPipe  -->
<xs:sequence>
<xs:complexType

<xs:element name="directory" type="directoryType">
   <xs:complexType>
<xs:sequence>
<xs:element name="name" type="sysString" />
<xs:element name="target" type="xs:string" default=""/>
<xs:element name="permission" type="permissionType"/>

<!-- xs:element name="umask"     default="likeTargetLocation" minOccurs="0"/-->
<!-- xs:element name="subdir_umask" default="likeTargetLocation" minOccurs="0"/-->

<xs:element name="file"  type="fileType2" minOccurs="0" maxOccurs="unbounded/">
<xs:element name="includeList"  type="listSourceType" minOccurs="0"  maxOccurs="unbounded/">

<xs:element name="subdirectory"type="directoryType" minOccurs="0" maxOccurs="unbounded" />

<!-- minSize maxSize setID stickybit -->
</xs:sequence>
   </xs:complexType>
</xs:element>

</xs:schema>

<!-- environment variables -->
```

```
<xs:element name="var">
   <xs:complexType>
<xs:sequence>
<xs:element name="name" type="sysString" />
<xs:element name="value" type="xs:string" default=""/>
<xs:element name="action" type="xs:string" default="modify"/>
<xs:simpleType>
<xs:restriction base="xs:string">
     <xs:enumeration value="unset" />
          <xs:enumeration value="new" />
<xs:enumeration value="modify" />
</xs:restriction base="xs:string">
</xs:simpleType>
</xs:element>

<xs:element name="insertPosition" type="xs:string" default="end"/>
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:pattern value="([1-9]*)"/>
</xs:restriction>
</xs:simpleType>
</xs:element>

</xs:sequence>
   </xs:complexType>
</xs:element>
```

C User Manual

Abstract

This manual gives a short description, how to use and create virtualised environments with two programs veCreator and veStarter. Both were developed to realize a virtualisation approach for the Harness Workbench.

Introduction

This virtualisation approach helps running applications on different computing platforms. There are four things to deal with:

- the application, which should be started in the virtualised environment,
- the veCreator program, which creates the virtual file structure,
- the veStarter program, which starts the application, and
- an environment description, which describes the properties of the environment, which should be emulated.

How To configure the System

For starting the veStarter or veCreator program, it is required that the operating system provide the `chroot` command. The execution of this command requires super user rights on many operating systems. Therefore it is required to have super user rights to start the both programs.

Normal user can be enabled to start the program by using the `sudo` command. Therefore, `sudo` needs to be configured. The following instructions require super user rights:

1. Open the sudo configuration file sudoers, this is mainly located in `/etc/sudoers`.
2. Attach one pair of the following lines at the end of the file to enable all or a specific user to start veStarter and veCreator. Then save and close the file.

```
ALL  localhost = NOPASSWD:veCreator
ALL  localhost = NOPASSWD:veStarter
```

Or

```
peter   localhost = NOPASSWD:veCreator
peter   localhost = NOPASSWD:veStarter
```

3. At last, it is required to change the permissions and ownerships of both executables.

```
chown root veCreator
chgrp root veCreator
chmod 744 veCreator

chown root veStarter
chgrp root veStarter
chmod 744 veStarter
```

For security reasons it is strongly recommended to omit a general execution rights for the chroot command.

How to use it

Using both programs requires neither starting a make file nor some installation instructions. There are two simple steps to process in order to run an application in new virtual environment.

1. start the veCreator
2. start the veStarter

After these steps, the application runs in the virtual environment. The application can be cancelled or stop as normal. It is not required to stop or delete the virtual environment, when the application is finished.

After step one, the application can be started arbitrary times in the virtual environment. It is also possible to start other application in the virtual environment, but this differs from application to application. It is better to start applications in their appropriated virtual environments.

veCreator

The veCreator reads the environment description and creates all file structure elements according to a given environment description. The veCreator application expects one parameter from the user, the file of the environment description.

The veCreator can be executed as follows:

```
sudo ./veCreator  FILE
```

veStarter

The veStarter is used to start applications in a created virtual environment. The application expects two parameters. The first parameter is a path to the description file of the virtual environment. The second parameter is the application which should be started.

```
sudo ./veCreator  FILE  APPLICATION
```

Trouble Shooting

Please make sure that your `sudo` configurations in `/etc/sudoers` is correct, if you got an error message related to missing executing permission.

For other starting problems please ensure that you are using the compatible shell. Both programs can be started form the bourne shell (sh) or bourne again shell (bash). If the programs do not run properly, please ensure that the used shell is really a bourne or bash shell. In some Unix/Linux distributions sh is only an symbolic link an other shell like ksh or csh. For this case please install a bourne or bash shell on your system.

For more help, and detailed information please contact bjoern.koenning@gmx.de.