FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Modular Approach to Configuration Storage

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Raab
Matrikelnummer 0425830

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 29.09.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Declaration

Markus Raab
Kandgasse 7/2/4/8, 1070 Wien

”Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.”

_____ _____

(Ort, Datum) (Unterschrift)

Acknowledgements

Many thanks to Kira Gysel for her courtesy and assistance in the beginning of the work.

I would like to thank Philipp Gühring for his valuable tips on binary values and forensic logging and also to Mathieu Lacage for his assistance on the dlopen issue. Special thanks to Kai-Uwe Behrmann, who is one of the pioneer users of Elektra.

Furthermore, I would like to thank the people of Debienna who were always there to give a helping hand. We discussed many topics concerning configuration. In particular, I would like to mention Christian Amsüss for his ideas for renaming keys and helping on character encoding as well as Albert Dengg for many technical details, especially for the hints on journalling. Harald Geyer was always very helpful for all kinds of questions and gave valuable tips related to LaTeX and layout. Special thanks to Andreas Leitgeb for supporting me with proof-reading. Nedko Tantilov sustained me with concrete suggestions.

I want to give distinct gratitude to Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam for guiding me through the work and for being highly responsive to all my concerns and questions.

Last, but not least, I am deeply grateful for the backing I got from my family.

I do apologise to all the people I have forgotten to mention here!

Zusammenfassung

Programmbibliotheken wie Elektra ermöglichen allen lokal installierten Programmen den Zugriff auf eine gemeinsame Konfigurationsdatenbank. Benutzer von Applikationen wünschen, dass ihre Programme stärker in das Gesamtsystem integriert sind, dass die Einstellungen – durch Benachrichtigungen über Änderungen – immer am neuesten Stand gehalten werden, dass ihre bevorzugten Konfigurationsformate unterstützt werden und dass sie trotzdem die Gewissheit haben, dass beim Speichern der Optionen nur funktionierende Einstellungen in die Konfigurationsdatenbank gelangen. Das konkrete Problem dabei ist, dass die genauen Anforderungen je nach Applikation und Betriebssystem variieren und dass es oftmals gleichwertige oder konkurrierende Möglichkeiten gibt, den Aufgaben gerecht zu werden.

In dieser Diplomarbeit wurde das Problem durch Modifikationen von Elektra gelöst. Die einzelnen Aufgaben werden in austauschbaren, wiederverwendbaren Plugins implementiert. Mehrere solche Plugins zusammen bauen einen Zugriff auf Teile der Konfigurationsdatenbank auf, welche dann gemeinsam die gesamten Anforderungen erfüllen.

Zur Überprüfung dieses Ansatzes wurde eine Reihe von Plugins implementiert und in verschiedenen Szenarien versuchsweise eingesetzt. Dabei zeigt sich, dass sich beim Herausschreiben der Konfiguration Vorteile ergeben, wenn die Überprüfung der Struktur der Optionen und die Validierung der einzelnen Werte in zwei Phasen gegliedert sind. Dadurch wird ermöglicht, verschiedenste Validierungen – mitunter auch applikations- oder plugin-spezifisch – zu kombinieren. Es wurden verschiedene Methoden erprobt, wie solche Plugins entwickelt werden können. Dabei stellte sich heraus, dass Plugins auch sehr schnell realisiert werden können, wenn sie das Potential von bereits existierenden Plugins und Bibliotheken nutzen. Es ist zudem elegant möglich, vom Betriebssystem abhängige Teile von unabhängigen zu trennen. Durch das Einhängen der Plugins in die gemeinsame Konfigurationsdatenbank während der Laufzeit werden die Eigenschaften und Funktionalitäten für diese Teile verändert. Aber durch Überprüfung von Kontrakten, welche die Plugins exportieren, kann trotz der zusätzlichen Flexibilität eine reibungslose Zusammenarbeit garantiert werden. Plugins können sich sogar darauf verlassen, dass bestimmte Aktionen bereits vorher durchgeführt wurden.

Abstract

Libraries like Elektra provide access to a shared configuration database across all locally installed applications. Users prefer applications integrated well within the system they use. They are in favour of settings that are always up to date. Users also want their favourite configuration file formats to be supported. They want a guarantee that the system storing their options only accepts a working configuration. The concrete problem is that the specific requirements vary according to the programs and the operating system. Similar or competing possibilities exist to achieve specific tasks.

We address the problem by modifying Elektra in this thesis. A reusable, exchangeable plugin cares for a specific concern. Several plugins together build up the access to a part of the configuration database that deals with the complete users' requirements.

We implemented a number of plugins to verify the approach. We tested the implementations in different use cases. It soon became clear that checking the configuration before storing is much more flexible if the validation of the structure and the values are separated in two phases. It facilitates combinations of several checks. Occasionally these checks are made specifically for an application or plugin. We evaluated and tried different ways in which to implement such plugins. We came to the conclusion that plugins can be realised quickly if the potential of existing plugins and libraries is used. Code needing operating system specific facilities can be easily separated into other plugins. As a result of mounting the plugins, the capabilities and features of the shared configuration database changes. But because of checking contracts exported by plugins, the interplay of plugins is guaranteed even though the system is now much more flexible. It even goes so far that plugins can count on actions taken by other plugins.

Contents

CHAPTER 1

Introduction

First we will clarify what definitions we use throughout this thesis. Then we will give an introduction to the Elektra software project that will accompany the reader throughout. After we have expanded on the background, we will define the goal of this thesis and describe how to reach it.

1.1 Preliminaries

1.1.1 Definitions

In this thesis we use the following definitions:

Configurations contain user preferences or other application settings.

Configuration storage makes this information permanent. The application will read the configuration at every start, but it is only stored if a user changes settings.

Key databases are used because of these constraints. They can do fast key lookups and the keys can be structured hierarchically by defining separators in the key names. Unlike SQL databases, the key name is the only primary key; there are no foreign keys, and no query language exists.

Global key database provides global access to all key databases of all applications in a system that wants to access a key database.

1.1.2 Conventions

Throughout the thesis we use the following conventions:

Names of classes and methods are written in `typewriter` style.

Method names start with a lower case letter and end with brackets, for example, `kdbOpen()`.

Empty brackets mean that parameters are irrelevant for the description; not that there are none.

Names of classes start with a capital letter, for example, `Key`.

Names of both, classes and methods, use `CamelCase` notation.

Newly introduced words that are explained in the surrounding text are written in SMALL CAPS.

Important words that send a central message are written in *italics*.

Commands, that can be typed inti a shell as a user, start with >.

Both newly introduced and important words will show up in the index at the end.

1.2 Elektra

ELEKTRA is a library implementing access to a global key database. When we refer to Elektra without a version number, we mean the version $0.8mile4$ which is the version developed while working on this thesis[1]. Information on Elektra can be found on the website `http://www.libelektra.org`. To ELEKTRIFY an application means to change the application so that it uses Elektra afterwards.

Figure 1.1: Elektra's Logo, thanks to Studio-HB

1.2.1 Motivation

Why Elektra?

Configurations, settings and preferences are hierarchical data structures of keys, each consisting of a name and a value. They can be used to configure software for the user's needs. Because these settings stay the same across restarts of the program, they need to be stored permanently. In the beginning this was done with primitive text files. Possibilities to structure the text were added later.

Nearly every system developed its own way to read preferences, but some systems can also change them. Because the graphical user interface can be tweaked in many ways, the most encompassing systems

[1] 4^{th} and last milestone

emerged from this area. Some got a de facto standard for a desktop environment (kconfig, gconfig) or even an operating system (Windows Registry, Open Directory). But they have a common problem: they are bound to the platform for which they were developed. On the other hand, there are many libraries that do a good job in parsing and writing configuration files. These tools are, however, not powerful enough to keep the configuration independent from the operating system's details, for example, how to resolve the file name of the configuration file.

That is where Elektra comes in to fill the gap. On the one hand, Elektra is not tied to any platform or operating system. On the other hand, Elektra is powerful enough to be useful immediately for what it is written for: to access configuration.

Why is it important?

The configuration files that represent key databases can have binary or humanly-readable formats. From the latter, an unmanageable number is established. Developers of programs tend to document the format of the configuration file extensively. The configuration file may give a special flavour to a specific program and users frequently need it.

Sometimes limitations in the configuration file even lead to rewrites of software. For example, INETD has a non-modular flat configuration file that is not extensible because of a limited number of rows. In order to extend its functionality, the program had to be rewritten with a new approach to configuration: XINETD emerged. Both of these projects are now almost defined by their configuration files giving them identity and separating them from each other. Elektra has introduced BACKENDS to support the storage of key databases in different formats.

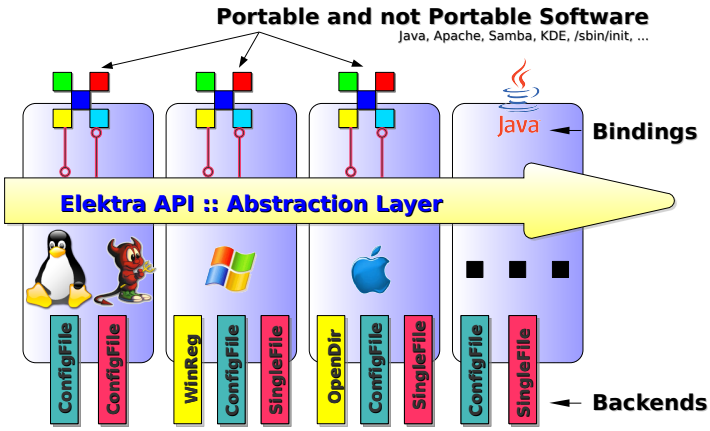

Figure 1.2: Elektra as Abstraction Layer, thanks to Avi Alkalay[1]

Elektra abstracts configuration so that applications can receive and store settings without carrying information about how and where these are actually stored. It is the purpose of the backends to implement these details. What makes the difference is the situation that every program can access any configuration because of the abstraction. In the example on page 3, Elektra allows an elektrified inetd respective xinetd to store its configuration in `/etc/inetd.conf` respective `/etc/xinetd.conf`. Additionally, each other program interested in these preferences can access them in a uniform way.

To support a global key database, a mutual agreement on some level is needed. Elektra provides this common layer with its data structures. Each elektrified application lies on top of this abstraction layer and it can talk to each part of the global key database using the classes presented next.

1.2.2 Classes

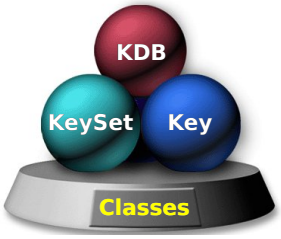

Figure 1.3: Three Classes, thanks to Avi Alkalay[1]

Key

A `Key` consists of a name, a value and metadata. It is the atomic unit in the key database. Its main purpose is that it can be serialised to be written out to permanent storage. It can be added to several aggregates using reference counting. Putting `Key` objects into other data structures of supported programming languages presents no problem.

KeySet

The central data structure in Elektra is a `KeySet`. It aggregates `Key` objects in order to describe configuration in an easy but complete way. As the name ”set” already implies every `Key` in a `KeySet` has a unique name. A user can iterate over the `Key` objects of a `KeySet`. `KeySet` sorts the keys by their names. This yields a deterministic order advantage. So, independent of the appending sequence and, in particular, the number of fetches and updates, `KeySet` guarantees the same order of the `Key` objects. Some configuration storage systems need this property, because they cannot remember a specific order. On the other hand, any particular order can easily be introduced [2].

[2] As we will show in Section 4.3.3 on page 73

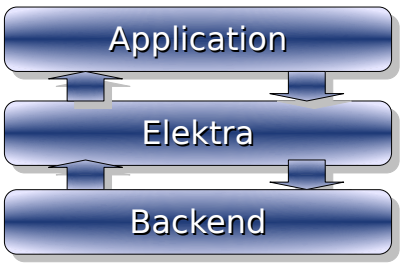

Figure 1.4: Flow of `KeySet`

The arrows in Figure 1.4 represent the flow of `KeySet` objects. We see that on the one side backends generate or store a `KeySet` object and, on the other side, elektrified applications receive and send a `KeySet` object. Both sides, as well as the core in between, have the possibility to iterate, update, modify, extend and reduce the key set. Appending of new or existing `Key` objects extends the key set. Otherwise it can be reduced if keys are POPPED out. The `Key` object becomes independent from the `KeySet` afterwards. The user can still change such a key or append it into another key set. The affiliation to a key set is not exclusive.

Every key in a `KeySet` object has a unique name. Appending `Key` objects with the same name will override the already existing `Key` object.

KDB

While objects of `Key` and `KeySet` only reside in memory, Elektra's third class `KDB` actually provides access to the global key database. `KDB`, an abbreviation of key database, is responsible for actually storing and receiving configuration. `KeySet` represents the configuration when communicating with `KDB`. The typical elektrified application collects its configuration by one or many calls of `kdbGet()`. As soon as the program finishes its work with the `KeySet`, `kdbSet()` is in charge of writing all changes back to the key database.

This technique has some advantages. First, applications have full control over modifying `Key` and `KeySet` objects without touching the key database. Second, the decision how many `KeySet` objects the application administrates is left to the application. It can choose how to split up the `KeySet` objects. The main reason for this technique is that for backend development the same data structure is used, and as we will see, the borderline between application and backend development becomes blurred.

The application adapts the configuration between `kdbGet()` and `kdbSet()` in memory. The modifications are not only faster, they also allow large atomic configuration upgrades, robust merging of settings and handling of complicated inter-relationships between keys without problematic interstages. Elektrified applications, however, should be aware of conflicts. It can happen that the key database is changed while

working with a `KeySet`. Then, attempts to use `kdbSet()` lead to a conflict. `KDB` detects such situations gracefully and lets the application decide which configuration should be used[3].

1.2.3 Concepts

Key Names

Every `Key` object with the same name will receive the very same information from the global key database. The name locates a *unique key* in the key database. Key names are always absolute; so no parent or other information is needed. That makes a `Key` self-contained and independent both in memory and storage.

Every key name starts with `user` or `system` prefixes that spawn two key hierarchies. The shared SYSTEM CONFIGURATION is identical for every user. It contains, for example, information about system daemons, network related preferences and default settings for software. These keys are created when software is installed, and removed when software is *purged*. Only the administrator can change system configuration.

Listing 1.1: Examples of valid system key names

```
system
system/hosts/hostname
system/sw/apache/httpd/num_processes
system/sw/apps/abc/current/default-setting
```

On the other hand, USER CONFIGURATION is empty until the user changes some preferences. User configuration affects only a single user. The user's settings can contain information about the user's environment, preferred applications and anything not useful for the rest of the system.

Listing 1.2: Examples of valid user key names

```
user
user/env/#1/LD_LIBRARY_PATH
user/sw/apps/abc/current/default-setting
user/sw/kde/kicker/preferred_applications/#1
```

The slash (`/`) separates key names and structures them hierarchically. If two keys start with the same key names, but one key name continues after a slash, this key is BELOW the other and is called a SUBKEY. For example `user/sw/apps/abc/current` is a subkey of the key `user/sw/apps`. The key is not directly below but, for example, `user/sw/apps/abc` is. `keyRel()` implements a way to decide the relation between two keys.

Cascading

CASCADING is the triggering of secondary actions. For configuration it means that first the user configuration is read and if this attempt fails, the system configuration is used as fallback.

[3] As we will show in Section 3.4.3 on page 59

The idea is that the application installs a configuration storage with default settings that can only be changed by the administrator. But every user has the possibility to override parts of this *system configuration* regarding the user's needs in the *user configuration*. To sum up, besides system configuration, users have their own key databases that can override the settings according to their preferences.

Mounting

MOUNTING allegorises a common technique for *file systems*. File systems on different partitions or devices can be added to the currently accessible file system. Mounting is typically used to access data from external media. A more advanced use case presents mounting a file system that is optimised for specific purposes, for example, one that can handle many small files well. Mounting also allows us to access data via network storage. As a result, mounting of file systems has proved to be extremely successful.

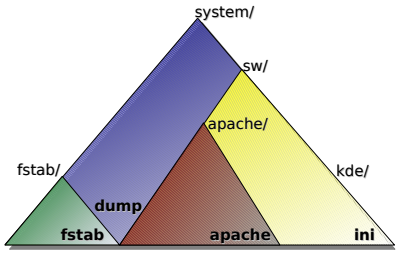

Figure 1.5: Mounting in Elektra

MOUNTING in Elektra[35] specifically allows us to map a part of the global key database to be handled by a different storage. A difference to file systems is that *key names* express what file names express in a file system. And instead of file systems writing to block devices, backends writing to key databases are mounted into the global key database. Mounting allows multiple backends to deal with configuration at the same time. Each of them is responsible for its own subtree of the global key database. In Figure 1.5, we see several such subtrees, for example `system/sw` or `system/sw/apache`. Backends, that are written in bold, handle the configuration in these subtrees. Mounting works for file systems only if the file system below is accessible and a directory exists at the mount point. Elektra does not enforce such restrictions.

Environment

Elektra solves the task of accessing the configuration storage. Additionally, an environment gathers around Elektra to help with minor problems that appear every day. Maybe the administrator needs a cron job that periodically changes the settings of a service. Maybe the user wants to have an overview of the whole configuration to learn what can be tweaked. Maybe the developer needs to fully export the

configuration the program had when a failure occurred. These tasks have in common that they become trivial once a programmatic access to a global key database exists.

In this subsection we give an overview of the command-line tool `kdb`. It is part of Elektra's environment and performs the mentioned tasks. `kdb` consists of individual subprograms. The programs are independent, but can access a shared part that provides functionality too specific to be in the library – for example, pretty printing of error messages and warnings. Most parts of this suite are short programs which basically call `kdbGet()`, do something with the data structure and eventually write it back using `kdbSet()`. Note that the command-line tool `kdb` should not be confused with the class `KDB`.

`kdb` was rewritten as part of this thesis with a new architecture. Now every part of the application suite will be able to accept its own command line arguments and will have its own documentation. Also a completely new feature `mount` arose.

Only a few commands are enough for daily use. We can retrieve a key by:

```
> kdb get user/keyname
```

We store a key permanently with a value given by:

```
> kdb set user/keyname value
```

We list all available keys arranged below a key by:

```
> kdb ls user/keyname
```

Many other tools beside `kdb` are possible. They may be more convenient depending on the situation. Preference dialogues, graphical editors, web-interfaces and web services can all provide access to the global key database.

1.3 Work Related to Elektra

Settings and preferences are ubiquitous in every software. For this reason, a vast amount of software has emerged on this topic. Most of the other projects, however, do not devote their work exclusively to configuration.

1.3.1 Configuration Libraries

We have already mentioned that Elektra provides the features of a *configuration library*. There are countless libraries out there that parse and some even generate configuration files. But there is a big catch: These libraries do not provide an abstract view of configuration. They require programmers to open the files themselves or at least to remember the path to them. Such information is itself configuration and inherently operating system dependent. Therefore, it is not possible to give default values that just work.

Another feature mostly missing in these libraries is cascading. It is not difficult to implement, but disturbing if it is not available at all or does not work correctly.

1.3.2 Other Projects

Uniconf

A project that shares some of the goals with Elektra, but uses a different approach is UNICONF[34, 26]. Besides a stand-alone library it supports a daemon mode. With the daemon running, it has the disadvantage of protocol overhead and a single point of failure. On the other hand, Uniconf can also notify applications when configuration files change and can work in a distributed mode. The project does not solve the problem of how to configure the configuration system. So we still have to pass a so-called MONIKER STRING[26] that contains the knowledge where to find the configuration. The plugin system is flexible, but does not depend on the key name. So it is not possible to use different plugins for different applications and still have a global key database.

Debconf

DEBCONF is a set of Debian-tools[24] that separates user interaction from the configuration process for the system's configuration when packages are installed or upgraded. Debconf itself does not change the configuration files. Instead, the administrator is asked questions depending on template files using different front ends. These answers are then cached. The configuration changes themselves are applied by post-install scripts of the particular package. Exactly for this last step Elektra could yield much better results, because it can compare configuration on a per-key basis and thus handle conflicts in a much more fine-grained way.

Freedesktop.org

The `freedesktop.org` initiative unifies different desktops using the X Window System in various ways. The main focus, however, lies in KDE and Gnome integration. One of the most disturbing and still unresolved problems is configuration. Elektra intends to fill this gap in the future.

Already available is the so-called XDG Base Directory Specification[5]. Perhaps it will define file formats in the future, but currently only the paths are specified. To do so, it introduces some environment variables that help to find the actual configuration. Only if they are unset or empty, a built-in fallback should be used. If desired, elektrified applications behave in a XDG conforming way.

1.3.3 Key Databases

We give an unavoidably incomplete overview of the huge variety of existing configuration files and other *key databases*.

Configuration Formats

Each of them has properties with debatable advantages and disadvantages. We concluded that none of these formats can be considered best, but they have some differences in field of applications. Elektra, however, makes it possible to implement each of these formats:

1. Many formats have been invented exactly for configuration. Parsers and generators are often available for them. Examples are INI and its dialects, xfree-, apache- and C-like configuration formats.

2. File formats invented for the exchange of information are often able to contain arbitrary hierarchical data – so they can also store configuration. Examples are XML, JSON and YAML[6]. But, for example, XML is not perfectly suitable for editable configuration because it has some inconvenient ROUND-TRIPPING PROPERTIES[38]. But it is adequate for exchanging configuration between systems.

Other Key Databases

It is currently unclear how well Elektra can support other types of key databases:

1. There are binary key databases developed for fast configuration access. When keys are stored, they are compressed and indexed in a way that they can be retrieved efficiently. Their obvious drawback is that the files are not humanly readable. `libeet`[20] is an example.

2. There are conventional databases that can be used when the applications already have data stored in them. The disadvantage is that they are not humanly readable and often not deducible[4] if something has changed. It is, however, possible that the first access is already fast, opposed to other key databases, where a cold startup can yield seriously slow results.

1.4 Goal of this Thesis

The *goal of this thesis* is to improve Elektra in a way so that developers, administrators and users can profit from a better modularity that allows them to configure the features, capabilities and behaviour of the configuration storage according to their needs.

Up to now, we have not considered these specific requirements. But each of Elektra's users has different considerations on how the configuration system should work:

- Elektra should support different configuration file formats.
- Elektra should support different ways to escape and encode the content of configuration files.
- Elektra should be able to log and notify other software on any configuration changes.
- Elektra should be able to avoid the problem that any invalid configuration is written into the permanent storage.
- Elektra should be able to provide different mechanisms to locate these configuration files.
- Elektra should be able to detect configuration updates and conflicts.

[4] On files it is possible with the syscall `stat()`.

Each of these items brings a new dimension to the overall problem because for each requirement a variety of competing solutions exists. Some concrete solutions work only for a specific operating system or even only for a specific application. The modular approach should be able to bring the wide range of different considerations under one roof.

1.5 Methodology

The way to reach the goal of this thesis consists of the following steps. First we will analyse the previous version of Elektra. We will show its problems and limitations in detail. Using this information we will locate improvement opportunities. Next we will propose several ways how to circumvent these points. We will consider and balance the different alternatives in a conscious way to find the best one. Then we will implement the chosen solution. We will evaluate the code, the running system in different scenarios and the new semantics of the key database. At that point, we will analyse the results and finally present our conclusions.

In the following part of the section we will specify the tools we will use for Elektra and describe why some tools will be replaced. After that we will finish this chapter by describing the benchmark methods.

1.5.1 Development Methods

We will employ an *agile* and *test-driven* development process. For this thesis we will conduct a *bidirectional* approach. On the one hand, the design is developed bottom-up and first basic facilities are implemented. On the other hand, it is also important to use a top-down approach to model the overall framework and to ensure that the facilities work together smoothly.

1.5.2 Tools

Elektra consists of many parts and provides some tools by itself as explained in Section 1.2.3 on page 7. All of them link against Elektra's core library. The core provides the fundamental data structures and functions to work with them. There is no other choice but to write this part in C because Elektra should be available at early stages during the boot phase[5] and for every software regardless of the language used. To support access to the global key database from other programming languages, language bindings are needed. It is possible to write bindings for nearly every other language if the library is written in C. Tools like SWIG[6] simplify this task.

A problem of C code is that the programmer needs to check for every function invocation if it has returned a failure code. It is also the programmer's challenge to clean up everything he or she allocated and to check every return value. These checks lead to a long code. So we keep the C++ bindings up to date. Because of exceptions and smart pointers, these disadvantages do not exist there. The executable can be as fast and compact as when working with C Code. So some plugins will be written taking advantage of C++.

[5] Specifically, under UNIX even `/sbin/init` can be elektrified.
[6] Simplified Wrapper and Interface Generator

For a library with modules as flexible as in Elektra, there are special requirements to the BUILD TOOLS. Elektra originally used autoconf and automake which was unsatisfactory for a number of reasons. First of all, there are no directory-based possibilities to check or depend on other libraries. But exactly this is a central feature requirement for a modularised version of Elektra. The tool of choice will be CMAKE which does not have these problems.

Building Elektra will consist of building some programs first. The purpose of these programs is to generate code needed to build other parts of Elektra. One of these programs will produce C code containing a list of the static modules. Another program will generate C code that maps error numbers to other error-related information. CMake directly supports the building of programs that generate code needed later.

Until recently, we used SVN as REVISION CONTROL SYSTEM. However, it has two major problems. The first one is that the history of a branch is linear only. It is slow to make branches and difficult to merge them, which does not meet the requirements for trying out new features in branches. The second one is that only a central development model is supported, but a decentralised and flexible methodology is needed. GIT will be the tool of choice for this method.

1.5.3 Benchmark Methods

For benchmarking, as in any other experiment, one aspect is crucial: *reproducibility*. It must be possible for any other scientist to set up the very same experiment and try it again. Ideally it should always yield equivalent results. This can be supported by providing the exact version of source code which was used to run the test, the build options used, the input data given and all output data produced. But there are some limits because someone else uses different hardware, other benchmarks tools, another file system or maybe another operating system. So considerable effort is necessary to eliminate these factors. One possibility is to give only comparative statements which are the same even if the absolute values differ. Nearly every operating and file system caches access so that in future requests the same data can be served faster[7]. The effects can be significant and should be considered in the benchmarks.

To make the benchmarks as reproducible as possible, all this information needs to be given for every benchmark. The methodology we will use is to first bring the source code to exactly the state which should be tested. This state will be committed with *git*. Then the input files and build options will be created and the tests will be executed. The process will be completed by committing all this information together with the results in a separate `benchmark` branch in git.

[7] A noteworthy exception is the file system tmpfs that is already completely in memory.

CHAPTER 2

Approach

2.1 Problem

Elektra 0.7, the version before the work on this thesis started, already fulfilled the basic principles needed for sharing configuration and the other features as described in Chapter 1. This section describes what we learned from this previous version.

2.1.1 File System Semantics

This section explains why using a file system for configuration storage is not a good idea.

filesys

FILESYS was the first backend. It implemented the principle that every key is represented by a single file. The key name was actually mapped to a file name and the value and the comment was written to that file.

If the backend `filesys` was the ideal solution, Elektra's API[1] would be of limited use[2] because well-established APIs for accessing files are available in every applicable programming language.

Elektra 0.7 already supported more than one backend, but `filesys` was the only backend implementing the full semantics.

Limitations of File Systems

One file per key turned out to be inefficient because of the file system's practical limitations. In most file systems, a file needs about four kilobytes[3] of space, no matter how little content is in it. Thus the file

[1] API is an abbreviation for application programming interface. APIs consist of functions and data structures that are implemented by a software program.

[2] Cascading (see Section 1.2.3 on page 6), type checking and optional cross-cutting features would be missing. The storage problem itself and the location of a key in a key database would be solved.

[3] Depends on the block size, four kilobytes is a common value often used as default.

system wastes 99.9% of the space if keys have a payload of four bytes. We can argue, however, that we can use a file system[36] which does not have this problem.

Many additional restrictions occur for portable access. The file name length in POSIX is limited to fourteen characters. Additionally, issues with case sensitivity are likely. The common denominator for all file systems is a surprisingly small one. If, for example, the traditional FAT file system should be supported, file names are limited to eight characters and case insensitivity.

An even more severe problem arises: The file system's semantics are not well suited for configuration at all.

On the one hand, there are many file system *features that are not needed* for configuration. Many security rules determine if it is possible to access a file or not. It has to be checked for every `open()` system call on *each* level if the current user is allowed to access it. File systems have a strict hierarchy. It is not possible to create a file in a non-existing directory. We will refer to such a missing object as HOLE. File systems do not support such holes. Some operating systems define files with a special name to be hidden. Such semantics are of limited use for a configuration system[4]. A single *root directory* is not a useful concept for configuration. Instead, the system configuration and each user configuration has its own root. These root keys themselves are typically not needed. The use of *current directories* is a rather problematic topic, even in file systems. Because current directories are defined per process, multithreaded applications using relative file names do not work well if the current working directory is changed. `openat()` was introduced because of that issue. There is additional metadata of files which is typically not needed for configuration: atime, mtime, ctime, uid, gid and mode just to name a few. Additional file types[5] are not needed either. Features like sparse files are ridiculous for the small strings, that key values typically are.

On the other hand, there are many *features missing* in file systems that we need in a serious key database. Creating a whole hierarchy of files at once atomically is not possible. Ways to achieve this are currently academic[45] and not portable. Directories cannot have any content next to the files below. Swap semantics are missing: it is not possible to rename a file without removing the target first. Locks are not powerful enough either. It is not possible to lock hierarchies of files at once. To sum up, file systems are not suitable to hold configuration with one entry per file. Instead, they are perfectly suitable to hold larger pieces of information like configuration files.

Capabilities

Every backend, except `filesys`, was unable to represent full file system semantics. CAPABILITIES described the differences between `filesys` and another backend[6]. Capabilities made it possible to implement a backend different from the way `filesys` works and let the backend still have predictable behaviour. The user could query a backend if it was capable of a specific detail of file system semantics.

[4] Elektra up to version 0.7 supported it anyway because of its tight integration to a file system.

[5] For example, device files, links, fifos and sockets.

[6] For POSIX file systems a similar technique is `pathconf()`. It allows the user to query the capabilities of a specific mounted file system given by path.

Capabilities were initially introduced to make backend development easier, because they also expressed the disabilities of a backend. For example, it was possible for a backend to claim that it is not aware of comments.

Getting a single key works well for `filesys`. On the other hand, for configuration files the whole content must be parsed even for a single key. Capabilities were able to describe that the backend will always retrieve and store all keys and will not be able to retrieve and store individual keys. This restriction simplified the implementation of such backends considerably. Capabilities allowed us to implement fstab, passwd and hosts backends.

But we soon found the limits of capabilities. Capabilities were unable to describe:

- that some key names are not allowed.
- that not every structure of configuration is allowed.
- that some characters are not allowed.
- that only specific key values are allowed.

The semantics of the `filesys` backend[7] were complex and bulky. The underlying storage contained metadata like modification times and file modes. Other backends deactivated all these semantics by capabilities. But the capabilities overburdened the API because of the comprehensive file system semantics. The capabilities system became hard to understand, ambiguous, and far from being stable.

The main problem was that the complexity was just moved to the applications. In the end, application developers had to understand what the backend can write out. They no longer had confidence that the storage was able to understand the full semantics. Elektra's core was unable to hide that fact. Capabilities did not turn out to make backend development much easier. Instead, Elektra was more difficult to use with capabilities. Capabilities are no longer part of Elektra for these reasons.

2.1.2 Missing Modularity

Sharing code between backends is known to be a critical task. In Elektra 0.7, contrary to what is expected, code from a backend could not be reused. Missing reuse presents a major problem because backends have many common aspects.

libhelper

LIBHELPER was an attempt to share code between backends. The idea was to write a library which helps backends to fulfil common tasks and that other backends can profit from the code.

`libhelper` tried to collect common code together. The backend `filesys` had existing code to escape characters, resolve file names from key names and more similar tasks. `libhelper` extracted this code because other backends could also need this code.

[7] Up to version 0.7, the file system semantics were also Elektra's semantics.

While it was possible to separate `libhelper` from the core, it was risky because it was optional to use it. Writers of backends would avoid using this additional library then. `libhelper` eventually became part of Elektra's core.

After some time, however, it became clear that it is hard to write such functions generally enough to be useful for every backend. In the end, `filesys` was nearly the only candidate that actually profited from `libhelper`.

Testing

The functions of `libhelper` were hardly testable because they often relied on environmental information which were dependent on the operating system. Because they were in the core, they should be portable and they need to react differently in various scenarios. The test cases have to take that into account.

The worst problem was that the functions were incapable of enforcing the backends to have specific behaviour because the backend developer could always choose not to use these functions. For example, the *path resolution* of the `kdb` utility was not always correct because of that fact. Elektra's users want to know where their configuration files are. This information was determined by calling the path resolution of `libhelper`. If backends did not use the path resolution of `libhelper`, the user obtains wrong information.

Functions of `libhelper` were not flexible enough because it was not possible to change their behaviour at run time for specific backends. Last, but not least, a potential backend developer had to learn the complete `libhelper` API before he or she could begin to write a backend. The approach of using a common library for all backends failed for these reasons.

2.1.3 Missing Features

The purpose of a backend is the access to configuration storage. But, additionally, it needs to cope with many other features like encoding key values.

Dependences

These concerns typically need external dependences. `libhelper` was able to change the encoding of strings and key names by using the external library ICONV[8]. Key values, names and comments that have encodings distinct from UTF-8 were converted to UTF-8 first. So, in the storage of `filesys` all strings were UTF-8 encoded. As practical as this idea is for some users, it is useless or even disturbing for others.

Elektra's core in the version 0.7 directly communicates with one backend. This approach makes it optional whether to implement advanced features in the core or in the backend. In order to be portable, Elektra's core must avoid dependences. In backends, dependences are acceptable if, and only if, they support the specific task of this backend. The main task of a backend, however, is reading and writing configuration and not converting between encodings.

[8] On some systems this functionality is actually integrated in `libc`.

Build System

It has been impossible to turn features of backends on and off at run time. A partial solution of this problem is the former complex *build system* in Elektra that allowed us to switch functionality on and off at compile time. The more combinations of build system options are available the more likely unexpected results occur in some rare cases. No information is available that would help us to understand how these features interact.

Precompiled versions of Elektra either lack some features or have considerable space and time overhead because of unnecessary features. The ultimate goal is that the administrator can choose at run time if he or she wants to have a feature.

Development Time

Writing backends has shown to be a time-consuming task because of the many requirements exposed to backends. The last part of this section connects the previous problems to the time effort needed by the backend programmer.

Let us assume that a programmer receives a list of requirements the configuration system needs to handle. That could be notification, logging and a specific configuration format. It is very unlikely that an existing backend already fulfils these requirements.

Writing a completely new backend means the same effort as writing a new configuration library. As a result, programmers will probably decide to build their own solutions. Elektra will only be considered useful in situations where its benefit of providing a global key database is really needed.

2.2 New Approach

We have seen that implementing too many features in one backend is problematic. It introduces unwanted external dependences and leads to less portable backends. Many different aspects clutter the code making the backends unmaintainable. Features of other backends cannot be taken with ease because they are interwoven with other code.

To support the reuse of a backend, they must be useful in different situations. Separation of concerns is required for that. Each backend needs to have a specific purpose rather than implementing the full semantics of Elektra.

It was impossible to implement powerful and feature-rich backends so far because of the lack of modularity. Desirable features like notification and type checking have always been in the developers' heads, but there was no place where it would fit in without making the system unmaintainable, complex and full of unwanted external dependences.

To solve this dilemma, we propose that MULTIPLE PLUGINS together build up a backend. The key set processed by one plugin will be passed to the next. This approach allows us to implement separate features in separate plugins. Plugins concerned with reading and writing to permanent storage are called STORAGE PLUGINS. We cleaned existing backends from other tasks so that their only job now is related to the storage. Using this approach, the plugins provide the desired separation of concerns inside a backend.

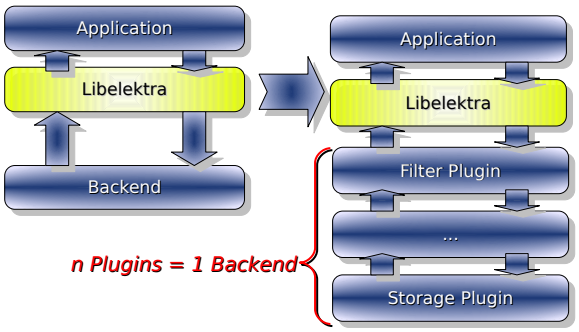

Figure 2.1: Introduction of Multiple Plugins

Each plugin implements a single concrete requirement and it does that well[30]. This architecture allows plugins to have external dependence. Not every plugin has the burden to be portable anymore. That is no problem because the plugins are separate subprojects and maintainers can decide if they should be built for a specific platform or not. Afterwards users can choose which plugins they want to install and use. And finally, the administrator can choose which of the plugins should be loaded for each backend. If a specific feature is not needed, it is not included and does not cause additional overhead. Given the chosen approach, the core of Elektra can stay minimal. In addition, even parts of the former core library[9] can be moved to plugins.

Now let us look at the *development time* with multiple plugins. The programmer will find many reusable plugins. Some of them already fulfil given requirements. While checking the code quality of the plugins, the programmer actually learns how the plugin works. Given that point of view, the programmer will decide to use Elektra because he or she can choose from a pool of existing plugins.

2.2.1 Metadata

Introduction

METADATA is data about data. Up to now, there has been a limited number of metadata entries suited for `filesys`. For `filesys` this was efficient, but it was of limited use for every other backend. This situation has now changed fundamentally by introducing arbitrary metadata.

In Elektra, METAKEYS represent metadata. They can be stored in a key database, often within the representation of the `Key`. Metakey is a `Key` that is part of the data structure `Key`. It can be looked up using a METANAME. Metanames are unique within a `Key`. Metakeys can also carry a value, called METAVALUE. It is possible to iterate over all metakeys of a `Key`, but it is impossible for a metakey to hold other metakeys recursively. The purpose of metakeys next to keys is to distinguish between configuration

[9] For example, the parts implemented by `libhelper`.

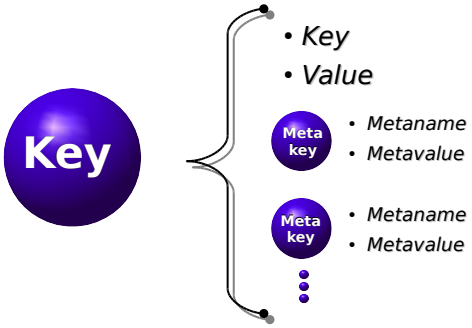

Figure 2.2: Metakeys

and information about settings[9].

Rationale

Metadata has different purposes:

1. Traditionally Elektra used metadata to carry file system semantics. The backend `filesys` stores file metadata[10] into the `Key` objects. This solution, however, only makes sense when each file shelters only one `Key` object.

2. The metaname `binary` shows if a `Key` object contains binary data. Otherwise it has a null-terminated C string.

3. An application can set and get a flag in `Key` objects.

4. Comments and owner, together with the items above, were the only metadata possible before arbitrary metadata was introduced.

5. Further metadata can hold information on how to check and validate keys using types or regular expressions. Additional constraints concerning the validity of values can be convenient. Maximum length, forbidden characters and a specified range are examples of further constraints.

6. They can denote when the value has changed or can be informal comments about the content or the character set being used.

7. They can express the information the user has about the key, for example, comments in different languages. Language specific information can be supported by simply adding a unique language code to the metaname.

[10]File metadata in POSIX is returned by `stat()` in a **`struct`** with the same name. It contains a file type (directory, symbolic link,..) as well as other metadata like uid, gid, owner, mode, atime, mtime and ctime.

8. They can represent information to be used by storage plugins. Information can be stored as syntactic, semantic or additional information rather than text in the key database. This could be ordering or version information.

9. They can be interpreted by plugins, which is the most important purpose of metadata. Nearly all kinds of metadata mentioned above can belong to this category.

10. Metadata is used to pass error or warning information from plugins to the application[11] as we will see in Section 3.3.2 on page 48. The information is uniquely identified by numerical codes. Metadata can also embed descriptive text specifying a reason for the error.

11. Applications can remember something about keys in metadata. Such metadata generalises the application-defined flag.

12. A more advanced idea is to use metadata to generate forms in a programmatic way. While it is certainly possible to store the necessary expressive metadata, it is plenty of work to define the semantics needed to do that.

2.2.2 Contracts

Each plugin in a backend can cause run time errors. Additionally, the chaining of the plugins can introduce further run time errors. For example, a plugin can modify keys so that the next plugin cannot process these keys anymore. Or a plugin can omit changes to the keys that are required by the next plugin. To deal with such situations in a controlled way, each plugin exports a *contract* that describes the interaction with other parts of the backend. A `KeySet` contains the description.

Mounting of backends actually takes place at run time, we will refer to it as MOUNT TIME. The time when applications access the key database, however, will be called RUN TIME.

The CONTRACT CHECKER revises contracts of plugins during the mount time. Afterwards, at run time, no such type errors can occur. `kdb mount` implements the contract checker. It can refuse to add a plugin to the backend because of a conflict or constraint. As long as not all contracts are satisfied `kdb mount` waits for more plugins to be attached.

The contract contains well-defined CLAUSES, but has no hidden clauses[31]. Some of them are introduced in the following text. The other clauses will be introduced later when it is appropriate. Use the index at the end of this thesis to find all clauses.

infos/provides introduces names for an abstract action that is implemented by this plugin. A plugin indicates that it is a *provider* for all activities necessary to fulfil a specific mission. The name already provides users with an understanding what this assignment is. Together with an informal text (`infos/description`) it exactly describes the responsibilities. Other plugins can utilise this service. Some of these services will be presented in Chapter 4.

[11] The application can decide to present it to the user.

infos/needs is the counterpart of provides. Using it, plugins delegate some work to other plugins. They need a provider of a specific service to work properly. The clause describes that the plugin can only fulfil its work if a specific service is present in the backend. It does not state when the work has to be done. The clause only declares that such a provider must be present anywhere in the backend. The name can also directly refer to another plugin's name.

infos/version gives the number of the version of Elektra's plugin interface the plugin is written for. The version implies how the plugin interface must look. For the contracts described in this paper, it is always "1".

infos/author blames the person responsible for the plugin. The plugin must behave as described in the contract. This clause should contain an e-mail address so that direct contact is possible.

infos/licence contains the plugin licence. Note that even if the plugin's code is BSD licenced[12] and it needs code, for example, under GPL[13] to work, the contract needs to say "GPL". "BSD" stands for the simplified, three-clause BSD licence that is used by Elektra itself. For any plugin which does not declare the licence to be "BSD" currently, a warning will be displayed during the mount process, because the overall licence of the installation concerned may change.

infos/description explains informally what the plugin does. It should be humanly readable and will not be parsed or checked. The person reading it should get an idea if, and why, this plugin should be used. This clause is important for reusability, but negligible for the contract's validity.

exports provides function names as subkeys. A plugin can export symbols for external use. Users of the library can look for these symbols and get a pointer to a function. Other plugins, however, are not supposed to use these symbols.

exports/name is such a symbol. *Name* can be any valid C name. It contains a function pointer to enable applications to call the function. Given the plugin and symbol's name the function pointer can be accessed by a unique locator in the global key database.

config/needs presents the plugin configuration needed so that the plugin works properly. Typically the configuration is not for the plugin itself, but for other needed plugins (`infos/needs`).

Plugins are the only place to export contracts, but they are not the only party. It is also possible that the administrator extends the requirements to every backend, for example, if notification is required. This implies that every backend has to provide a specific additional service that will be checked using contracts. Plugins are also involved in contracts with Elektra's core. This topic will be discussed in Section 3.4 on page 52 in which the algorithm used by Elektra's core is explained.

[12] Berkeley Software Distribution
[13] GNU General Public License

Assertions

Contracts as introduced by [31] define PRECONDITIONS and POSTCONDITIONS on routines. Assertions handle these conditions by checking them before entering or after leaving a routine. Because most of this can happen only at run time[25, 18, 31], it often leaves the user alone with an exception.

Elektra goes a step further. Instead of exiting the normal control flow when a precondition is not met, it is the responsibility of a special plugin to handle the situation and make sure that the precondition is met afterwards. Sometimes this is not possible. In these cases, the plugins check the necessary conditions and return with error code when they are not met. As we will see, these situations do not occur very often.

Because of the modular approach, we can have several checkers, and correcting and checking can be combined. Plugins can work together to reach a certain goal.

`KeySet` and `Key` already handle most parts of checking pre- and postconditions imposed on data structures. Elektra's core provides preconditions and weak postconditions for the plugins.

2.2.3 Consequences

We will discuss here the consequences of the new approach by comparing the previous situation in Elektra 0.6 with *multiple plugins*, *contracts* and *metadata* adopted in Elektra $0.8mile4$.

Elektra Semantics

The use of arbitrary metadata has extensive effects in Elektra's semantics. They become simpler and more suited to carry key value pairs. The semantics now gives us independence of the underlying file system. So none of the file system's restrictions apply anymore. No constraints on the length of a key name disturbs the user any more. Additionally, key names can be arbitrarily deep nested[14].

The directory concept does not exist anymore. Keys can be created everywhere. Keys can always have a value. The only constraint is that they are unique and occur either in the user or system hierarchy. Every Key has an absolute name. There is no concept of relative names in Elektra except for metadata belonging to a key. Every Key is independent of each other. We just do not care if there is another key below or above the accessed one in the storage or not.

Some applications need specific structure in the keys. Plugins can introduce and enforce relationships between keys. They can implement a type system, check if holes are present and check the structure and interrelations. They may propagate the metadata and introduce inheritance. We see that plugins are able to add more semantics to Elektra.

There are no symbolic links, hard links, device files or anything else different from key value pairs. Again, most of these behaviours can be mimicked using metadata.

Hidden keys are not useful for Elektra. Instead comments or other metadata contain information about keys that is not considered to belong to the configuration. If hidden keys are desired, we can still write a plugin to filter specific keys out.

[14] Depth is the number of `/` in the key name.

Elektrifying Software

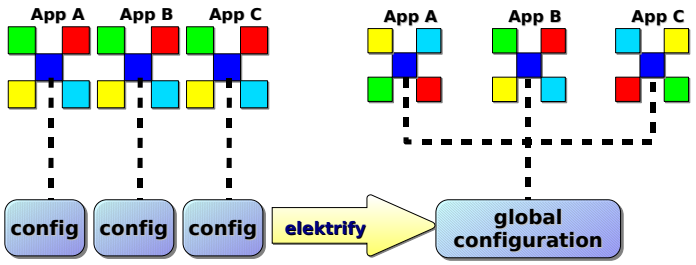

Figure 2.3: Elektrify Software, thanks to Avi Alkalay[1]

When a software starts to use Elektra, it is called to be *elektrified*. The first steps are basically the same in the current version as in Elektra 0.7. The places where configuration is parsed or generated must be located. Afterwards, Elektra's data structures must be used instead at these locations. In Elektra 0.7, that was nearly everything the software developer could do.

Now more optional possibilities are open. First, the parser and generator code can be moved to a storage plugin without having to rewrite additional parts. Doing this, the syntax of the configuration file stays exactly the same as it was before. The application immediately profits from Elektra's infrastructure solving basic issues like update and conflict detection and resolving of the file name.

Moreover, a huge variety of plugins can be utilised to extend the functionality of the configuration system and the programmer can write supplementary plugins. They can do syntactic checks, write out events in their log files and notify users if configuration has changed. So we get a bunch of plugins that are reusable, and we gain a lot because every other program can also access the configuration of this software.

For new software the situation is similar. Additionally, there is the option to reuse mature and well-tested plugins to achieve the optimal result for configuration. New applications simply do not have the burden to stay compatible with the configuration system they had to before. But they can also use self-written plugins for adding needed behaviour or cross-cutting concerns.

To sum up, if a developer wants to *elektrify* software, he or she can do that without any need for changes to the outside world regarding the format and semantics of the configuration. But in the inter-connected world it is only a matter of time until other software also wants to access the configuration, and with elektrified software it is possible for every application to do so.

On Purging Applications

There are two ways of uninstalling software: REMOVING and PURGING. After removing, the software can be reinstalled and it will work as before. If software is purged, its system configuration is also gone. But user configuration of the application will stay. Elektra, however, allows users to purge their configuration of applications, that are not installed, or if the user does not need the settings anymore. One way to achieve this is based on a central repository containing a list of key names for every program. This central repository only moves the problem to another place: What is responsible for removing entries from there?

Another way to solve this problem is to tag keys with metadata to which software they belong. It still needs a repository of installed software. But after the item in the repository has been purged, the user's configuration with the metadata referencing to nothing persists. Because the reference is gone one knows that it belongs to software that is not installed at that moment.

Security Concerns

The tight integration of software also includes some risks. Software may locate more places where it can plant hidden configuration – in the configuration of other software. We believe, however, that with further developed techniques, as described in the subsection above[15], and especially with revision-control systems it is easy to detect such misusage.

Additionally, potentially sensitive information like proxy settings, preferred e-mail servers and similar information may be abused by malicious software. This can easily be avoided because the file restrictions of the configuration files still apply and cannot be circumvented with Elektra. Untrusted software must be executed by another user without permission on the sensitive files.

On the other hand, security can even be improved with Elektra. For example, Elektra makes it possible to split configuration in parts of fine granularity. Each of these configuration files can be restricted separately. In Elektra 0.7 `filesys` illustrates an extreme variant of what is possible. Because every key was in its own file it even allowed users to change their passwords without giving `passwd`[16] extended privileges by giving specific users write-only-access to their key storing the password.

Universal Plugins

When the backend architecture was introduced nearly all code was simply moved to a backend. Elektra's core was only a delegator. It mainly consisted of some trivial loops in the case that the plugin supported only one `Key` and not a whole `KeySet`.

Only one plugin dedicated to be externally called was present then: `libelektratools`. It was used to import and export configuration from and to XML. But it could not be used as a backend because of its incompatible API although an XML backend was a common request.

[15] Using cryptography so that it cannot be faked.
[16] The program that changes passwords in UNIX.

This situation changed with the introduction of contracts. Now UNIVERSAL PLUGINS can export function pointers using the clause `export` as described in Section 2.2.2 on page 21. Plugins can provide, next to their actual task, functions for applications.

The main purpose is explained by the import and export feature that can now be provided by every storage plugin. One only needs to import a `KeySet` using one plugin and export it using another plugin. Some filters in between also become handy. So `libelektratools` is superseded by this new way to access functionality of regular plugins. To sum up, plugins are now universal to applications because they can export arbitrary functions.

2.3 Choices

We established the basic approach and looked at its direct consequences. Some consequences, however, yield problems. In this section we look at some problems, give some ideas how to address them and decide how we will solve them. Additionally, we will give a rationale for the decision where appropriate.

2.3.1 Storage Plugins

The storage plugin needs a significant part of the development time compared with other plugins of a backend. Many approaches exist to reduce the needed effort significantly:

Grammar: Writing the parser by hand may lead to faster parsing times in some cases. But the resulting code is often hard to maintain. Various ways exist to create storage plugins using grammars as the next two paragraphs describe.

Lex (Flex) and Yacc (Bison) are powerful tools which make it easy to parse even complex files and fill up data structures during the process. They are not intended to generate configuration files.

The C++ boost libraries contribute with `spirit::qi` and `spirit::karma` a full solution to parse and even generate files using a grammar. They are implemented with C++ templates forming a domain-specific language.

Configuration Libraries: Another idea is to reuse libraries that contain already written parsers or generators.

Copy Existing Plugins: Some programmers prefer to start from a similar plugin when they intend to implement another one. Large plugins with many features are not suitable to do so. Instead, they rather confuse programmers and show that writing such a plugin is a really hard job. Storage plugins of the new generation do nothing else than deal with storage and are good candidates as starting points.

Reuse: The most important aspect is that sometimes a plugin needs a feature exactly as it is implemented in another plugin. Filter plugins often help in these situations. The support of reuse of plugins can improve development time significantly.

We decided to implement and compare some of these variants because it is not clear which ones will yield the best results in term of development time, features and robustness without concrete data. The results are given in Section 5.1 on page 83.

2.3.2 Resolver Plugins

With the introduction of multiple plugins, we have the possibility to write portable, as well as, operating system specific plugins interacting within the same backend. While many plugins are designed to be portable, the so-called *resolver plugin* is not.

The plugin takes over the responsibility for resolving the configuration's file name depending on the key name[17], the environment and other operating system specific properties. In addition, it overwrites this file in an atomic way. Finally, it guarantees that the configuration is only updated if needed. In UNIX, these operations use the same operating system facilities and need to remember the same information.

The resolver compares the timestamp or revision of the file with the one it had in the previous attempt. On *reading configuration*, a newer key database just means that an update is needed. But on *writing configuration*, a more recent key database means that a conflict has occurred.

The resolver can lock and rename files to accomplish an overall atomic property of `kdbSet()`. The resolver must always get a chance to undo its actions. If an error has occurred, the resolver has to do a rollback. Otherwise it can commit the changes.

Reverse Resolving

We already learned that it is the job of the resolver to find the appropriate file name. While many approaches exist to solve this problem, each of them is simple. The reverse question, however, is more difficult. We want to know which key names a specific configuration files holds. An approach to answer the question is to enrich every `Key` with the filename as metadata. The resolver could be asked to do that. Then it would be possible to search for keys within a given configuration file. But that would be an expensive operation because the whole configuration tree needs to be traversed.

Rationale

The idea of extracting the operating system-dependent parts from the storage plugins opens many possibilities. The most important one is to add the support of another operating system by writing a new resolver plugin. Moreover, different strategies can be used on a single operating system depending on the user's requirements. This approach allows the resolving to behave differently to fit the user's needs even better.

[17] If the key starts with `system` the path is usually known at compile time, but if the key starts with `user` the path is usually the home directory of the current user.

2.3.3 Cross-Cutting Concerns

This section describes cross-cutting concerns of applications that plugins can solve. Because they concentrate on a sole purpose, specialised plugins are possible.

Notification

Today, programs are often interconnected in a dense way. Such applications should always be informed when something in their environment changes. For user interactive software, notification about configuration changes is expected. The only alternative is polling, which wastes resources. It additionally is no option, because for interactive software the latency needs to be low. Instead, the software which changes the configuration has to notify all other interested applications that can reread their configuration without significant delay. In Elektra, a notification plugin ensures that a notification is actually sent on each change.

Applications can wait for such a notification with hand-written code. Bindings, however, allow for better integration. It is a common approach for toolkits to provide a main loop. Applications using such toolkits can integrate notification services into this main loop.

The actions that occur in such events are application or toolkit specific because of the non-invasive nature of Elektra. Software reacts in many different ways to update events. Hence, the frequency of update events should be kept at a minimum. Changes are kept atomic with a single attempt to write out configuration. Notification callbacks shall not change configuration because this can lead to a longer chain of unwanted modifications. That might not be true, however, if a programmer of the whole system knows that a chain of reactions will terminate. When doing such *event-driven programming*, care is needed to avoid infinite loops. Elektra guarantees consistency of the key database even in such cases.

The preferred way to interconnect desktop applications and even embedded system applications on mobile devices running Linux is D-BUS[27]. The idea of D-Bus accords to that of Elektra: to provide standards to let software work together more tightly. D-Bus provides a simple and lightweight IPC[18] system to be used within desktop systems. Next to RPC[19], which is of no interest in this work, it supports signals which can notify an arbitrary number of other applications about changes. Given software like a D-Bus library, notification itself is a rather easy task, but it involves additional library dependences. So it is the perfect task to be implemented as a plugin. The information about the channels to be used can be stored in the global key database.

D-Bus supports a *system-wide bus* and a *session bus*. The system configuration can be accessed by each user and the user configuration is limited to a single user. Both buses can immediately be used for the system and user configuration notification updates to get pleasing results. But, there is a problem with the session bus: It is possible within D-Bus that a user starts several sessions. The user configuration should be global to the user and is not aware of these sessions. So if several sessions are started, some of the user's processes will miss notification updates.

[18] Inter-Process Communication
[19] Remote Procedure Call

Other OS or platforms provide other communication channels. Additional plugins are needed to support them.

Forensic Logging

Up to now, we have only discussed problems of configurations on a single system. Even though Elektra provides a global key database on a stand-alone computer, some plugins can help regarding computer network issues.

Elektra can provide *forensic logging* through a plugin. It can enable full information about[19]:

- which keys are affected
- what exactly was changed
- who caused the change
- on which computer in sync with its environment the change was initiated
- exact local timestamp in milliseconds together with time zone information
- the process doing so

To ensure that the log is not faked it should be written on a remote computer and be secured there.

Security Policies

Elektra can provide powerful tools to enforce a security policy by mounting backends into the user's key databases. Only the administrator controls the mounting of backends. The user is not able to change that mount configuration.

A common requirement is to enforce a specific proxy in a company. Bypassing the proxy poses a severe security concern for that company. The administrator can force the use of a proxy by mounting a special plugin which rejects proxies not allowed to be used according to company rules. If the user has no choice at all, a plugin that returns an unchangeable key set can be mounted. These possibilities give the administrator full control over the configuration.

KIOSK MODE usually denotes a group of actions with the purpose of reducing software functionality in order to deploy it safely on interactive kiosk systems. Most of the desired effects can be achieved by forcing the configuration to have specific values and do not give the user any chance to change them. The goal is that the administrator can do that on an option basis. An administrator using kiosk mode wants to have full control over values allowed to occur in settings.

Elektra can help achieve that in several ways. First, it is possible than an application loads system configuration only. But this restriction needs severe modifications to the software because configuration remains a cross-cutting concern. So this is only an option if it is the usual case to ignore user configuration as for user credentials databases.

A special kiosk plugin can solve the problem much better. It is mounted for user configuration only, and changes the values as requested by the administrator. It also sets a meta flag so that it is known that this key is forced because of the kiosk mode. If the application is kiosk aware, it could deactivate the options in the settings dialogue. Even if the setting is changed, the plugin will always return the same value for a specific user as defined by the administrator. To disable the user hierarchy or parts of it, the administrator can mount a plugin returning no configuration.

2.3.4 Ordering

Multiple plugins open up many ways in which they can be arranged in a backend. A simple way is to have one array with pointers to plugins as shown in Figure 2.4. To store a `KeySet`, the first plugin starts off with the `KeySet` passed to it. The resulting `KeySet` is given to the next plugin. This is repeated for every plugin in the array.

To obtain a `KeySet`, the array of plugins is processed the other way round. An empty `KeySet` is passed to the last plugin. The resulting `KeySet` is handed over to the previous one until the first plugin is executed.

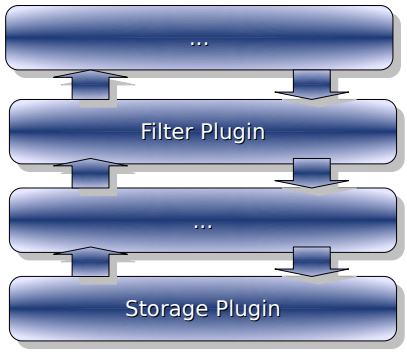

Figure 2.4: Ordering using Stacking

This approach has shown not to be powerful enough to express all use cases by counter-evidence. Logging should take place after the storage plugin performs its actions in both directions. It is not possible to do this with a *single array* processed in a way as described above.

Separated Lists

Two individual lists of plugins solve the problem as shown in Figure 2.5. Instead of bidirectional processing of a single list two separate arrays are needed. It turns out that error scenarios also make this approach unsuccessful. When a plugin fails, no other plugin must be executed later on because it might depend on the previous plugin to work correctly. In `kdbGet()`, this works well – the update process will be stopped. But during `kdbSet()`, changes to the file system must be reverted. Plugins can leave a lock, temporary file or journal. These resources need to be cleaned up properly.

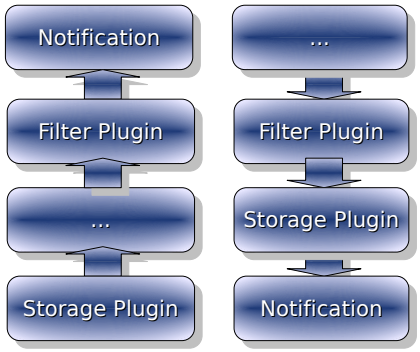

Figure 2.5: Ordering using Separated Lists

Error List

So every backend additionally needs a third array that is executed in error scenarios during `kdbSet()`. Figure 2.6 shows the situation. Plugins responsible for the cleanup, rollback or error notification are inserted into it.

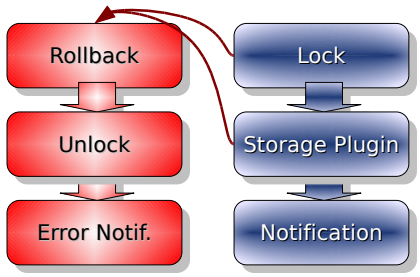

Figure 2.6: Ordering with Error List

The resolver plugin requires this error list to do a proper rollback. Another use case is logging after a failure has happened.

Contracts

The ordering of plugins inside these three arrays is controlled by contracts. Each of the three arrays has ten slots. These slots have names to be referred to in the contract. Table 2.1 contains all names.

The following *clauses* determine the placement:

infos/placement gives a list of places in which the plugin must appear. The places are referred to with the names given in Table 2.1. During mounting, the algorithm checks if a free slot is available in the requested position. Because of this clause the placement works automatically.

Table 2.1: Placement of Plugins

	error	get	set
0	prerollback	getresolver	setresolver
1	prerollback	pregetstorage	presetstorage
2	prerollback	pregetstorage	presetstorage
3	prerollback	pregetstorage	presetstorage
4	prerollback	pregetstorage	presetstorage
5	rollback	getstorage	setstorage
6	postrollback	postgetstorage	precommit
7	postrollback	postgetstorage	commit
8	postrollback	postgetstorage	postcommit
9	postrollback	postgetstorage	postcommit

infos/ordering requests that a list of plugins or provided names is not present at the time of insertion. If such a plugin is already there, the order constraint is violated. Note, that the relation to the storage and resolver plugins is already determined using the clause placement.

infos/stacking disables the stacking of plugins. By default, plugins in `postgetstorage` are ordered in reverse order than in `presetstorage`. This is called *stacking*. The stacking reintroduces the feature which would automatically be available when only one array of plugins is processed bidirectionally as shown in Figure 2.4.

CHAPTER 3

Implementation

3.1 Architecture

In this chapter we will explain the implementation of the modular approach. We discuss problems and the solution space so that the reader can understand the rationale of how problems were solved.

To help readers to understand the algorithm that glues together the plugins, we first describe some details of the data structures. Full knowledge of the algorithm is not presumed to be able to develop plugins.

3.1.1 API

The aim of the Elektra Project is to design and implement a powerful API for configuration. When the project started, we assumed that this goal was easy to achieve, but dealing with the semantics turned out to be a difficult problem. For the implementation, an ambitious solution is required because of the necessary modularity to implement flexible backends as introduced in this thesis. But also the design of a good API has proved to be much more difficult than expected.

Changes in the API

For this thesis from Elektra 0.7 to Elektra $0.8mile4$, we changed the API of Elektra as little as possible. It should be mentioned that `KeySet` is now always sorted by name. The function `ksSort()` is now depreciated and was removed. The handling of removed keys was modified. Additionally, the API for metadata has fundamentally changed, but the old interface still works. These changes will be described in Section 3.2.2 on page 41. On the other hand, the implementation of Elektra changed radically as discussed in Section 3.4 on page 52.

API Design

API Design presents a critical craft every programmer should be aware of [7, 44]. We will shortly present some of the main design issues that matter and show how Elektra has solved them.

A design goal is to detect errors early. As easy as it sounds, as difficult it is to actually achieve this goal. Elektra tries to avoid the problem by checking data being inserted into `Key` and `KeySet`. Elektra catches many errors like invalid key names soon. As we will see in Section 4.4 on page 77, Elektra allows plugins to check the configuration before it is written into the key database so that problematic values are never stored.

”Hard to use it wrong” tends to be a more important design objective than ”Easy to use it right”. Searching for a stupid bug costs more time than falling into some standard traps which are explained in documentation. In Elektra, the data structures are robust and some efforts were taken to make misuse unlikely.

Another fundamental principle is that the API must hide implementation details and should not be optimised towards speed. In Elektra, the actual process of making configuration permanent is completely hidden.

”Off-by-one confusion” is a topic of its own. The best is to stick to the conventions the programming language gives. For returning sizes of strings, it must be clear whether a terminating `'\0'` is included or not. All such decisions must be consistent. In Elektra the terminating null is always included in the size.

The interface must be as small as possible to tackle problems addressed by the library. Internal and external APIs must be separated. Internal APIs in libraries shall be declared as **`static`** to prevent its export. In Elektra, internal names start with `elektra` opposed to the external names starting with `key`, `ks` or `kdb`.

Elektra always passes user context pointers, but never passes or receives a full data structure by value. It is impossible to be ABI[1] compatible otherwise. Elektra is restrictive in what it returns (strong postconditions), but as liberal as possible for what comes in (preconditions are avoided where possible). In Elektra even null pointers are accepted for any argument.

”Free everything you allocate” is a difficult topic in some cases. If Elektra cannot free space or other resources after every call, it provides a `close()` function. Everything will be freed[2] when deleting all created `Key` and `KeySet` objects and closing the `KDB` handle.

We always provide a default branch in the unlikely case – it might become possible some day. As a final statement, we note that the UNIX philosophy[30] should always be considered: ”Do only one thing, but do it in the best way. Write it that way that programs work together well.”

[1] We will read more about ABI in Section 3.2.1 on page 41.

[2] The tool *Valgrind* with *Memcheck* helps us locate problems. The whole test suite runs without any memory problems.

3.1.2 Concepts

Default Backend

One important aspect of a configuration library is the out-of-the-box experience. How does the system work before anything is configured? The optimal situation is that everything works fully, and applications, that just want to load and store configuration, do not see any difference in a well-configured, fine-tuned system.

To support that experience, a so-called DEFAULT BACKEND is responsible in the case that nothing was configured so far. It must have a storage that is able to store full Elektra semantics including every type and arbitrary metadata. To avoid reimplementation of storage plugins, for default storage plugins a resolver plugin additionally takes care of the inevitable portability issues.

The default backend is guaranteed to stay mounted at `system/elektra` where the configuration for Elektra itself is stored. After mounting all backends, Elektra checks if `system/elektra` still resides at the default backend. If not, it will be mounted there.

To summarise, this approach delivers a good out-of-the-box experience capable of storing configuration. For special use cases, applications and administrators can mount specific backends anywhere except at, and below, `system/elektra`. On `kdbOpen()`, the system BOOTSTRAPS itself starting with the default backend.

The default backend consists of a default storage plugin and default resolver plugin. The default resolver has no specific requirements, but the default storage plugin must be able to handle full Elektra semantics. The backend is not mounted anywhere in particular, so any keys can be stored in it. The implementation of the core guarantees that user and system keys always stay separated.

Granularity

Keys of a backend can only be retrieved as a full key set. Currently it is not possible to fetch a part of the keys of a backend. So the user needs to cut out the interesting keys with `ksCut()` afterwards. If the keys should be committed again, the whole key set must be preserved. Otherwise, the clipped keys will be removed permanently.

This restriction simplifies storage plugins while it does not limit the user. Other plugins would not be able to fulfil their purpose without the full `KeySet`. For example, a plugin that checks the availability and structure of all keys cannot work with a partial key set.

It is problematic to have too many keys in one backend. The applications would need memory for unnecessary configuration data. Instead we recommend introducing several mount points to split up the keys into different backends. Splitting up key sets makes sense if any application requests only a part of the configuration. No benefits arise if every application requests all keys anyway.

Let us assume that many keys reside in `user/sw` and an application only needs the keys in `user/sw/app`. To save memory and get better startup times for the application, a new backend can be mounted at `user/sw/app`. On the other hand, every mounted backend causes a small run time overhead in the overall configuration system.

The solution in Elektra is flexible, because the user decides the granularity. It is possible to mount a backend on every single key, so that every key can be requested for itself. If no backends are mounted, all keys reside in the default backend.

To sum up, Elektra's core searches for the nearest mount point and gets the configuration from there. It is possible that the user gets more configuration than requested. The user can decide by means of mounting how much configuration on specific requests are returned.

Sync Flag

Elektra keeps track of changes the user makes on data structures. Because the `KeySet` can be composed in the way the user wants it, the `KeySet` does not give enough information if changes occurred within. Instead, changes will be marked in individual `Key` objects. On modifying functions, the so-called SYNC FLAG will be set.

3.1.3 Modules

Elektra's core can be compiled with a C compiler conforming to the ISO/IEC 9899:1999 standard[3], called C99 in the following text. Functions not conforming to C99 are considered to be not portable enough for Elektra and are separated into plugins. But there is one notable exception: it must be the core's task to load plugins. Unfortunately, C99 does not know anything about *modules*. POSIX[4] provides `dlopen()`, but other operating systems have dissimilar APIs for that purpose. They sometimes behave differently, use other names for the libraries and have incompatible error reporting systems. Because of these requirements Elektra provides a small internal API to load such modules independently from the operating system. This API also hides the fact that modules must be loaded dynamically if they are not available statically.

Plugins are usually realised with modules. Modules and libraries are technically the same in most systems[5]. After the module is loaded, the special function PLUGIN FACTORY is searched for. This function returns a new plugin. With the plugin factory the actual plugins are created.

Static loading

For the static loading of modules, the modules must be built-in. With `dlopen(const char* file)` POSIX provides a solution to look up such symbols by passing a null pointer for the parameter `file`. Non-POSIX operating systems may not support this kind of static loading. Therefore, Elektra provides a C99 conforming solution for that problem: a data structure stores the pointers to the plugin factory of every plugin. The build system generates the source file of this data structure because it depends on built-in plugins as shown in Figure 3.1.

[3] One line comments, inline functions, `snprintf()`, `inttypes.h` and variable declaration at any place are used in addition to what is already defined in the standard ISO/IEC 9899:1990.

[4] Portable Operating System Interface

[5] One exception is Mac OS X.

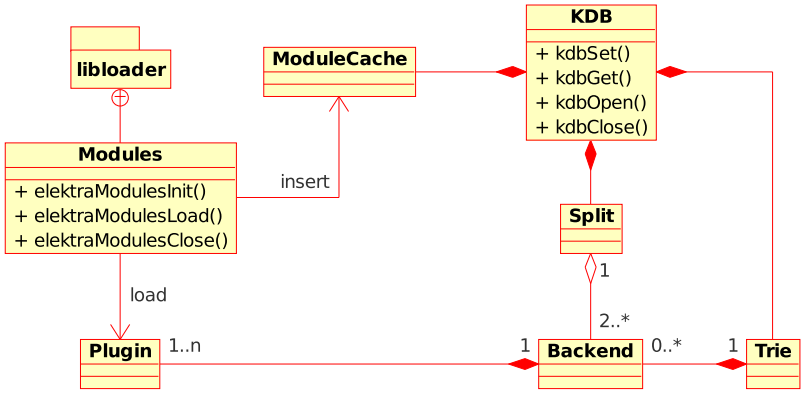

Figure 3.1: Architecture

Elektra distinguishes internally between modules and plugins. Several plugins can be created out of a single module. During the creation process of the plugin, dynamic information – like the configuration or the data handle – is added.

API

The API of LIBLOADER consists of the following functions:

Listing 3.1: Interface of Module System

```
int elektraModulesInit (KeySet *modules, Key *error);
elektraPluginFactory elektraModulesLoad (KeySet *modules,
        const char *name, Key *error);
int elektraModulesClose (KeySet *modules, Key *error);
```

`elektraModulesInit()` initialises the module cache and calls necessary operating system facilities if needed. `elektraModulesLoad()` does the main work by either returning a pointer to the plugin factory from cache or loading it from the operating system. The plugin factory creates plugins that do not have references to the module anymore. `elektraModulesClose()` cleans up the cache and finalises all connections with the operating system.

Not every plugin is loaded by `libloader`. For example, the *version plugin*, which exports version information, is implemented internally.

3.1.4 Mount Point Configuration

`kdb mount` creates a MOUNT POINT CONFIGURATION as shown in Listing 3.2. `fstab` is a unique name within the mount point configuration provided by the administrator.

Listing 3.2: Example for a mount point configuration

```
system/elektra/mountpoints
system/elektra/mountpoints/fstab
system/elektra/mountpoints/fstab/config
system/elektra/mountpoints/fstab/config/path=fstab
system/elektra/mountpoints/fstab/config/struct=list FStab
system/elektra/mountpoints/fstab/config/struct/FStab
system/elektra/mountpoints/fstab/config/struct/FStab/device
system/elektra/mountpoints/fstab/config/struct/FStab/dumpfreq
system/elektra/mountpoints/fstab/config/struct/FStab/mpoint
system/elektra/mountpoints/fstab/config/struct/FStab/options
system/elektra/mountpoints/fstab/config/struct/FStab/passno
system/elektra/mountpoints/fstab/config/struct/FStab/type
system/elektra/mountpoints/fstab/errorplugins
system/elektra/mountpoints/fstab/errorplugins/#5#resolver#resolver#
system/elektra/mountpoints/fstab/getplugins
system/elektra/mountpoints/fstab/getplugins/#0#resolver
system/elektra/mountpoints/fstab/getplugins/#5#fstab#fstab#
system/elektra/mountpoints/fstab/mountpoint /fstab
system/elektra/mountpoints/fstab/setplugins
system/elektra/mountpoints/fstab/setplugins/#0#resolver
system/elektra/mountpoints/fstab/setplugins/#1#struct#struct#
system/elektra/mountpoints/fstab/setplugins/#2#type#type#
system/elektra/mountpoints/fstab/setplugins/#3#path#path#
system/elektra/mountpoints/fstab/setplugins/#3#path#path#/config
system/elektra/mountpoints/fstab/setplugins/#3#path#path#/config/path/allow=proc tmpfs none
system/elektra/mountpoints/fstab/setplugins/#5#fstab
system/elektra/mountpoints/fstab/setplugins/#7#resolver
```

Let us look at the subkeys below the key `system/elektra/mountpoints/fstab`:

config Everything below `config` is the system's configuration of the backend. Every plugin within the backend will find this configuration directly below `system/` in its PLUGIN CONFIGURATION. For example,

`system/elektra/mountpoints/fstab/config/struct/FStab/mpoint`

will be translated to

`system/struct/FStab/mpoint`

and inserted into the plugin configuration for all plugins in the `fstab` backend.

It is the place where configuration can be provided for every plugin of a backend. The contract checker deduces this configuration to satisfy the contract for a plugin. Fstab, for example, claims in a contract that it needs ”struct”. But the struct plugin needs a configuration to work properly. Fstab will provide this configuration. The *contract checker* writes out the configuration looking like the one in this example.

config/path is a common setting needed by the resolver plugin. It is the relative path to a filename that is used by this backend. On UNIX systems, the resolver would determine the name `/etc/fstab` for system configuration.

mountpoint is a key that represents the mount point. Its value is the location where the backend is mounted. If a mount point has an entry for both the user and the system hierarchy, it is called CASCADING MOUNT POINT. A cascading mount point differs from two separate mount points because internally only one backend is created. In the example, the mount point `/fstab` means that the backend handles both `user/fstab` and `system/fstab`. If the mount point is `/`, the backend will be mounted at both `user` and `system`.

errorplugins presents a list of all plugins to be executed in the error case of `kdbSet()` which will be explained in Section 3.3.3 on page 52.

getplugins is a list of all plugins used when reading the configuration from the key database. They are executed in `kdbGet()`.

setplugins contains a list of all plugins used when storing configuration. They are executed in `kdbSet()`.

Each of the plugins inside the three lists may have the subkey `config`. The configuration below this subkey provides plugin specific configuration. This configuration appears in the user's configuration of the plugin. Configuration is renamed properly. For example, the key
`system/elektra/mountpoints/fstab/setplugins/#3#path#path#/config/path/allow`
is transformed to
`user/path/allow`
and appears in the plugin configuration of the path plugin inside the fstab backend.

Referencing

The same plugin often must occur in more than one place within a backend. The most common use case is a plugin that has to be executed for both `kdbGet()` and `kdbSet()`. It must be the same plugin if it preserves state between the executions.

Other plugins additionally have to handle error or success situations. One example of exceptional intensive use is the resolver plugin. It is executed twice in `kdbSet()`. In `kdbGet()` it is also used as shown in Listing 3.2.

`#n<name>` introduces a new plugin from the module `name` which cannot be referenced later. The cypher `n` appoints the actual placement of the plugin. `#n#<name>#<label>#` also introduces a new plugin from the module `name` and gives it the name `label`. The last `#` shows that a new name is being introduced. `#n#<ref>` references back to a label which was introduced before. This configuration does not create a new plugin. `kdb mount` already implements the generation of these names as described above.

Changing Mount Point Configuration

When the user changes the mount point configuration, without countermeasures, applications already started will continue to run with the old configuration. This could lead to a problem if backends in use are changed or removed. It is necessary to restart all such programs. Notification is the best way to deal with the situation. Changes of the mount point configuration, however, do not occur often. For some systems, the manual restart may also be appropriate.

In this situation, applications can receive warning or error information if the configuration files are moved or removed. The most adverse situation occurs if the sequence of locking multiple files produces a *dead lock*. Under normal circumstances, the sequence of locking the files is deterministic, so either all locks can be requested or another program will be served first. But several programs with different mount point configurations running at the same time can cause a disaster. The problem gets even worse, because `kdb mount` is unable to detect such situations. Every specific mount point configuration for itself is trouble-free.

But still a dead lock can arise when multiple programs run with different mount point configurations. Suppose we have a program `A` which uses the backends `B1` and `B2` that requests locks for the files `F1` and `F2`. Then the mount point configuration is changed. The user removes `B1` and introduces `B3`. `B3` is in a different path mounted after `B2`, but also accesses the same file `F1`. The program `B` starts after the mount point configuration is changed. So it uses the backends `B2` and `B3`. If the scheduler decides that first `A` and then `B` both successfully lock the files `F1` and `F2`, a dead lock situation happens because in the afterwards the applications `A` and `B` try to lock `F2` and `F1`.

A manual solution for this problem is to enable `kdb` to output a list of processes that still use old mount point configuration. The administrator can restart these processes. The preferred solution is to use notification for mount point configuration changes or simply to use a lock-free resolver.

3.2 Data Structures

3.2.1 Introduction

Data structures define the common layer in Elektra. They are used to transfer configuration between Elektra and applications, but also between plugins.

ADT

Both the `KeySet` and the interface to metadata within a `Key` are actually *abstract data types*, also known as ADT. The API is designed so that different implementations of the data structures can be used internally. More information on this topic is given in [35, 39].

A HASH presents a good candidate as alternative data structure, especially for the metadata interface. It is believed to be much faster on lookup, but considerably slower on sorted enumeration.

AVL TREES also serve as a competitor. AVL trees are expected to be faster for inserting keys at any place, but may be slower for appending because of the needed reorganisations. Their disadvantage

is that they need to allocate a large number of small pieces of memory. Further investigations, namely implementation and benchmarks, are required to decide.

Currently the `KeySet` is implemented as a sorted array[35]. It is fast on appending and iterating, and has nearly no size-overhead.

ABI compatibility

Application binary interface, or ABI, is the interface to all data structures of an application or library directly allocated or accessed by the user.

Special care has been taken in Elektra to support all changes within the data structures without any ABI changes. ABI changes would entail the recompilation of applications and plugins using Elektra. The functions `keyNew()`, `ksNew()` and `kdbOpen()` allocate the data structures for the applications. The user only gets pointers to them. It is not possible for the user to allocate or access these data structures directly when only using the public header file `<kdb.h>`. The functions `keyDel()`, `ksDel()` and `kdbClose()` free the resources after use. Using the C++ binding deallocation is done automatically.

3.2.2 Metadata

In this section, we discuss the implementation of metadata. Metakey is implemented directly in a `Key` as shown in Figure 2.2. Every metakey belongs to a key *inseparable*. Unlike normal key names there is no absolute path for it in the hierarchy, but a relative one only valid within the key.

The advantage of embedding metadata into a key is that functions can operate on a key's metadata if a key is passed as a parameter. Because of this, `keyNew()` directly supports adding metadata. A key with metadata is self-contained. When the key is passed to a function, the metadata is always passed with it. Because of the tight integration into a `Key`, the metadata does not disturb the user.

A disadvantage of this approach is that storage plugins are more likely to ignore metadata because metakeys are distinct from keys and have to be handled separately. It is not possible to iterate over all keys and their metadata in a single loop. Instead only a nested loop provides full iteration over all keys and metakeys.

The metakey itself is also represented by a `Key`. So the data structure `Key` is nested directly into a `Key`. The reason for this is to make the concept easier for the user who already knows how to work with a `Key`. But even new users need to learn only one interface. During iteration the metakeys, represented through a `Key` object, contain both the metaname and the metavalue. The metaname is shorter than a key name because the name is unique only in the `Key` and not for the whole global configuration.

The implementation adds no significant memory overhead per `Key` if no metadata is used. For embedded systems it is useful to have keys without metadata. Special plugins can help for systems that have very limited memory capacity. Also for systems with enough memory we should consider that adding the first metadata to a key has some additional overhead. In the current implementation a new `KeySet` is allocated in this situation.

Interface

The interface to access metadata consists of the following functions:

Listing 3.3: Interface of metadata

```
const Key *keyGetMeta(const Key *key, const char* metaName);
ssize_t    keySetMeta(Key *key, const char* metaName,
    const char *newMetaString);
```

Inside a `Key`, metadata with a given metaname and a metavalue can be set using `keySetMeta()` and retrieved using `keyGetMeta()`. Iteration over metadata is possible with:

Listing 3.4: Interface for iterating metadata

```
int keyRewindMeta(Key *key);
const Key *keyNextMeta(Key *key);
const Key *keyCurrentMeta(const Key *key);
```

Rewinding and forwarding to the next key works as for the `KeySet`. Programmers used to Elektra will immediately be familiar with the interface. Tiny wrapper functions still support the old metadata interface.

Sharing of Metakey

Usually substantial amounts of metadata are shared between keys. For example, many keys have the type **`int`**. To avoid the problem that every key with this metadata occupies additional space, `keyCopyMeta()` was invented. It copies metadata from one key to another. Only one metakey resides in memory as long as the metadata is not changed with `keySetMeta()`. To copy metadata, the following functions can be used:

Listing 3.5: Interface for copying metadata

```
int keyCopyMeta(Key *dest, const Key *source, const char *metaName);
int keyCopyAllMeta(Key *dest, const Key *source);
```

The name `copy` is used because the information is copied from one key to another. It has the same meaning as in `ksCopy()`. In both cases it is a flat copy. `keyCopyAllMeta()` copies all metadata from one key to another. It is more efficient than a loop with the same effect.

`keyDup()` copies all metadata as expected. Sharing metadata makes no difference from the user's point of view. Whenever a metavalue is changed a new metakey is generated. It does not matter if the old metakey was shared or not. This is the reason why a const pointer is always passed back. The metakey must not be changed because it can be used within another key.

3.2.3 KeySet

The data structure of `KeySet` did not change between Elektra 0.7 and Elektra $0.8mile4$ as explained in [35]. This subsection describes what has changed and deals with some general implementation issues.

Operations

KeySet resembles the classical mathematical set. Operations like union, intersection or difference are well defined. In mathematics typically every operation yields a new set. Instead, we try to reuse sets in the following ways:

1. A completely new and independent KeySet as return value would resemble the mathematical ideal closely. This operation would be expensive. Every Key needs to be duplicated and inserted into a new KeySet.

   Such a DEEP DUPLICATION was only needed in kdbSet() as we will see in Section 3.4.3 on page 58.

2. The resulting KeySet is created during the operation, but only a flat copy is made. This means that the keys in it are actually not duplicated, but only their reference counter is increased. This method is similar to the mathematical model. Compared with a deep copy it can achieve good performance. But all changes to the values of keys in the resulting KeySet affect the original KeySet, too.

   ksDup(**const** KeySet *source) produces a new KeySet that way. The source is not changed as shown by the **const** modifier.

3. The result of the operation is applied to the KeySet passed as argument directly. This is actually quite common, but for this situation other names of the operations are more suitable.

   For example, a union which changes the KeySet is called ksAppend().

4. A new KeySet is created, but the KeySet passed as parameter is reduced by the keys needed for the new KeySet. This is useful in situations where many operations have to be applied in a sequence reducing the given KeySet until no more keys are left. None of the reference pointers changes in this situation.

   ksCut(KeySet *ks, **const** Key *cutpoint) works that way. All keys below the cutpoint are moved from ks to the returned key set.

Consistency

There are several ways to define consistency relations on key sets. For STRICT CONSISTENCY every parent key must exist before the user can append a key to a key set. For example, the key set with the keys

```
system
system/elektra
system/elektra/mountpoints
```

would allow the key system/elektra/mountpoints/tcl to be added, but not the key system/apps/abc because system/apps is missing. File systems provide this kind of consistency.

These semantics are however not useful for configurations. Especially for user configurations often only some keys need to be overwritten. It is not a good idea to copy all parent keys to the users configuration just because some constraint forces us to do so. For this reason we use a less strict definition of consistency supporting such holes.

We also evaluated a form of WEAK CONSISTENCY. It avoids adding some unnecessary keys. A constraint is that a key can be added only if it has a parent key. But the constraint does not apply if no other key exists above the key about to be inserted. From that moment it will serve as parent key for other keys. With the current implementation of `KeySet`, however, it is not possible to decide this constraint in constant time. Instead its worst-case complexity would be $log(n) * x$ where n is the number of keys currently in the key set and x is the *depth* of the key. The depth is the number of `/` in the key name. The worst-case of the complexity applies when the inserting works without a parent key. For example, with the keys

```
user/sw/apps/abc/current/bindings
user/sw/apps/abc/current/bindings/key1
user/sw/apps/abc/current/bindings/key2
```

the weak consistency would allow inserting `user/sw/apps/abc/current/bindings/key3` because it is directly below an existing key. It would also allow adding `user/sw/apps/xyz/current` because it does not have any parent key. But it would not allow `user/sw/apps/abc/current/bindings/dir/key1` to add. The worst-case complexity was found to be too expensive, and hence `KeySet` has NO CONSISTENCY check at all.

This means any key with a valid key name can be inserted into `KeySet`. The `KeySet` is changed so that it is now impossible to append keys without a name. `ksAppendKey(ks, Key *toAppend)` takes ownership of the key `toAppend` and will delete the key in that case. The caller does not have to free `toAppend`: either it is in the key set or it is deleted.

BINARY SEARCH determines the position where to insert a key. The C version of binary search `bsearch()` cannot tell where to insert a key when it is not found. So the algorithm has to be reimplemented. Java's binary search `binarySearch()` uses a trick to both indicate where a key is found and where to insert it with the same return code by returning the negative value $((-insertionpoint) - 1)$ indicating where the new value should be inserted when the key is not found. Elektra now also uses this trick internally.

Internal Cursor

`KeySet` supports an *external iterator*[16] with the two functions `ksRewind()` to go to the beginning and `ksNext()` to advance the *internal cursor* to the next key. This side effect is used to indicate a position for operations on a `KeySet` without any additional parameter. This technique is comfortable to see which key has caused an error after an unsuccessful key database operation.

Elektra only has some functions to change the cursor of a key set. But these allow the user to compose powerful functions. Plugins do that extensively as we will see later in Section 4.4.2 on page 79. The user can additionally write more such functions for his or her own purposes. To change the internal cursor,

it is sufficient to iterate over the `KeySet` and stop at the wanted key. With this technique, we can, for example, realise lookup by value, by specific metadata and by parts of the name. Without an additional index, it is not possible that such operations perform more efficiently than by a linear iteration key by key. For that reason, Elektra's core does not provide such functions. The function `ksLookupByName()`, however, uses the more efficient *binary search* because the array inside the `KeySet` is ordered by name.

External Cursor

External cursor is an alternative to the approach explained above. Elektra provides a limited *external cursor* through the interface `ksGetCursor()` and `ksSetCursor()`. It is not possible to advance this cursor. The difference to the internal cursor is that the position is not stored within `KeySet`.

We considered providing an external cursor for performance reasons. But we found out that the speed of iteration is mainly limited because of safety checks. The investigated methods become slower proportional to the ease of use and safety. When using null pointers and range checks, the function is noticeably slower than without. With the same amount of checks, using an external cursor is not much faster than the `ksNext()` interface[6].

But an external cursor directly accessing the array can be much faster[7]. For this endeavour, Elektra's private header files need to be included. Including private header files, however, should not be done with levity because ABI compatibility will be gone on any changes of the data structures. This fact means the application or plugin needs to be recompiled when any of the internal structures of Elektra are changed. We strongly discourage including these header files.

Nevertheless, the overall performance impact for iteration is minimal and should not matter too much. Even if only a single `keySetMeta()` is used inside the iteration loop, the iteration costs are insignificant. Only for trivial actions such as just changing a variable, counter or marker for every key the iteration costs become the lion's share. In such situations an *internal iterator* yields better results. For example, `ksForEach()` applies a user defined function for every key in a `KeySet` without having null pointer or out of range problems.

3.2.4 Trie vs. Split

Up to now, we have discussed external data structures visible to the user of the library. The application and plugin programmer needs them to access configuration. Last, but not least, we will show two internal data structures. The user will not see them. To understand the algorithm, however, the user needs to understand them as well.

Trie

TRIE or prefix tree is an ordered tree data structure. In Elektra, it provides the information to decide in which backend a key resides[35]. The algorithm, presented in Section 3.4 on page 52, also needs a list

[6] External cursor with checks is in a benchmark about 10% faster.

[7] Using an unchecked external cursor can be about 50% faster than using the internal cursor with `ksNext()`.

of all backends. The initial approach was to iterate over the `Trie` to get a list of all backends. But the transformation of a `Trie` to a list of backends, contained many bugs caused by corner cases in connection with the default backend and cascading mount points.

Split

So, instead of transforming the trie to a list of backends, we introduced a new data structure SPLIT. The name `Split` comes from the fact that an initial key set is split into many key sets. These key sets are stored in the `Split` object. `Split` advanced to the central data structure for the algorithm:

```
typedef struct _Split   Split;

struct _Split {
    size_t size;
    size_t alloc;
    KeySet **keysets;
    Backend **handles;
    Key **parents;
    int *syncbits;
};
```

The data structure `Split` contains the following fields:

size contains the number of key sets currently in `Split`.

alloc allows allocating more items than currently in use.

keysets represents a list of key sets. The keys in one of the key sets are known to belong to a specific backend.

handles contains a list of handles to backends.

parents represents a list of keys. Each `parentKey` contains the root key of a backend. No key of the respective key set is above the `parentKey`. The key name of `parentKey` contains the mount point of a backend. The resolver writes the file name into the value of the `parentKey`.

syncbits are some bits that can be set for every backend. The algorithm uses the `syncbits` to decide if the key set needs to be synchronised.

3.3 Error Handling

3.3.1 Terminology

It is sometimes unavoidable that ERRORS or other problems occur that ultimately have an impact for the user. Examples for such an error are that memory and hard disc space is exhausted. For a library it is

necessary to pass information about the facts and circumstances to the user because the user wants to be informed why a requested action failed. So Elektra gathers all information in these situations. We call this resulting information ERROR INFORMATION and WARNING INFORMATION depending on the severity.

If the occurred error is critical and ultimately causes a situation that the post conditions cannot be fulfilled we say that Elektra comes into a FAULTY STATE. Such a faulty state will change the control flow inside Elektra completely. Elektra is unable to resolve the problem without assistance. After cleaning up resources, a faulty state leads to immediate return from the function with an ERROR CODE. As a user expects from a library, Elektra never calls `exit()` or something similar, regardless of how fatal the error is. In this situation error information should be set.

On the other hand, for many problems the work can go on with reasonable defaults. Nevertheless, the user will be warned that a problem occurred using the *warning information*. These situations do not influence the control flow. But applications can choose to react differently in the presence of warning information. They may not be interested in any warning information at all. It is no problem if warning information is ignored because they are stored and remain accessible in a circular buffer. The implementation prevents an overflow of the buffer. Instead the oldest warnings are overwritten.

When error or warning information is presented to the user, it is called ERROR MESSAGE or WARNING MESSAGE. The user may reply to this message in which way to continue.

Error vs. Warning Information

When an error in an faulty state occurs, the error information must still hold the original error information. So even in problems that would cause a faulty state, otherwise, the error information must be omitted or transformed to a warning information. In some places only the adding of warning information is possible:

- The main purpose of `kdbClose()` is to free the handle. This operation is always successful and is carried out even if some of the resources cannot be freed. Therefore, in `kdbClose()`, setting error information is prohibited. Warning information is, however, very useful to tell the user the circumstance that some actions during cleanup failed.

- Also in `kdbOpen()`, only adding warning information is allowed. If `kdbOpen()` is not able to open a plugin, the affected backend will be dropped out. The user is certainly interested why that happened. But it was decided not to make it an faulty state, because the application might not even access the faulty part of the key hierarchy. An exception to this rule is if `kdbOpen()` fails to open the default backend. This situation will induce an faulty state.

- In `kdbSet()`, the cleaning up of resources involves calling plugins. But during this process Elektra is in a faulty state, so only adding of warning information is allowed. This ensures that the original error information is passed unchanged to the user.

On the other hand, any access to the key database can produce *warning information*. Plugins are allowed to yield warning information at any place and for any reason. For example, the storage plugin

reading a configuration file with an old syntax, is free to report the circumstance as warning information. Warning information is also useful when more than one fact needs to be reported.

3.3.2 Error Information

Reporting Errors

Reporting errors is a critical task. Users expect different aspects:

- The *user of the application* does not want to see any error message at all. If it is inevitable, he or she wants little, but very concrete information, about what he or she needs to do. The message should be short and concise. Some error information may already be captured by the application, but others like ”no more free disk space” have to be displayed. Conflicts should also be presented to the user. It is a good idea to ask how to proceed if a diversity of possible reactions exists. In case of conflicts, the user may have additional knowledge about the other program which has caused the problem. The user is more likely to decide correctly by which strategy the configuration shall be restored.
- The *user of the library* wants more detailed information. Categories of how severe the error is can help to decide how to proceed. Even more important is the information if it makes sense to try the same action again. If, for example, an unreliable network connection or file system is used, the same action can work in a second try.
- A *developer of the library*[8] wants full information about anything needed to be able to reproduce and locate potential bugs. Ideally the error information should even mention the file and line where the error occurred. This can help developers to decide if there is a bug inside Elektra or if the problem lies somewhere else.
- Vast information is needed to support correct error handling in *other programming languages*. In languages supporting exceptions, class name, inheritance or interface information may be necessary. Language specific extensions are, however, not limited to exceptions. Other ways of handling errors are continuations or `longjmp` in C. A plugin is free to add, for example, `jmp_buf` information to the error information.

It is certainly not a good idea to put all this previously mentioned information into a single string. Elektra chooses another way described in the next chapter.

Metadata

As stated above, a library always informs the user about what has happened, but does not print or log anything itself. One way to return an error information is to add a parameter containing the error information. In the case of Elektra, all `KDB` methods have a key as parameter. This key is additionally passed to every

[8] Library refers to both Elektra’s core and plugins.

plugin. The idea is to add the error and warning information as metadata to this key. This approach does not limit flexibility, because a key can hold a potentially unlimited number of metakeys.

The error information is categorised in metadata as follows:

error indicates that an faulty state is present. The value of the metakey contains the name of all the subkeys that are used for error indication. Metakeys do not guarantee any particular order on iteration. Instead the user can find out the information by looking at this metavalue.

Additional metakeys yield all the details.

error/number yields a unique number for every error.

error/description is a description for the error to be displayed to the user. For example, the metavalue can hold the text ”could not write to file”.

error/reason specifies the reason of the error. The human readable message is in the metavalue of `error/reason`. It states why the error occurred. One example for it is ”no disc space available”.

error/ingroup contains ”`kdb`” if the error occurred inside the core. It contains ”`module`” if the error happened while loading a module. The metavalue is ”`plugin`” if the error information comes from a plugin.

error/module indicates the name of the specific module or plugin.

error/file yields the source file from where the error information comes.

error/line represents the exact line of that source file.

As we see, the system is powerful because any other text or information can be added in a flexible manner by using additional metakeys.

Error Specification

The error specification in Elektra is written in simple colon-separated text files. Each entry has a unique identifier and all the information we already discussed above. No part of Elektra ever reads this file directly. Instead it is used to generate source code which contains everything needed to add a particular error or warning information. With that file we achieved a central place for error-related information. All other locations are automatically generated instead of having error-prone duplicated code. This principle is called ”Don’t repeat yourself”[22].

In Elektra’s core and plugins, C macros are responsible for setting the error information and adding warning information. In C only a macro is able to add the file name and line number of the source code. In language bindings other code may be generated.

Sources of Errors

`Key` and `KeySet` functions cannot expose more error information than the error code they return. But, of course, errors are also possible in these functions. Typically, errors concern invalid key names or null pointers. These problems are mostly programming errors with only local effects.

The most interesting error situations, however, all occur in `KDB`. The error system described here is dedicated to the four main `KDB` functions: `kdbOpen()`, `kdbGet()`, `kdbSet()` and `kdbClose()`. The place where the configuration is checked and made persistent is the source of most error information. At this specific place a large variety of errors can happen ranging from opening, locking up and saving the file. Sometimes in plugins, nearly every line needs to deal with an error situation.

3.3.3 Exceptions

Exceptions are a mechanism of the language and not just an implementation detail[41]. Exceptions are not intended to force the user to do something, but to enrich the possibilities. In this section, we discuss two issues related to exceptions. On the one hand, we will see how Elektra supports programming languages that provide exceptions. On the other hand, we will see how the research in exceptions helps Elektra to provide more robustness.

Language Bindings

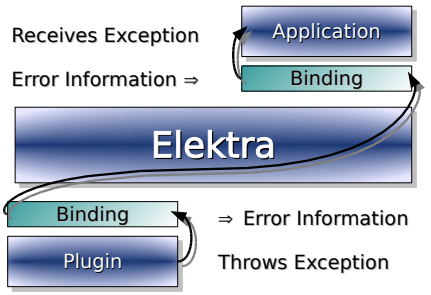

Figure 3.2: Exception Flow

C does not know anything about exceptions. But Elektra was designed so that both plugins and applications can be written in languages that provide exceptions as shown in Figure 3.2. One design goal of Elektra's error system is to transport exception-related information in a language neutral way from the plugins to the applications. To do so, a language binding of the plugin needs to catch every exception and transform it into error information and return an error code.

Elektra recognises the error code, stops the processing of plugins, switches to a faulty state and gives all the plugins a chance to do the necessary cleanups. The error information is passed to the application as usual. If the application is written in C or does not want to deal with exceptions for another reason, we are finished because the application gets the error information inside metadata as expected. But, if

the application is written in another language, the binding translates the error code to an exception and throws it. It is worth noting that the source and target language do not need to be the same.

Such a system needs a central specification of error information. We already introduced such a specification file in Section 3.3.2 on page 49. The exception classes and converters can be generated from it. An EXCEPTION CONVERTER is either a long sequence of try-catch statements that transforms every known exception into an appropriate metakeys. Each exception thrown by the plugin has to be caught. Alternatively, a converter can be a long switch statement for every error number. In every case the appropriate exception is thrown.

The motivation for using exceptions is that in C every return value has to be checked. On the other hand, the C++ exception mechanism allows the programmer to throw an exception in any place and catch it where it is needed. So in C++ the code is not cluttered with parts responsible for error handling. Instead, in a single place all exceptions of a plugin can be transformed to Elektra's error or warning information. The code for this place can be generated automatically using an exception converter.

Applications not written in C can also benefit from an exception converter. Instead of using the metadata mechanism, the error information can be converted to the exception mechanism already used for that application. We see that Elektra is minimally invasive in this regard.

Exception Safety

We can learn from the way languages define the semantics for EXCEPTION SAFETY. Exception safety is a concept which ensures that all resources are freed regardless of the chosen return path[40]. *Basic guarantees* make sure that some invariants remain on an object even in exceptional cases. On the other hand, *strong guarantees* assure that the investigated operation is successful or has no effect at all. Methods are said to be exception safe if the object remains in a valid state. The idea of exception safety is to ensure that no resource leaking is possible. `kdbSet()` written in C++ would look like:

Listing 3.6: Exception safety for plugins

```
try {
    plugin[1].set(); // may throw
    plugin[2].set(); // may throw
    plugin[3].set(); // may throw
    ...

    plugin[PLUGIN_COMMIT].set();
    // now all changes are committed
} catch (...) {
    // error situation, roll back the changes
    plugin[1].error(); // does not throw
    plugin[2].error(); // does not throw
    plugin[3].error(); // does not throw
    ...
```

```
        // now all changes are rolled back
        return -1;
    }
    // now do all actions on success after commit
    plugin[POSTCOMMIT].set(); // does not throw
    ...
    return 1;
    // commit successful
```

This pseudo code is much clearer than the corresponding C code. Let us explain the guarantee Elektra provides using this example. One by one plugin gets its chance to process the configuration. If any plugin fails, the semantics resemble that of a thrown exception. All other plugins will not be executed. Instead, the plugins get a chance to recover from the faulty state. In this catch part, the plugins are not allowed to yield any error information, but they are allowed to add warnings.

3.4 Algorithm

3.4.1 Introduction

In this section, we will explain the heart of Elektra. `kdbOpen()` is responsible for the setup and the construction of the data structures needed later. `kdbGet()` does, together with the plugins, all actions necessary to read in the configuration. `kdbSet()` orchestrates the plugins to write out the configuration correctly. `kdbClose()` finally frees all previously allocated data structures.

kdbOpen

`kdbOpen()` retrieves the *mount point configuration* with `kdbGet()` using the *default backend*. During this process, the function sets up the data structures which are needed for later invocations of `kdbGet()` or `kdbSet()`. All backends are opened and mounted in the appropriate parts of the key hierarchy. The resulting backends are added both to the `Split` and the `Trie` object. `kdbOpen()` finally returns a `KDB` object that contains all this information as shown in Figure 3.1.

The reading of the mount point configuration and the consequential self configuring of the system is called BOOTSTRAPPING. Elektra builds itself up from the simple variant with a default backend only to the sophisticated configuration system presented in this thesis.

`kdbOpen()` creates a `Split` object. It adds all backend handles and `parentKeys` during bootstrapping. So the buildup of the `Split` object takes place once. The resulting object is then used for both `kdbGet()` and `kdbSet()`. This approach is much better testable because the `Split` object is first initialised using the mount point configuration – separated from the filtering of the backends for every specific `kdbGet()` and `kdbSet()` request.

Afterwards the key hierarchy is static. Every application using Elektra will build up the same key database. Application specific mount points are prohibited because changes of mount points would de-

stroy the global key database. Elektra could not guarantee that every application retrieves the same configuration with the same key names any longer.

In `kdbOpen()`, nearly no checks are done regarding the expected behaviour of the backend. The contract checker guarantees that only appropriate mount points are written into the mount point configuration. `kdbOpen()` checks only if the opening of plugin was successful. If not, the backend enclosing the plugin is not mounted at all.

Removing Keys

In Elektra version 0.6, removing keys was an explicit request. Only a single `Key` object could be removed from the database. For configuration files this method is inapplicable. For `filesys`, however, it was easy to implement.

In Elektra version 0.7, the behaviour changed. Removing keys was integrated into `kdbSet()`. The user tagged keys that should be removed. After the next `kdbSet()`, these keys were removed from the key database. On the one hand, backends writing configuration files simply ignored the keys marked for removal. On the other hand, `filesys` needed that information to remove the files. To make this approach work for `filesys`, the marked keys were located at the very end of the `KeySet` and sorted in reverse. With this trick, recursive removing worked well. But this approach had major defects in the usage of `KeySet`. Because marking a key to be removed changed the sort order of the key set `ksLookupByName()` did not find this key anymore.

So in the present version removing keys is consistent again. A `KeySet` describes the current configuration. The user can reduce the `KeySet` object by *popping* keys out. The `kdbSet()` function applies exactly this configuration as specified by the key set to the key database. Contrary to the previous versions, the popped keys of the key set will be permanently removed.

The new circumstance yields IDEMPOTENT[33] properties for `kdbSet()`. The same `KeySet` can be applied multiple times, but after the first time, the key database will not be changed anymore[9].

It is, however, not known if keys should be removed permanently only by investigating the `KeySet`. But only if this knowledge is present, the core can decide if the key set needs to be written out or if the configuration is unchanged. So we decided to track how many keys are delivered in `kdbGet()`. If the size of the `KeySet` is lower than this number determined at the previous `kdbGet()`, Elektra's core knows that some keys were popped. Hence, the next `kdbSet()` invocation needs to change the concerned key database.

The situation is now much clearer. The semantics of popping a key will result in removing the key from the key database. And the intuitive idea that a `KeySet` will be applied to the key database is correct again.

[9]Note that `kdbSet()` actually detects that there are no changes and will do nothing. To actually show the idempotent behaviour the KeySet has to be regenerated or the key database needs to be reopened.

3.4.2 kdbGet

It is critical for application startup time to retrieve the configuration as fast as possible. Hence, the design goal of the `kdbGet()` algorithm is to be efficient while still enabling plugins to have relaxed postconditions. To achieve this, the sequence of SYSCALLS must be optimal. On the other hand, it is not tolerable to waste time or memory inside Elektra's core, especially during an initial request or when no update is available.

The synopsis of the function is:

```
int kdbGet(KDB *handle, KeySet *returned,
    Key * parentKey);
```

The user passes a key set, called `returned`. If the user invokes `kdbGet()` the first time, he or she will usually pass an empty key set. If the user wants to update the application's settings, `returned` will typically contain the configuration of the previous `kdbGet()` request. The `parentKey` holds the information below which key the configuration should be retrieved. The `handle` contains the data structures needed for the algorithm, like the `Split` and the `Trie` objects, as shown in Figure 3.1.

`kdbGet()` does a rather easy job, because `kdbSet()` already guarantees that only well formatted, non-corrupted and well-typed configuration is written out in the key database. The task is to query all backends in question for their configuration and then merge everything.

Responsibility

A backend may yield keys that it is not responsible for. It is not possible for a backend to know that another backend has been mounted below and the other backend is now responsible for some of the keys that are still in the storage. Additionally, plugins are not able to determine if they are responsible for a key or not. Consequently, it can happen that more than one backend delivers a key with the same name.

`kdbGet()` ensures that a key is uniquely identified by its name. Elektra's core will pop keys that are outside of the backend's responsibility. Hence, these keys will not be passed to the user and we get the desired behaviour: The nearest mounted backend to the key is responsible.

For example, a generator plugin in the backend A always emits following keys[10]:

```
user/sw/generator/akey (A)
user/sw/generator/dir (A)
user/sw/generator/dir/outside1 (A)
user/sw/generator/dir/outside2 (A)
```

It will still return these keys even if the plugin is not responsible for some of them anymore. This can happen if another backend B is mounted to `user/sw/generator/dir`. In the example it yields the following keys:

```
user/sw/generator/dir (B)
user/sw/generator/dir/new (B)
```

[10](A) and (B) indicate from which backend the key comes from.

```
user/sw/generator/dir/outside1 (B)
user/sw/generator/outside (B)
```

In this situation `kdbGet()` is responsible to pop all three keys at, and below, `user/sw/generator/dir` of backend A and the key `user/sw/generator/outside` of backend B. The user will get the resulting key set:

```
user/sw/generator/akey (A)
user/sw/generator/dir (B)
user/sw/generator/dir/new (B)
user/sw/generator/dir/outside1 (B)
```

Note that the key exactly at the mount point comes from the backend mounted at `user/sw/generator/dir`.

Sequence

`kdbOpen()` already creates a `Split` object for the whole configuration tree. In this object, `kdbOpen()` will append a list of all backends available. A specific `kdbGet()` request usually includes only a part of the configuration. For example, the user is only interested in keys below `user/sw/apps/userapp`. All backends that cannot contribute to configuration below `user/sw/apps/userapp` will be omitted for that request. To achieve this, parts of the `Split` object are filtered out. After this step we know the list of backends involved. The `Split` object allocates a key set for each of these backends.

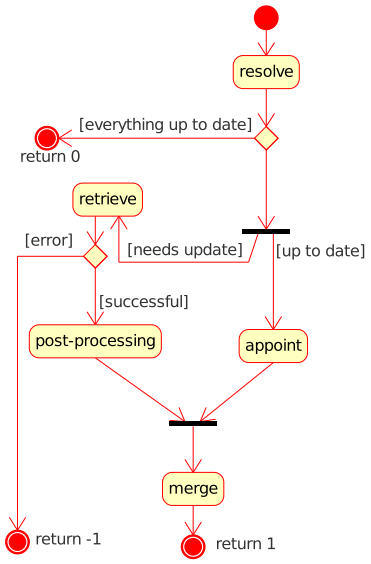

Figure 3.3: `kdbGet()` Algorithm

Afterwards the first plugin of each backend is called to determine if an update is needed. If no update is needed, the algorithm has finished and returns zero.

Now we know which backends do not need an update. For these backends, the previous configuration from `returned` is appointed from to the key sets of the `Split` object. The algorithm will not set the *syncbits* of the `Split` object for these backends because the storage of the backends already contains up-to-date configuration.

The other backends will be requested to *retrieve* their configuration. The initial empty `KeySet` from the `Split` object and the relevant file name in the key value of `parentKey` are passed to each remaining plugin. The plugins extend, validate and process the key set. When an error has occurred, the algorithm can stop immediately because the user's `KeySet returned` is not changed at this point. When this part finishes, the `Split` object contains the whole requested configuration separated in various key sets.

Subsequently the freshly received keys need some *post-processing*:

- Newly allocated keys in Elektra always have the *sync flag* set. Because the plugins allocate and modify keys with the same functions as the user, the returned keys will also have their sync flag set. But from the user's point of view the configuration is unmodified. So some code needs to remove this sync flag. To relax the post conditions of the plugins, `kdbGet()` removes it.
- To detect removed keys in subsequent `kdbSet()` calls, `kdbGet()` needs to store the number of received keys of each backend.
- Additionally, for every key it is checked if it belongs to this backend. This makes sure that every key comes from a single source only as designated by the `Trie`. In this process, all duplicated and overlapping keys will be popped in favour of the responsible backend as described in Section 3.4.2 on page 54.

The last step is to *merge* all these key sets together. This step changes the configuration visible to the user. After some cleanup the algorithm finally finishes.

Updating Configuration

The user can call `kdbGet()` often even if the configuration or parts of it are already up to date. This can happen when applications reread configuration in some events. Examples are signals[11], notifications, user requests and in the worst case periodical attempts to reread configuration.

The given goal is to keep the sequence of needed syscalls low. If no update is needed, it is sufficient to request the timestamp[12] of every file. No other syscall is needed. Elektra's core alone cannot check that because getting a timestamp is not defined within the standard C99. So instead the resolver plugin handles this problem. The resolver plugin returns 0 if nothing has changed.

This decision yields some advantages. Both the storage plugins and Elektra's core can conform to C99. Because the resolver plugin is the very first in the chain of plugins, it is guaranteed that no useless work is done.

Initial kdbGet Problem

Because Elektra provides self-contained configuration, `kdbOpen()` has to retrieve settings in the *bootstrapping* process below `system/elektra` as explained in Section 3.1.2 on page 35. Because of the new way to keep track of removed keys, the internally executed `kdbGet()` creates a problem. Without countermeasures even the first `kdbGet()` of a user requesting the configuration below `system/elektra` fails because the resolver finds out that the configuration is already up to date. The configuration delivered by the user is empty at this point. As a result, the empty configuration will be appointed and returned to the user.

[11] SIGHUP is the signal used for that on UNIX systems. It is sent when the program's controlling terminal is closed. Daemons do not have a terminal so the signal is reused for reloading configuration.

[12] On POSIX systems using `stat()`.

A simple way to resolve this issue is to reload the default backend after the internal configuration was fetched. Reloading resets the timestamps and `kdbGet()` works as expected.

3.4.3 kdbSet

Not the performance, but robust and reliable behaviour is the most important issue for `kdbSet()`. The design was chosen so that some additional in-memory comparisons are preferred to a suboptimal sequence of *syscalls*. The algorithm makes sure that keys are written out only if it is necessary because applications can call `kdbSet()` with an unchanged `KeySet`. For the code to decide this, performance is important.

Properties

`kdbSet()` *guarantees* the following properties:

1. Modifications to permanent storage are only made when the configuration was changed.
2. When errors occur, every plugin gets a chance to rollback its changes as described in Section 3.3.3 on page 51.
3. If every plugin does this correctly, the whole `KeySet` is propagated to permanent storage. Otherwise nothing is changed in the key database. Plugins developed during the thesis meet this requirement.

The synopsis of the function is:

```
int kdbSet(KDB *handle, KeySet *returned,
    Key * parentKey);
```

The user passes the configuration using the `KeySet returned`. The key set will not be changed by `kdbSet()`. The `parentKey` provides a way to limit which part of the configuration is written out. For example, the `parentKey user/sw/apps/myapp` will induce `kdbSet()` to only modify the key databases below `user/sw/apps/myapp` even if the `KeySet returned` also contains more configuration. Note that all backends with no keys in `returned` but that are below `parentKey` will completely wipe out their key database. The `KDB` handle contains the necessary data structures as shown in Figure 3.1.

Search for Changes

As a first step, `kdbSet()` *divides* the configuration passed in by the user to the key sets in the `Split` object. `kdbSet()` searches for every key if the *sync flag* is checked. Then `kdbSet()` decides if a key was removed from a backend by comparing the actual size of the key set with the size stored from the last `kdbGet()` call. We see that it is necessary to call `kdbGet()` first before invocations of `kdbSet()` are allowed.

We know that data of a backend has to be written out if at least one key was changed or removed. If no backend has any changes, the algorithm will terminate at this point. The careful reader notices that the process involves no file operations.

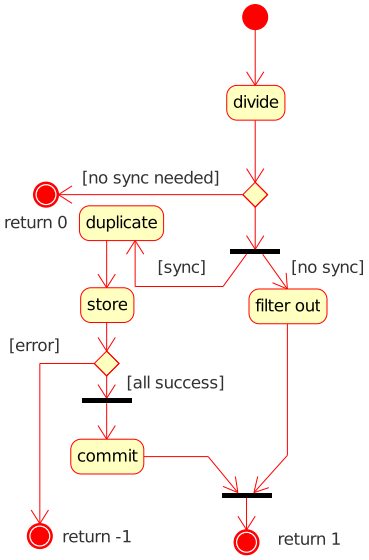

Figure 3.4: `kdbSet()` Algorithm

Duplicated Key Sets

If some backends need synchronisation, the algorithm continues by filtering out all backends in the `Split` object that do not have changes. At this point, the `Split` object has a list of backends with their respective key sets.

Plugins in `kdbSet()` can change values. Other than in `kdbGet()`, the user is not interested in these changes. Instead, the values are transformed to be suitable for the storage. To make sure that the changed values are not passed to the user, the algorithm continues with a *deep duplication* of all key sets in the `Split` object.

Resolver

All plugins of each included backend are executed one by one up to the resolver plugin. If this succeeds, the resolver plugin is responsible for committing these changes. After the successful commit, *error codes* of plugins are ignored. Only logging and notification plugins are affected.

Atomic Replacement

For this thesis only file-based storage with atomic properties were developed. The replacement of a file with another file that has not yet been written is not trivial. The straightforward way is to lock a file and start writing to it. But this approach can result in broken or partially finished files in events like ”out of disc space”, signals or other asynchronous aborts of the program.

A temporary file solves this problem, because in problematic events the original file stays untouched. When the temporary file is written out properly, it is renamed and the original configuration file is over-

written. But another concurrent invocation of `kdbSet()` can try to do the same with the result that one of the newly written files is lost.

To avoid this problem, locks are needed again. It is not possible to lock the configuration file itself because it will be unlinked when the temporary file is renamed. So a third file for locking is needed. The resolver currently implements this approach.

An alternative to this approach without locks is to completely rely on the modification time. The modification time typically has only a resolution of one second. So any changes within that time slot will not be recognised. For this approach, however, the name of every temporary file must be unique because concurrent `kdbSet()` invocations each try to create one. The temporary file must also be unlinked in case of a rollback. The opened temporary file can be passed to the storage plugins using a file name in the directory `/dev/fd`. This approach may be more practical than the currently implemented way because it does not need the additional lock file[13].

Errors

The plugins within `kdbSet()` can fail for a variety of reasons. CONFLICTS occur most frequently. A conflict means that during executions of `kdbGet()` and `kdbSet()` another program has changed the key database. In order not to lose any data, `kdbSet()` fails without doing anything. In conflict situations Elektra leaves the programmer no choice. The programmer has to retrieve the configuration using `kdbGet()` again to be up to date with the key database. Afterwards it is up to the application to decide which configuration to use. In this situation it is the best to ask the user, by showing him the description and reason of the error, how to continue:

1. Save the configuration again. The changes of the other program will be lost in this case.
2. The key database can also be left unchanged as the other program wrote it. After using `kdbGet()` the application is already up to date with the new configuration. All configuration changes the user made before will be lost.
3. The application can try to merge the key sets to get the best result. If no key is changed on both sides the result is clear, otherwise the application has to decide if the own or the other configuration should be favoured. The result of the merged key sets has to be written out with `kdbSet()`.

   Merging the key sets can be done with `ksAppend()`. The source parameter is the preferred configuration. Note that the downside of the third option is that a merged configuration can be an not validating configuration.

Sometimes a concrete key causes the problem that the whole key set cannot be stored. That can happen on validation or because of type errors. Such errors are usually caused by a mistake made by the user. So the user is responsible for changing the settings to make it valid again. In such situations, the *internal cursor* of the `KeySet returned` will point to the problematic key.

[13] Nevertheless, the other way was chosen to test if the algorithm is exception safe as described in Section 3.3.3 on page 51.

A completely different approach is to export the configuration when `kdbSet()` returned an error code. The user can then edit, change or merge this configuration with more powerful tools. Finally, the user can import the configuration into the global key database. The export and import mechanism is called ”streaming” and will be explained in Section 4.3 on page 70.

CHAPTER 4

Plugins

Figure 4.1 presents an overview of the plugins we developed in this thesis. The notation of generalisation[1] has the usual meaning as a subtype or is-a relationship. No code-reuse or inheritance is needed or present between the plugins. The notation of abstract classes[2] groups the plugins together and is not reflected by any real code. These groups represent a shared concept and often can be expressed by the clause `infos/provides`.

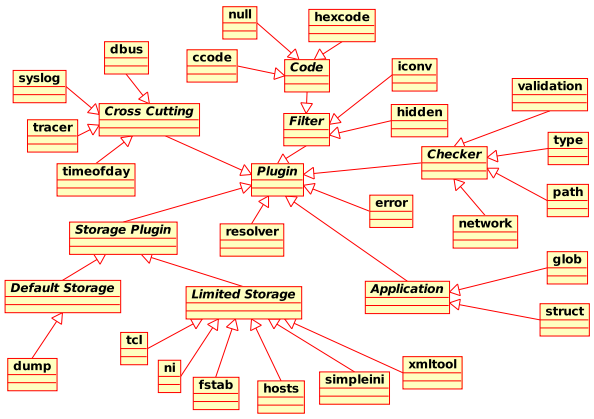

Figure 4.1: Overview of Plugins

We will describe two large groups of plugins in their own sections: storage and filter plugins. But we will handle checker plugins, and plugins which apply metadata together because they are often in combined use.

[1] the arrows between two boxes
[2] the class names written in italic

We introduced the cross-cutting concerns in Section 2.3.3 on page 27. For some of these concerns, the implementation pointed out to be straightforward and will not be discussed here. The *error plugin* comes in handy for test purposes. It yields error information and returns error status codes on request and will be discussed in Section 5.2.2 on page 87. The *version plugin*[3] exports information about the core library. The constants below the key `system/elektra/version` indicate the exact version of Elektra.

The *hidden plugin* hides `Key` objects whose names start with a "`.`". We presented the resolver plugin in Section 2.3.2 on page 26. We explained the details of the resolver throughout Chapter 3, because the control flow in the algorithm needs the return value of the resolver plugin to decide how to proceed.

4.1 Introduction

4.1.1 Interface

The collaboration of plugins stands and falls with the interface. Only with the same interface can the framework guarantee exchangeability of plugins. Elektra solves this problem by presuming a specific interface for every plugin.

This thesis introduces a new API for the plugins. The name of the functions in plugins is recommended[4] to be `elektraPluginAction()`. `Plugin` stands for the real plugin's name. In the text, however, `Plugin` will be used if we do not refer to a specific plugin. The interface basically consists of five different actions with the function names: `elektraPluginGet()`, `elektraPluginSet()`, `elektraPluginOpen()`, `elektraPluginClose()`, and `elektraPluginError()`.

They are invoked as described:

- `kdbOpen()` calls `elektraPluginOpen()` of every plugin to let them do their initialisation.
- `kdbGet()` requests `elektraPluginGet()` of every plugin in the queried backends to return a key set.
- `kdbSet()` usually calls `elektraPluginSet()` of every plugin in the queried backends to store the configuration.
- `kdbSet()` also calls `elektraPluginError()` for every plugin when an error happens as explained in Section 3.3.3 on page 51. Because of `elektraPluginError()`, plugins are guaranteed to have their chance for necessary cleanups.
- `kdbClose()` makes sure that plugins can finally free their own resources in `elektraPluginClose()`.

[3] The Figure 4.1 does not show the version plugin because it is internal and not developed separately.

[4] Actually the function name can be arbitrary as long as it is unique within the application. Only the name exported by the contract and the plugin factory matters.

The synopsis of the functions are:

```
int elektraTracerOpen(Plugin *handle, Key *errorKey);
int elektraTracerClose(Plugin *handle, Key *errorKey);
int elektraTracerGet(Plugin *handle, KeySet *returned, Key *parentKey);
int elektraTracerSet(Plugin *handle, KeySet *returned, Key *parentKey);
int elektraTracerError(Plugin *handle, KeySet *returned, Key *parentKey);
```

handle is a pointer to the plugin's data structure.

returned is the data structure to be processed.

errorKey is a key where error information can be added as metadata.

parentKey has the mount point as a key name and the path to the configuration file as a value. Additionally, the key has the same purpose as the `errorKey`.

A plugin can call three functions with its parameter `handle`: `elektraPluginGetConfig()`, `elektraPluginGetData()` and `elektraPluginSetData()`. The *plugin configuration* can be retrieved with `elektraPluginGetConfig()`. Within this configuration, cascading is recommended so that both the `user` and the `system` configuration of the plugins are taken into account. Plugins are only allowed to have local variables to be reentrant. Note that also **`static`** variables are not allowed. But plugins sometimes need a state, a cache or some data to be preserved between invocations. The last two functions serve this purpose.

Information Flow

Most component systems pass information between the various components by calling methods of each other. This is not the way Elektra's plugin system works. Instead, the core passes a `KeySet` object in one direction from plugin to plugin. Each of the plugins can modify the configuration or add any other information using metakeys. While this approach is in general less flexible, this information flow still allows powerful chaining. Because plugins do not have to bother to call other plugins, the plugin development is also easier. The ordering of plugins in backends is controlled using contracts.

4.1.2 Contracts

Every plugin should provide a full contract to give information how it will work with other plugins. Most parts of the contract are obligatory. Plugins cannot be loaded without this information. For example, plugins must provide the clause `infos/version`. It is vital so that the plugin loader knows which version of Elektra the plugin was built for.

Conditions

It is, however, up to the plugin not to have every clause of the contract. For example, the plugin might not tell what it provides or needs. It can also leave any description out. In this situation it is unclear what the plugin will do. Such a plugin can add or remove keys, and changes values and metadata regardless what other plugins expect. If only such plugins existed there would be chaos. It would be impossible to determine the behaviour of a backend which uses a multiple of such plugins.

To avoid this situation, every plugin written during the thesis exports a contract describing how the plugin modifies the `KeySet returned`. Most often it is enough to state that it is a *storage* plugin or that it will *filter* keys.

The data structures, however, are already responsible for most of the pre- and postconditions. Every condition the data structure guarantees, takes away a concern for the plugins. All the parts that are already guaranteed by data structures do not need to be stated in the contract.

Plugins should not be burdened to check too many postconditions. Instead, plugins focus on their task. The plugin does not need to check the sync flag of keys or if the keys are below the mount point. The core already guarantees correct behaviour as described in Section 3.4 on page 52.

To sum up, contracts give the information how a plugin interacts with others. It describes if, and how, the `KeySet returned` is changed. Using contracts, we get a predictable behaviour, but still support every kind of plugin.

Exporting Contracts

As already stated, some parts of the contracts are obligatory. `kdb mount` needs to know which symbols the plugin exports. Only the `elektraPluginGet()` symbol is mandatory - it is used to generate this information. Elektra's core also uses the functions `elektraPluginSet()`, `elektraPluginError()`, `elektraPluginOpen()` and `elektraPluginClose()` if available. Other functions like `serialise`, `unserialise` or `lookup` which implement special features can be supported, but are ignored by the core. For the user of the library these functions can be very useful. These functions shall either belong to the concern of the plugin or be implemented within the plugin because of the dependences.

As described in Section 2.2.2 on page 20, the plugin can also provide descriptive information, for example about the author and the licence. Advanced plugins can also export plugin configuration for other plugins so that the overall backend works properly. Last, but not least, as enumerated in Section 2.3.4 on page 30 dependency and placement information makes the system reliable and robust. With that information, plugins can be placed into a backend in an automatic and secure way.
`system/elektra/modules` provides for every module the information described above. The entry exists once a plugin of that module is loaded. For each module a special *module backend* is generated and mounted at `system/elektra/modules/pluginname`. The `elektraPluginGet()` function generates this described contract on requests.

For example, the *ccode plugin*, implements:

Listing 4.1: Exporting of Contract

```
int elektraCcodeGet(Plugin *handle, KeySet *returned, Key *parentKey)
{
    if (!strcmp (keyName(parentKey), "system/elektra/modules/ccode"))
    {
        KeySet *contract = ksNew (30,
            keyNew ("system/elektra/modules/ccode",
                KEY_END),
            keyNew ("system/elektra/modules/ccode/exports",
                KEY_END),
            //...
            KS_END);
        ksAppend (returned, contract);
        ksDel (contract);
        return 1;
    }
    // implementation of elektraCcodeGet
```

We see in Listing 4.1 that the plugin generates and returns the contract if, and only if, the name of the `parentKey` is `system/elektra/modules/ccode`. The user and the contract checker can access the contract of ccode below the key `system/elektra/modules/ccode` in the same way other configuration is accessed.

Changing Plugins

This configuration is static and contains the contract information. In theory, the contract can be changed without any problems in ways that it provides more and obligates less. But the problem is that it will not be checked if this is the case because a recheck of the contracts of a backend is very expensive. The contract checker doing this, only runs once during mount time. Changing contracts in an incompatible way forces the user to remove all mount points where the plugin is and mount it again. Such actions are only sustainable in a development phase and not in a productive environment.

But the plugin's implementation is allowed to change without being remounted if it is a subtype of the earlier version. Only in this situation it can be a drop-in placement. With a good testing framework the behaviour can be checked to some extent.

We also see in Listing 4.1 that the code responsible for generating the contract and the code for the implementation are next to each other. Plugins need to satisfy those self-imposed obligations that are described in contracts. They ensure that plugins interact in predictable ways. So the process of writing individual plugins and composing them together can be described as Component-Based Software Engineering[4].

Plugins can also be viewed as framework extensions. A component abstracts plugins. But this term is misleading in our case, because components usually can choose which interfaces they implement.

Elektra's plugins, however, are restricted to implement one specific interface. Without contracts, plugins could not interact as described in this chapter.

4.2 Filter

FILTER PLUGINS process keys and their values in both directions. In one direction they undo what they do in the other direction. Most filter plugins available now encode and decode values. Storage plugins that use characters to separate key names, values or metadata will not work without them.

4.2.1 Encoding

All occurrences of delimiting characters in storage plugins have to be eliminated using other characters. A common approach is to define a so-called ESCAPE CHARACTER giving characters another meaning. After the escape character, there might be the same character or another sequence of characters that can be decoded back to the original characters. Using other characters has the advantage that the storage plugin does not have to be aware of this character because the unwanted character will not show up at all. Another way to en- or decode is to split up the content into small blocks of data and translate each block to a character. *base64*[23] uses this technique. Plugins doing such operations will be referred to *code plugins*.

Contracts define which characters must disappear so that the storage plugin can be used without a hassle. During mounting the user can add one of the code plugins. If no code plugin is added, the backend will not be validated, because of the `infos/needs` clause in the storage plugin. When the user selects a code plugin, the plugin configuration will be extended. The storage plugin provides the necessary configuration in the clause `config/needs` for the code plugin. The code plugin will automatically retrieve the configuration it needs so that they can work together. The user, however, may choose to escape even more characters.

Character Reduction

To store a large range of different characters in a sink which allows only a subset of characters to occur there, a plugin must *reduce* the range of characters in a reversible way. For example, the ASCII character set supports characters in the range $0-127$. A storage plugin may only be able to represent the characters in the ranges $22-50$ and $62-127$. Code plugins are able to reduce the character set. Key values can have characters in the range $1-255$. Binary values even can have characters in the range $0-255$.

The contract checker must check if a plugin is present. It will do the necessary reduction at run time because a code plugin is able to transform the character range with a given configuration.

The configuration of the code plugins define which characters should be escaped. The configuration is interchangeable between these plugins, yielding the advantage that they are a drop-in replacement for each other.

A more sophisticated character reduction would negotiate the optimal configuration. An open problem is that key names, metanames and metavalues are currently not escaped. For key names renaming may change the order.

Hexcode

This code plugin translates each unwanted character into a two cypher hexadecimal character. The escape character itself always needs to be encoded, otherwise the plugin would try to interpret the following two characters in the text as a hexadecimal sequence.

Let us look at an example. Consider the following input:

```
value=abc xyz
```

Assuming the escape character is % the input would be encoded to:

```
value%3Dabc%20xyz
```

The disadvantage is that the length of the resulting string increases. In the worst case the hexcode plugin makes the value three times larger.

Ccode

When writing a C string in C code some characters cannot be expressed directly but have a special one letter abbreviation. The CCODE PLUGIN allows us to map any single escaped character to be replaced by another single character and vice versa. The user can configure this mapping. Table 4.1 shows the default mapping.

Table 4.1: Ccode escape characters, thanks to Wilson Mar[29]

Escape	Name	Hexadecimal
b	backspace	08
t	horizontal tab	09
n	new line feed	0A
v	vertical tab	0B
f	form feed	0C
r	carriage return	0D
	back slash	5C
’	single quote	27
"	double quote	22
0	null	00

This method of encoding characters is not as powerful as the hexcode plugin in terms of reduction. The hexcode plugin allows reduction of the character set to $'0' -' 9'$, $'a' -' f'$ and one escape character. So it can represent any key value with only 17 characters. On the other hand, ccode cannot reduce the set more than by half. So when all control characters and non-ASCII characters need to vanish, it cannot be done with the ccode plugin. But it is perfectly suitable to reduce by some characters. The advantages are that the size only doubles in the worst case and that it is much easier to read.

Iconv

Consider a user insisting on a `latin1` character encoding because of some old application. All other users already use, for example, UTF-8. For these users, the configuration files are encoded in UTF-8. So we need a solution for the user with `latin1` to access the key database with proper encoding.

On the other hand, contemplate an XML file which requires a specific encoding. But the other key databases work well with the users encoding. So a quick fix for that backend is needed to feed that XML file with a different encoding.

Iconv plugin provides a solution for both scenarios. It converts between many available character encodings. With the plugin's configuration the user can change the `from` and `to` encoding. The default values of the plugin configuration are:

from encoding will be determined at run time.

to encoding is UTF-8.

Note that for writing the configuration `from` and `to` is swapped. A key database that requires a specific encoding can make use of it. To sum up, every user can select a different encoding, but the key databases are still properly encoded for anyone.

4.2.2 Reversibility

For filter plugins reversibility is essential. Their placement is in `postgetstorage` and `presetstorage`. They are not allowed to deactivate `infos/stacking` in the contract, because this would destroy the overall reversibility.

Let us look at the following example. We will encode a string twice with two different configured ccode plugins. The ccode plugin A has the rule to encode:

`U` $\xrightarrow{A}$ `\u`

and the reverse to decode:

`\u` $\underset{A}{\longrightarrow}$ `U`

The B ccode converts:

`=` $\xrightarrow{B}$ `\e`

Both need the rule of escaping the escape character. Otherwise the encoder would be confused if escape characters appear in the text:

`\` $\xrightarrow{AB}$ `\\`

So a sequence in correct order would be:

`aUx=` $\xrightarrow{A}$ `a\ux=` $\xrightarrow{B}$ `a\\ux\e` $\underset{B}{\longrightarrow}$ `a\ux=` $\underset{A}{\longrightarrow}$ `aUx=`

Obviously it also yields the same if B is applied first. But the intermediate steps are different then:

`aUx=` $\xrightarrow{B}$ `aUx\e` $\xrightarrow{A}$ `a\ux\e` $\underset{A}{\longrightarrow}$ `aUx\e` $\underset{B}{\longrightarrow}$ `aUx=`

Note that in the correct order it will always lead to the correct result, because the input of each plugin for decoding is the same as the output of the previous encoding pass.

And because for any code plugin X

$x \xrightarrow{X} y \underset{X}{\longrightarrow} x$

must be valid for any value x the overall en- and decode process of multiple code plugins must also work.

If we apply the decoding of the B plugin in the wrong order, however, we run into a problem:

`aUx=` $\xrightarrow{A}$ `a\ux=` $\xrightarrow{B}$ `a\\ux\e` $\underset{A}{\longrightarrow}$ `a\Ux\e`

which cannot be decoded by the B plugin that is unaware of `\U`. It is not guaranteed, however, that this encoding error will be found in such situations. `\U` could be well defined and the sequence could lead to a wrong result.

4.2.3 Null values

Elektra supports null values for binary keys. They differ from an empty string substantially in how they are handled. Some plugins that work on values are not aware of null values. They would crash when they try to access a value that is believed to be a string.

Other storage plugins also have problems with empty strings. The storage *plugin simpleini* that parses the $key = value$ expression by:

```
fscanf (fp, "%ms␣=␣%ms\n", &key, &value)
```

fails badly in recognising a line like:

```
user/simpleini/something =
```

The *null plugin* effortlessly solves both problems. It transcodes all null values to `@NULL`, empty strings to `@EMPTY` and every string starting with `@` to a string beginning with `@@`.

Using the null plugin simpleini generates lines like:

```
user/simpleini/example_empty_string = @EMPTY
user/simpleini/example_null_value = @NULL
user/simpleini/example_text = @@text
```

If a plugin states in its contract that it needs a null plugin, it is guaranteed that no empty or null value is passed to it. That might simplify the programming in some cases. It also reduces the possibility of errors. So if the null values are not handled, the plugin must write ”null” in the `infos/needs` clause of the contract.

4.3 Storage

Storage plugins belong to the most important plugins in Elektra. Their purpose is to make configuration permanent in key databases and to read preferences from there.

Storage plugins are usually implemented in `elektraPluginGet()` and `elektraPluginSet()`, but do not provide `elektraPluginOpen()`, `elektraPluginClose()` and `elektraPluginError()`.

On the one hand, in `elektraPluginSet()` they try to open the file in write-only mode. The file name is passed into the value of the parent key. Now the whole content of the `KeySet` needs to be

serialised into this file. The algorithm consists of iterating over the `KeySet` and writing out the keys and their values. If the file syntax or format is not suitable to store every type of a key, the contract has to declare what the configuration must look like by providing the clause `config/needs` together with an `infos/needs`.

On the other hand, `elektraPluginGet()` opens the file read only. The absence of the file is not considered to be an error, but the resolver will generate a warning. The `KeySet returned` is empty and waits to be filled by parsing the file content. Note that `elektraPluginGet()` also has to return the contract on request.

Serialising

SERIALISATION is based on a simple principle. It is about converting data structure into a string. These strings can be written on a hard disk or sent over a network. Afterwards the data structure can be rebuilt by parsing the string. *Storage plugins* in Elektra do nothing more than serialising a `KeySet` to make them persistent.

C can express this conversion in a function, but has the problem that serialising into files or strings needs a different interface. So it is not possible with `<stdio.h>` to provide a method which can serialise a `KeySet` to either a string or a file. Moreover, the API to strings does not have a way to enlarge the buffer when it is too small. In C99, it is at least possible with `snprintf` to check for buffer overruns and allocate enough space afterwards.

When using C++, the described issue presents no problem because there are streams which support the desired behaviour out of the box. And even better – once a C++ serialiser is available it is trivial to wrap it to additionally provide a C serialiser.

It is complicated in its details to fully serialise any data structure possible in a programming language like C. There are different levels of complexity regarding serialisation. But the key database semantics were chosen to make the serialisation as simple and efficient as possible. Only back referencing to old metadata can be implemented for memory optimisation purposes as explained in Section 3.2.2 on page 41. There are no other pointers in a `KeySet`. So the hard problems to detect and handle cycles are not present. There is also no class hierarchy. `Key` and `KeySet` are the only classes which need to be serialised.

Another problem of serialisation is the byte order. Elektra supports binary values, but the byte order for writing data is application specific. So applications must make sure that they use a portable byte order or mark the byte order using metadata.

Streaming

STREAMING of configuration is basically the same as serialisation. It only differs in that the sink and the source are streams rather than the key database. The stream could be `stdout`, a network or a user-provided file. With streaming it is possible to serialise and deserialise without having to use the global key database.

Storage plugins should export symbols to functions that provide such streaming. The functions are usually present anyway. The approach becomes especially useful if the API across plugins is the same, so

there should be a suggestion how to call their C and C++ pendants. The plugin's contract is responsible to export these symbols. Applications can find the pointer to the function, for example, in `system/elektra/modules/dump/exports/serialize`.

The streaming comes in handy when converting configuration files from any source to any target. One plugin unserialises the stream to a `KeySet` object while a second plugin serialises it back to a stream. We see that the key database is not involved.

4.3.1 Dump Storage

Dump is the name of the storage plugin that supports full Elektra semantics. Combined with a resolver plugin it already assembles a fully featured backend. No other plugins are needed.

The file format consists of a simple command language with arguments. When an argument is binary or string data the length needs to passed first. Because the size is known in advance, any binary dump is accepted. Terminating characters present no problem. The commands are assembled similar to the ones present in Elektra's API.

Configuration Format

The file starts with the magic word `kdbOpen` followed by a version number. Processing can be stopped immediately when it is not in Elektra's dump format at all. A wrong version number most likely indicates that the version of the plugin is too old to recognise all commands in the file.

The basic idea of the dump plugin is to write out the way that the `KeySet` needs to be constructed. The dump plugin interprets such a file. The file also looks similar to C code that would create the `KeySet`.

Keys can contain any binary values and arbitrary metadata and are still stored and parsed correctly. The dump plugin can even reconstruct pointers to metadata to save memory. When a pointer to the same region of memory is found, a special command `keyCopyMeta` is written out that is able to reconstruct the data structure the way it was before. The commands were designed to make parsing of the file an easy task.

The plugin helps to define the semantics of Elektra's key database as presented in Section 2.2.3 on page 22. With the given commands, any possible `KeySet` can be written out and reconstructed identically. It contains the commands which are needed to represent all semantics a `KeySet` has.

Example

The serialised configuration can look like:

Listing 4.2: Dump Example

```
kdbOpen 1
ksNew 207
keyNew 27 1
system/elektra/mountpoints
keyMeta 8 27
```

```
commentBelow are the mountpoints.
keyEnd
keyNew 32 19
system/elektra/mountpoints/dbusserialised Backend
keyEnd
keyNew 39 1
system/elektra/mountpoints/dbus/config
keyMeta 8 72
commentThis is a configuration for a backend, see subkeys for more information
keyEnd
keyNew 53 1
system/elektra/mountpoints/fstab/config/struct/FStab
keyCopyMeta 59 11
system/elektra/mountpoints/file systems/config/struct/FStabcheck/type
keyEnd
ksEnd
```

Because C strings are null terminated a null character is at the end of each parameter containing a name or value. Null characters are, however, omitted in Listing 4.2.

4.3.2 Limited Storage

Storage plugins may be limited in various ways. The goal of the mount process is to determine a backend that hides these disabilities from the user. Two issues cannot be compensated by filter plugins. A problem occurs if the syntax is limited in such a way that it is not possible to store metadata. The other matter occurs if the file format is restricted in which key names the configuration is allowed to have. Here we say that the plugin is limited in which *types* it can express. If a plugin has one of these problems, it will be referred to as a limited storage plugin. Applications have to be aware of that fact.

Metadata

A way to support arbitrary metadata in a limited configuration storage is to separate the storage plugin in two parts: the value storage and the METADATA STORAGE. Of course, both need to persist key names to identify a `Key` within the storage. But the value storage's sole concern is to write out the key's name and value. It does not have to be aware that metadata exists at all. This makes the semantics simple so that even trivial configuration files can be supported. In the following example, we see the fully compliant value storage *simpleini*. It needs the equal sign and line break to be escaped.

Listing 4.3: INI Example

```
user/sw/app/key name1 = key value 1
user/sw/app/key name2 = key value 2
```

We will also describe another way to store metadata in a storage not aware of metadata. The idea is to create a new key name that consists of the key name and the metaname concatenated. To recognise that such a renaming took place, a special prefix is prepended as the following examples shows:

```
user/sw/app/key name1/_METAcomment = here is a comment
```

This technique can be used for encrypted, zipped or otherwise changed values so that the user never gets the wrong value with a given key name even if the responsible plugin is missing[2]. Otherwise, a plugin in the chain knows what to do when such a renaming is done.

Types

Configuration storage may also be limited in the types they can express. This situation cannot be circumvented. The user of the key database needs to know that in a specific place the keys must satisfy some restrictions.

For example, the configuration file may be limited that only numbers can occur at specific places. Configuration files can also have limits in the structure of configuration they can express. Some do not have support for any nesting.

An example of such a format is `/etc/hosts` implemented by the *hosts plugin* as we will see in the next subsection.

4.3.3 Ordering of Keys

Sometimes the built-in ORDERING of `KeySet` is not the desired order in the key database. Application or plugin programmers may want a particular sequence not given by the name. For example, in an application a list of file names is needed. The order of this list matters because the application tries out the files in that sequence. Another example is a plugin that wants to preserve the ordering that was found in a configuration file. A `KeySet`, however, is not directly able to represent such an alternative order. An obvious way is to prefix the key name with a number to order them, but this is no option if the key name is canonical. But metadata solve both problems without changing the names.

Hosts

To preserve the order of configuration items in the configuration file `/etc/hosts`, the HOSTS PLUGIN uses the metadata `order` that indexes a particular element. It is a number larger than one and indicates the position within the file.

The hosts file looks like:

Listing 4.4: Hosts Example

```
#/etc/hosts file

# a new comment
192.168.0.23    first_entry
```

```
192.168.0.24    second_entry alias2 alias3 alias4 alias5

# another comment



127.0.0.1   localhost
192.22.22.3 third_entry
```

The hostname acts as a canonical item in the file, so the key name is defined by appending the hostname. The IP address is the value of this key. Alias names also happen to be unique per hostname, so their names are also appended.

Using that schema the following keys are generated:

```
user/hosts /home/markus/.config/hosts
user/hosts/first_entry 192.168.0.23
user/hosts/localhost 127.0.0.1
user/hosts/second_entry 192.168.0.24
user/hosts/second_entry/alias2
user/hosts/second_entry/alias3
user/hosts/second_entry/alias4
user/hosts/second_entry/alias5
user/hosts/third_entry 192.22.22.3
```

Preserving Configuration Files

Empty lines and lines beginning with # are comments. They will be preserved together with comments in the same line in the metakey `comment` of the following key. Comments after the last valid entry will be discarded. They would need special handling. Alternatively, the first comment could be in the root key and the comments below always go to the respective keys. The user's expectations in western countries, however, is rather that text above the `Key` describes what the `Key` is for. We see that *localisation* issues can also be a topic within configuration files.

For the alias names no order preserving is done. So they will always be in alphabetical order when the file is written out again. A similar strategy for preserving the order could be used as well.

Whitespaces are not kept either in all places, because the parser skips them. It would require a rewrite of the parser to achieve that detail.

The problems mentioned above should give the plugin programmer an insight what needs to be done in order to preserve a file exactly. These problems are, however, fixable with the arbitrary metadata introduced in this thesis.

The easier variant of this problem is called ROUND-TRIPPING[38]. If one converts a `KeySet` to the external representation and back again, the new `KeySet` must be equal to the old one. Every storage

plugin must fulfil this criteria for configuration. Not every storage plugin, however, can guarantee the same for metadata.

Implementation of Order Preservation

The actual implementation of order preservation is done using a C array. In the array, the `Key` objects are sorted by the index given by the metavalue `order`. During the serialisation process for every key in the C array, `ksLookup()` finds out if the key has any aliases. Consequently, they just need to be written out.

The same `Key` can be aliased from both the array and the `KeySet`. The *hosts storage plugin* does not free the keys referenced by the array. If the application wants to free the keys of the array, the programmer can prevent double-frees by using the reference counter when the key is added or removed from the array. This will happen automatically using the C++ bindings.

4.3.4 Other Representations

TCL Lists

Like other programming languages, TCL is able to represent data. That means that the same file which is parsed by an Elektra plugin as configuration can also be executed by a TCL interpreter. The format of dictionaries and lists is something which works well using TCL syntax[34]:

```
% set ol {{key val {a b c d}} {key val {a b c d}}}
{key val {a b c d}} {key val {a b c d}}
```

The *tcl plugin* implemented another variant. It uses a whitespace skipper and so whitespaces are not allowed to be significant. So we introduced a similar version where spaces can be removed without changing the configuration:

```
{     {user/key=val {metakey=b} {comment  =  huhu  }   }  }
```

Which is equivalent to:

Listing 4.5: TCL Lists Example

```
{
    user/key = val
    {
        metakey = b
    }
    {
        comment = huhu
    }
}
```

The advantage of TCL style lists is that also arbitrary metadata can be embedded in a natural and distinguishable style.

XML

The XMLTOOL PLUGIN supports a verbose *XML representation*:

Listing 4.6: Elektra's traditional XML Representation

```
<keyset xmlns="http://www.libelektra.org"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.libelektra.org␣elektra.xsd"

    parent="system/filesystems">


    <keyset parent="mediacdrecorder">
        <key basename="device">
            <value>/dev/hdc</value>
            <comment>The device or label to be mounted</comment>
        </key>
    </keyset>
</keyset>
```

The same information can be presented much shorter:

Listing 4.7: Proposal for other XML Representation

```
<system>
    <mediarecorder>
        <device comment="The␣device␣or␣label␣to␣be␣mounted">/dev/hdc</device>
    </mediarecorder>
</system>
```

Both XML formats have a problem deciding if a key has a null value or an empty value. The proposed XML format additionally has the problem of deciding if a key exists at all. The null filter plugin can help on that issue. It needs to fill in `@NULL` or `@EMPTY` if the key exists but does not have a value.

So if the key `mediarecorder` is actually present, but has an empty value, the example would be changed to:

```
<system>
    <mediarecorder>@EMPTY
        <device comment="The␣device␣or␣label␣to␣be␣mounted">/dev/hdc</device>
    </mediarecorder>
</system>
```

An encoding plugin must also take care of characters not allowed to occur in XML. A CDATA section supports nearly all characters, but not the sequence `]]>`, so CDATA just changes the problem to another sequence to escape.

C code

The C code which creates a new `KeySet` is another possible representation for a storage. It looks like:

```
ksNew (30,
    keyNew ("system/key",
        KEY_VALUE, "value",
        KEY_END),
    keyNew ("system/testkey",
        KEY_META, "metaname", "metavalue",
        KEY_END),
    KS_END);
```

`ksNew()` introduces a new `KeySet`. `keyNew()` allocates a new `Key` that will be inserted in the key set. `keyNew()` has an arbitrary number of arguments, each starting with a keyword.

4.4 Pluggable Checker

4.4.1 Introduction

PLUGGABLE CHECKER are plugins which are executed before the storage plugin writes out the configuration. They make sure that the configuration is valid, consistent and complete. Pluggable checkers assure that problematic configuration is never passed to the storage plugin.

Types

Types are a defense shield against wrong interpretation in programming languages. On the other hand, in a key database it is only possible to ensure that the value stored is in a given range and fulfils specific properties or validation criteria. But the interpretation itself is done in the programming language. A mapping between the values in the key database and the values of the variables of the programming language is needed.

Types of a `Key` describe a set of allowed instances. Elektra stores such instances in the value. They should be passed to a program in a safe way. Elektra stores type information to check `Key` objects within the key. Metadata provides the place to store that information. The checking itself is done in `kdbSet()` by a pluggable checker. This guarantees that no invalid data is stored in the key database. The information on how to check a key may be stored in the key database, so that it is persistent and available when a value is changed the next time.

Two-Phase Checking

Elektra works completely without integrated type checking on keys, but provides various ways to do checks in a flexible way. All these checks rely on metadata. To provide maximum flexibility, the applying and checking of types is separated. This establishes a *two-phase type checking*.

In a first phase, concrete dynamic types are applied as metadata to keys. Also in this step, optional structure checks can take place. Structure checking involves inspecting the key names and allows definitions of interrelations between keys. The plugin doing these checks can overhaul if a specific child, sibling or parent `Key` is present.

In a second phase, checks given by metadata will be executed. Checker plugins can also use metadata from the storage applied by the user. These two steps, however, are independent from each other and have their own use cases.

4.4.2 Checker Plugins

CORBA Types

A common and successful type system happens to be CORBA with IDL[43, 14]. The system is outstanding because of the many mappings it provides to other programming languages.

The TYPE CHECKER PLUGIN supports all basic CORBA types: **`short`**, `unsigned_short`, **`long`**, `unsigned_long`, `long_long`, `unsigned_long_long`, **`float`**, **`double`**, **`char`**, `boolean`, `any` and `octet`. In Elektra `octet` is the same as **`char`**. When checking `any` it will always be successful, regardless of the content. Elektra also added other types. `empty` will only yield true if there is no value. `string` allows any non-empty sequence of octets.

Sometimes the type should expresses that, for example, both an empty or another type is valid. In Elektra a space-separated list of types expresses that. If any of those types match, the whole type is valid. For example, the type `string empty` equals the type `any`. This facility builds a union of the sets of instances existing types specify.

`enum` works with a list of choices. Any of these choices confirms to the type, others do not. For example, the type `FSType` accepts all file system names, for example, `ext2`, `jfs` or `vfat`.

The CORBA type system goes far beyond these basic types. In IDL, it is also allowed to define classes, interfaces and generic containers. These user-defined types are however not useful on a single `Key`.

To sum up, many basic types like **`int`** or **`char`** are convenient and CORBA ensures that they can be converted to the specific type of the programming language. The CORBA type system also has its limits. The types `string` and **`enum`**, however, can be unsatisfactory. While string is too general and makes no limit on how the sequence of characters is structured, the enumeration is too finite. For example, it is not possible to say that a string is not allowed to have a specific symbol in it. The next subsections introduce alternative ways to check values.

Validation Check Plugin

The VALIDATION PLUGIN gives a powerful tool to check strings using regular expressions. They provide a way to reduce the set of possible values for a key value.

The validation plugin looks for two metakeys. `check/validation` gives a regular expression to check against. If it is present, `check/validation/message` may contain an optional humanly readable

message that will be passed with the error information.

The implementation consists of a loop checking for every key if it has the mentioned metakey. The check itself is done by the POSIX regular expression library with the interface `regcomp`, `regexec`, `regerror` and `regfree`. The flag `REG_EXTENDED` is passed so that the regular expression will be compiled as an extended regular expression. `REG_NOSUB` gives a better perfomance and subexpressions cannot be used in this setup anyway.

The plugin also exports the function `ksLookupRE()` that does a lookup in a `KeySet` using a regular expression. It starts from the current cursor of the `KeySet` and stops when the first value matches. Finally this key is returned.

Network Check Plugin

While, in theory, a regular expression can express if a string is a network address, in practice, such an attempt does not work well. The reason is that an unmanageable number of valid shortenings for IPv6 addresses makes the regular expression hard to write and understand.

So the idea of building such a complicated regular expression was discarded, but instead a dedicated checker was introduced. The idea is to use the operating system facilities to resolve the network address. If this succeeds, it is guaranteed that this network address will be valid when it is resolved by the same interface afterwards.

Many network address translators coexist. In POSIX.1-2001 a powerful address translator is provided with the interface `getaddrinfo()`. It is a common network address translation for both IPv4 and IPv6. We used it to implement the NETWORK PLUGIN.

Path Check Plugin

The motivation to write this plugin is given by the two paths that exist in `/etc/fstab`: the device file and the mount point. A missing file is not necessarily an error, because the device file may appear later when a device is plugged in and the mount point may be there when another subsequent mount was executed. So only warnings are yielded in that case. One situation, however, presents an error: Only an absolute path is allowed to occur for both device and mount point. When checking for relative files, it is not enough to look at the first character if it is a `/`, because remote file systems and some special names are valid, too.

If the metakey `check/path` is present, it is checked if the value is a valid absolute path. If a metavalue is present, an additional check will be done if it is a directory or device file.

4.4.3 Apply Metadata

Glob Plugin

The GLOB PLUGIN provides coping metadata given by the plugin's configuration to keys identified using GLOB EXPRESSIONS. Globbing resembles regular expressions. They do not have the same expressive power, but are easier to use. The semantics are more suitable to match path names.

- ”*” matches with any key name of just one hierarchy. This means it complies with any character except slash or null.

- ”?” satisfies single characters with the same exclusions.

- Additionally, there are ranges and character classes. They can also be inverted.

The glob plugin iterates over a list of glob expressions for every key. Metadata is applied only for the first expression that matches. So later expressions can be used as default values.

Glob statements are very useful together with contracts. Storage plugins can request the glob plugin to fill up metadata before they receive the keys in `elektraPluginSet()`. In `config/needs`, the plugin declares which keys should obtain which metadata. If the glob expression starts with a slash, the contract checker will automatically prepend the mount point.

For example, the hosts plugin contract contains:

Listing 4.8: Contract for Glob

```
keyNew ("system/elektra/modules/hosts/config/needs/glob/#1",
    KEY_VALUE, "/*",
    KEY_META, "check/ipaddr", "", /* Preferred way to check */
        /* Can be checked additionally */
    KEY_META, "check/validation", "^[0-9.:]+$",
    KEY_META, "check/validation/message",
        "Character␣present␣not␣suitable␣for␣ip␣address",
    KEY_END),
keyNew ("system/elektra/modules/hosts/config/needs/glob/#2",
    KEY_VALUE, "/*/*",
        /* Strict character validation */
    KEY_META, "check/validation", "^[0-9a-zA-Z.:]+$",
    KEY_META, "check/validation/message",
        "Character␣present␣not␣suitable␣for␣host␣address",
    KEY_END),
```

We see that the `hosts` plugin adds two glob statements with the clause `config/needs`. The first one matches with hostnames, the second with aliases.

The glob plugin only fills the metadata in `kdbSet()`. This makes a difference compared with adding the metadata already in `kdbGet()`. Using the glob plugin, the user will not see the metadata, but later plugins in `kdbSet()` will.

To sum up, the glob plugin replenishes the keys with metadata. The plugin applies metadata in a flexible way. This metadata can be used for later checks. Limited configuration storage plugins, like the `hosts` plugin, use this feature. They need it because they are not able to store metadata themselves. It is obviously not possible to apply values to non-existing keys.

Structure Checker

The glob plugin together with the check plugins create a good combination to check keys which are present in the `KeySet`. For some storage plugins like fstab, however, missing keys are as fatal as not validating keys – it is not possible to write a valid configuration file without them.

The problem can be described as a structure built with lists and other structures that must match with the key names of a `KeySet`. If a structure must have an element, contrary to glob, the plugin can yield an error if it is missing. During the matching, the plugin also applies metadata to the individual keys.

Such plugins that check the structure and interrelations of keys are called *structure checker*. They can require that various subkeys have to or must not exist. This can happen recursively to specify any structure.

The *struct plugin* implements such a behaviour. It allows enforcement of a *strong consistency* within the keys of one backend. It needs a plugin configuration to know which structure it should check for. This configuration can be passed from a storage plugin's `config/needs` clause. For example, the contract for fstab looks like:

Listing 4.9: Contract of fstab

```
keyNew ("system/elektra/modules/fstab/config/needs/struct",
    KEY_VALUE, "list␣FStab",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab",
    KEY_META, "check/type", "null␣empty",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/device",
    KEY_META, "check/type", "string",
    KEY_META, "check/path", "device",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/mpoint",
    KEY_META, "check/type", "string",
    KEY_META, "check/path", "directory",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/type",
    KEY_META, "check/type", "FSType",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/options",
    KEY_META, "check/type", "string",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/dumpfreq",
    KEY_META, "check/type", "unsigned_short",
    KEY_END),
keyNew ("system/elektra/modules/fstab/config/needs/struct/FStab/passno",
    KEY_META, "check/type", "unsigned_short",
```

```
KEY_END),
```

The key value of `needs/struct` within the plugin configuration marks the starting point. "`list`" describes the first structure to be generated. It is a built-in structure of the struct plugin that supports all subkeys in one level. It applies to every direct subkey the structure check received by the *template parameter*. The template parameter is, in this case, `FStab`. The rest of the configuration specifies how entries of `FStab` must look.

The information applied to the keys is given through metadata. This metadata is copied to each key during the structure check. If, however, a key is missing, the structure check will terminate with a failure. Any additional key will also lead to an error.

The metadata may be evaluated by subsequent checks. In the situation of `fstab` a type checker and a path checker are both useful.

This approach for defining the structure works recursively. Every element can have a value with a new structure check. Additionally, multiple template parameters can support even more generic data structures. This is, however, not yet implemented.

The purpose of such structure checks is that only a valid configuration can be stored and that neither applications nor storage plugins are surprised by configuration they do not understand.

The fstab plugin trusts the fact that no invalid configuration is passed. It does not check it again. Missing configuration would lead to partially set data structures. The internally used API `setmntent` crashes in that case. This leads us to the *purpose of contracts*: We want a guarantee that specific conditions are already met because we know that the code of the plugin cannot handle it.

Let us look at a different scenario with the same configuration. Instead of using the fstab plugin, we will use a general purpose storage plugin[5]. No plugin exports a `config/needs` clauses for the struct plugin in this situation. But the user can add the struct plugin and the plugin configuration, as shown in Listing 4.9, to the backend manually. Applications still can be sure that only a specific configuration will be stored and passed to them. The unwritten contract is between the application and the backend. No contract checker, however, would detect the missing configuration.

[5]For example, the dump plugin. Note that the metadata will be stored permanently in this situation.

CHAPTER 5

Evaluation

5.1 Development Time

In this section we will evaluate how the modular approach can affect development time.

5.1.1 On Using Configuration Libraries

XML

We implemented a plugin for reading and writing XML as described in Section 4.3.4 on page 76. XML has grown to a rather complicated standard and it is not easy to write an XML parser. So we decided not to implement the parser, but to use the XML library `libxml2`[1].

The effort to implement this storage plugin was still significant. We needed several days to implement it. The code allocated many XML elements and attributes. The problem was that all of them needed to be freed properly. That was not done in the first implementation. So it took additional development time to make sure that the code has no memory leaks after running.

INI Configuration Libraries

Dozens of other libraries, that can help for storage plugins, exist. For example, appconf, CFL, simpleXMLconf, OSSP, cfg, varconf, ConfigLib, UT_var, Nickel, SimpleIni, rudeConfig, libINI, RudeConfig are some of the INI-like parsers available.

We investigated these parsers and decided in favour of Nickel as the basis for a case study. It is as portable as Elektra needing only the standard C99. The licence, called zlib, is similar. Nickel is fast with a reasonable memory footprint and can parse many variants of INI files with sections including the KDE variants[2]. The disadvantage of Nickel is, however, that it discards comments.

[1] The XML C parser and toolkit of Gnome.
[2] We evaluated this with a test reading in a typical KDE configuration.

Case Study: Nickel

The implementation of a proof-of-concept NI PLUGIN using Nickel took slightly more than *one hour*. Half of this time was spent integrating Nickel into the build system. Arbitrary strings work out-of-the-box gracefully even without any *code plugins*.

The ni plugin immediately profits from the Nickel's error handling. Checking some return values is all that is needed. To sum up, no other way of implementing storage plugins yields so many features in so short a time.

Conclusion

All these advantages also apply if an application directly uses a configuration library. But the advantage of Elektra is that the resolver plugin determines the file name and detects conflict and update issues. But more than this, many other existing plugins can extend the overall functionality of the backend. Because of the dynamic mounting properties of Elektra, it is possible to change the backend for specific operating systems or use-cases as needed for resolving file names.

We ascertained that storage plugins can benefit a lot from existing configuration libraries. We found out that the API design of these libraries significantly influences the needed time to write a storage plugin.

5.1.2 On Using Grammars

Writing Grammars

Typically, hand-written parsers are used in Elektra to implement storage plugins. We implemented some, for example, for parsing `hosts` and `fstab` configuration files. These files can be considered as simple configuration files because they do not have nested sections in them and do not support arbitrary metadata.

Nevertheless, writing a parser for them took considerable time and effort. It took about one day to implement each of these storage plugins. On the other hand, to preserve the ordering of the items in the `/etc/hosts` configuration file, as presented in Section 4.3.3 on page 73, we only needed some hours. But for other "simple" changes like preserving white spaces a rewrite of the parser would be necessary.

Case Study: TCL Lists

We developed the TCL Lists storage plugin with rapid application development. Instead of parsing the content ourselves we used `spirit::qi`. It generates the code that parses the configuration file. We only had to write the grammar and functions that fill up the data structures during parsing. The main problem was to get used to the `spirit::qi` library. Once we understood the concepts it was easy to actually write the code. The generated parser is much more powerful than our hand-written code for `hosts` or `fstab` storage. The structure of the file is nested and it supports metadata. Nevertheless, we needed about the same time to implement it.

The inline grammar specification is short and gives a good idea of what the configuration file looks like:

Listing 5.1: Grammar for TCL Lists

```
query =  '{' >> pair >> *(pair) > '}';
pair  =  '{' >> key > '=' >> val >>
    *('{' >> metakey > '=' >> metaval > '}') >
    '}';
```

The specification of the `key`, `val`, `metakey` and `metaval` also binds some functions to be called by the parser:

```
key     =  (+(qi::char_ - qi::char_("={}[]<>")))
    [boost::bind(&Printer::add_key, &p, _1)];
val     =  (+(qi::char_ - qi::char_("={}[]<>")))
    [boost::bind(&Printer::add_val, &p, _1)];
metakey =  (+(qi::char_ - qi::char_("={}[]<>")))
    [boost::bind(&Printer::add_metakey, &p, _1)];
metaval =  (+(qi::char_ - qi::char_("={}[]<>")))
    [boost::bind(&Printer::add_metaval, &p, _1)];
```

The code for the functions themselves is trivial. For example to add a metakey:

```
void Printer::add_metaval(std::vector<char> const& c)
{
    std::string s(c.begin(), c.end());
    current.current().setMeta<string>(metaname, s);
}
```

Because we used a white space skipper it was not possible to store the white spaces in metadata.

Result

An unexpected disadvantage is the additional compile time when using this approach. `spirit::qi` builds up the parser at compile time using C++ meta programming techniques. Compiling the source file with the grammar takes about[3]:

```
4,96s user 0,36s system 99% cpu 5,379 total
```

This fact was annoying during development with short compile-test cycles.

In fact, a storage plugin using grammar is not necessarily implemented in a shorter time. Especially not, if developers do not use grammars daily. We rewrote the grammar once so that the configuration file looks more like TCL lists. The rewrite of the grammar, however, was done within minutes. We concluded that the setup and the learning of the syntax was the problem why it took so long the first time.

To sum up, it depends on the configuration file which technique should be preferred. But mainly the skills of the developers influence which variant should be chosen. The advantage of the grammar is that the specification of the configuration file format can be modified easily without changing the other code around it. For other changes, such as remembering white spaces, both variants can have their problems.

[3] Measured with the setup as described in Section 5.3.1 on page 89 using the tool `time`.

5.2 Modularity

In this section, we will evaluate how modularity affects the overall flexibility, development time and other properties of the system. In the Elektra situation of version 0.7, every feature needs to be implemented in each backend. The modular approach allows us to implement every feature in a dedicated plugin. To evaluate the difference, we developed about twenty plugins as described in the Chapter 4.

Then we tried out in some scenarios whether the different combinations of the plugins work together. We present the results in the following subsections.

5.2.1 Separation of Non-Portable Code

We justify the decision of splitting up the resolving of the file name and the other storage activities with the statement that it is possible afterwards to write portable storage plugins. We implemented seven storage plugins.

During evaluation, we found out that six of these plugins work completely portably. They only use C99 or ISO/IEC C++ 2003. They do not need any operating system dependent code. The exception is the `fstab` plugin that uses the BSD 4.3 specific `setmntent` interface. It is not portable for that reason.

We conclude that our initial promise, that it will be possible to write portable storage plugins, could be proved.

Code Plugins

The code plugins allow transparent encoding and decoding of all values in a key set. Using contracts, storage plugins can pass a configuration to the code plugins to control which characters should be escaped. We implemented two code plugins: hexcode and ccode plugin. We tested both plugins in many combinations with the storage plugins. As expected, the encoded values were written into the configuration files.

Storage plugins that exported configuration worked with both ccode and hexcode plugins without any changes in the code. We concluded that it is possible to transparently exchange the code plugins with the effect that the configuration file will be encoded differently but without any changes to the application.

5.2.2 Pluggable Types

Elektra does not have a built-in type system. Instead, we implemented some plugins that check the keys according to metadata.

Two-Phase Approach

We learned that for key databases it is useful to determine the type of a specific `Key` object first and then check if the value conforms to it. We have had success in separating these two tasks in two different sets of plugins. Even though metadata is applied to the keys, we managed to make this metadata invisible

to the user if the storage plugin discards it. Typical administrator mistakes are now caught with this technique.

For example, we tested storing host names with invalid IP addresses:

```
> kdb set user/hosts/new_hostname 192.168.0.333
create a new key user/hosts/new_hostname with string 192.168.0.333
a key was thrown
number: 51
description: Value of key is not a valid IP Address
ingroup: plugin
module: network
at: /home/markus/Elektra/current/src/plugins/network/network.c:120
reason: name: user/hosts/new_hostname value: 192.168.0.333
        message: Name or service not known
1 Warnings were issued
warnings/#00: number description ingroup module file line function reason
number: 36
description: could not unlink file
ingroup: plugin
module: resolver
at: /home/markus/Elektra/current/src/plugins/resolver/resolver.c:326
```

The network plugin correctly detects that this IP address is invalid and refuses to write it out. We also see the warning that it was not possible to unlink the file because the storage plugin did not even write the temporary file, so the resolver could not remove it.

To sum up, Elektra's *pluggable type checker* uses a two phase approach. In the first phase, a plugin assigns built-in or basic types to keys using metadata. In the second phase, the actual checks are accomplished. Elektra actually prevents committing any data that does not conform to given types or refused by other checkers.

Error Plugin

The error plugin is also a checker plugin. Like the other checker plugins, it will only do anything when specific metadata is present. But unlike the other checker plugins, no key value will be successfully validated. Instead, the error plugin will always return an error status. With the error plugin, it is possible to trigger an error state with any desired error information on purpose.

The resolver plugin creates a temporary file in the beginning of the `kdbSet()` algorithm. When it gets its chance, it will move the temporary file on success or remove it on error. So both on commit or rollback the temporary file should be gone if the algorithm works correctly.

During tests, the temporary file was always removed in both situations: When `kdbSet()` was successful, but also when the error plugin triggered an error. We concluded that the exception safety works as described in Section 3.3.3 on page 51.

Type Checking

We implemented a type checker that is both able to check the structure of the configuration and the type of the specific values.

Storage plugins like fstab use this type checker. Because of previous tests, we knew that the fstab plugin crashes when not well-structured configuration is passed. But together with the type checker no more crashes occurred, but the system now yields a detailed error message that the structure of the configuration does not look like the contract of the fstab plugin described.

Because the contract describes the structure and types in a general way, we concluded that other plugins can make use of the same checks, too.

We also tried to use the structure checker with dump. This storage plugin does not limit the type and structure of the configuration by itself. We tried to apply the same plugin configuration as the fstab plugin exports for the struct plugin. The result was that we were not able to store any configuration that would not conform to fstab configuration into the backend using the dump storage.

We also learned, however, that the fstab plugin has some semantic differences. It does not remember the name of a specific entry. That means that the fstab storage does not fulfil the *round-tripping* properties. So, in fact, the current implementation of the fstab storage cannot be considered as practically usable.

We conclude that applications can make use of type checking. If they have specific requirements what the configuration must look like, they can add the struct plugin together with their own plugin configuration. Note that this approach improves the situation compared to an application that checks configuration while reading. With type checking, it is possible to avoid that the malfunctioning configuration is even stored.

5.2.3 Cross-Cutting Concerns

We have already learned that Elektra now supports various cross-cutting concerns in separate plugins. We will evaluate how well this works. Our test setup was the `kdb` tool with different backends mounted. Some backends have only syslog, some only D-Bus, some none and some both plugins inserted.

During tests we saw that in all scenarios the correct side effects took place. We found out that it is easy for the administrator to create a new backend with the desired cross-cutting concerns supported.

When applications can run post-install scripts, they can also create their backends with the desired side effects during installation. Our conclusion is that this system works in a flexible manner and supports cross-cutting concerns of applications and administrators well.

5.2.4 Conclusion

We evaluated different combinations of filter plugins, type checking plugins and cross-cutting concern plugins with storage plugins. We asked ourselves if the system is also capable of handling any combination of these plugins.

So we tested a backend with dbus, syslog and validation plugins at once. It successfully logged and notified changes of the persistent configuration. When the validation failed, the rollback was successfully

logged, but no notification was sent. Each of these plugins exactly fulfilled its purpose as before. They did not influence each other.

Even struct and glob plugins can be used together. In that case, however, some metadata might be overwritten by the respective other plugin. This combined usage of plugins support many more possibilities than the resolver, storage and one other plugin as described before.

In this thesis, 9 plugins related to filtering and cross-cutting concerns were implemented. 4 plugins can validate configuration[4]. 7 plugins actually read and write configuration. Using a storage plugin with another plugin yields $13 \cdot 7 = 91$ possibilities to form a backend. It would not have been possible for us to add all these features to all of the 7 storage plugins.

But the approach is even more powerful because it allows us to use more than one additional plugin. If we take 4 plugins out of the pool of 13 plugins, we have $\frac{13!}{4!(13-4)!} = 715$ ways to enrich the 7 storage plugins with additional features.

5.3 Run Time

In this section, we will evaluate how Elektra and its plugins will change the run time behaviour of applications. Because of the massive changes in both algorithm and features, only the most recent version at the time of writing this thesis, will be studied. It is not compared with any previous versions because they lack, for example, updating keys and detecting conflict situations.

5.3.1 Setup

Hardware setup

The processor to run the tests is an AMD Athlon(tm) 64 X2 Dual Core Processor 3600+. It has a total of 5980 Megabyte free memory space as measured with the tool `free`. The hard disks run in a software RAID 5[5] array. On top of the RAID 5 array, LVM[6] supports flexible partitions formatted with jfs. The output of `hdparm` version 8.9 gives a conclusion concerning the throughput of data on these LVM partitions:

```
Timing cached reads:   1946 MB in  2.00 seconds = 973.97 MB/sec
Timing buffered disk reads:  618 MB in  3.01 seconds = 205.59 MB/sec
```

This LVM partition was used to store the configuration as described later.

Software setup

The software setup is basically completely defined by Debian Lenny 5.0.6. The only relevant exception is the usage of an older kernel:

[4] The two plugins that apply the metadata will help them.
[5] redundant array of inexpensive disks
[6] Logical Volume Manager

```
Linux version 2.6.24-1-amd64 (Debian 2.6.24-7) (dannf@debian.org) (gcc
version 4.1.3 20080114 (prerelease) (Debian 4.1.2-19))
#1 SMP Sat May 10 09:28:10 UTC 2008
```

Nevertheless, we will provide a list of version numbers of the software that influences the results:

- Valgrind version valgrind-3.3.1-Debian
- gcc (Debian 4.3.2-1.1) 4.3.2
- cmake version 2.6-patch 0
- GNU Make 3.81
- GNU libc 2.7

Tested Program and Library

All tests run with Elektra $0.8mile4$ that was developed in the thesis. The compile options vary and are explained in the respective places.

The tested program was written specifically for the benchmarks. The C macros were set as listed below:

```
#define KEY_NAME_LENGTH 1000
#define KEY_ROOT "user/benchmark"

#define NUM_DIR 100
#define NUM_KEY 100
```

The time measurement code is presented below:

```
struct timeval start;

void init_time (void)
{
    gettimeofday (&start,0);
}

void print_time (char * msg)
{
    struct timeval measure;
    time_t diff;

    gettimeofday (&measure, 0);

    diff = (measure.tv_sec - start.tv_sec) * 1000000 + (measure.tv_usec - start.tv_usec);
    fprintf (stdout, "%20s: %20d Microseconds\n", msg, (int)diff);

    gettimeofday (&start,0);
}
```

The code below generates the key names that will be stored and retrieved:

```
int creator (KeySet *large)
{
    int i,j;
    char name [KEY_NAME_LENGTH + 1];
    char value [] = "data";

    for (i=0; i< NUM_DIR; i++)
    {
        snprintf (name, KEY_NAME_LENGTH, "%s/%s%d", KEY_ROOT, "dir", i);
        ksAppendKey(large, keyNew (name, KEY_VALUE, value, KEY_END));
        for (j=0; j<NUM_KEY; j++)
        {
            snprintf (name, KEY_NAME_LENGTH, "%s/%s%d/%s%d", KEY_ROOT, "dir", i, "key", j);
            ksAppendKey(large, keyNew (name, KEY_VALUE, value, KEY_END));
        }
    }

    return 1;
}
```

Listing 5.2: Benchmark program

```
int main()
{
    KeySet * large = ksNew(NUM_KEY*NUM_DIR, KS_END);

    init_time ();

    creator (large);
    print_time ("New large keyset");

    Key *key = keyNew (KEY_ROOT, KEY_END);
    KDB *kdb = kdbOpen(key);
    keySetName (key, KEY_ROOT);
    print_time ("Opened key database");

    KeySet *n = ksNew(0);
    kdbGet(kdb, n, key);
    ksDel (n);
    print_time ("Read in key database");

    kdbSet(kdb, large, key);
    print_time ("Write out key database");

    kdbClose(kdb, key);
    print_time ("Closed key database");

    ksDel (large);
    keyDel (key);
    return 0;
}
```

5.3.2 Size of the Library

On the 64bit setup as described above, the size of the library $libelektra.so.0.8.0mile4$ is:

- The library needs 192 kilobytes compiled in debug mode without optimisations.

- With the debug symbols removed[7] the size of the program is reduced to 93 kilobytes.
- The library needs 96 kilobyte compiled in release mode.
- With the debug symbols removed the size of the program is reduced to 85 kilobyte.

Although it occupies only little space it supports all the features as presented in this thesis.

5.3.3 Dependences of the Library

The library $libelektra.so.0.8.0mile4$ does have the following dependences determined by the program `ldd`:

```
linux-vdso.so.1 =>  (0x00007fff87bfe000)
libdl.so.2 => /usr/lib/debug/libdl.so.2 (0x00002af9230ec000)
libc.so.6 => /usr/lib/debug/libc.so.6 (0x00002af9232f1000)
/lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

These are the typical dependences for a library that only needs libc and uses dynamic linking. No additional dependences are needed in Elektra's core even though it allows us to accomplish the tasks described in this chapter.

5.3.4 Benchmarks

Preparation

In `user/benchmark`, the following plugins were mounted: resolver, dump, null, hexcode, glob and type. None of the additional plugins changes anything because no umlauts, no types, no empty values and so on are present in the keys. We stored ten thousand keys in `user/benchmark` already before running the tests using the same program given in Listing 5.3.1. During the tests the load of the system was 2.39 2.94 2.73 6/158 14296, as given by `/proc/loadavg`. Some programs like a web browser, editor and similar were running to give a daily use scenario.

Time Measurements

First we will test the code compiled without optimisations in debug mode and with twelve backends mounted. Below is the output of the time measurements of the program as given in Listing 5.2 describing how long the typical actions take for this setup:

```
       New large keyset:              75689 Microseconds
    Opened key database:              41640 Microseconds
   Read in key database:             257556 Microseconds
Write out key database:              325842 Microseconds
    Closed key database:                737 Microseconds
```

[7] Using the program `strip`.

In the second test we compiled Elektra in release mode. The output of the program is:

```
      New large keyset:               76016 Microseconds
   Opened key database:               11953 Microseconds
  Read in key database:              207632 Microseconds
Write out key database:              300069 Microseconds
   Closed key database:                 665 Microseconds
```

We see that – although the number of keys is high for a single application, a text configuration file is used, and many useless actions are done – the whole task is barely noticeable for the user. Time measurement, however, is very inaccurate and it would be difficult to determine the actual costs of the single functions.

So for comparison between functions, the tool CALLGRIND from the VALGRIND suite was used. Callgrind determines the costs by counting instructions[42]. It is well suited to compare the cost of individual functions.

Instructions Measurements

The whole benchmark program as shown in Listing 5.2 takes 922 million instructions, which is divided by `kdbSet()` needing 437 million instructions and `kdbGet()` needing 377 million instructions.

`kdbGet()` already has a surprisingly good ratio with nearly all the costs devoted to the `dump::unserialize` function needing 316 million instructions. It is, however, feasible that `dump::unserialize` is inefficient. The rest is devoted to splitting the key set with 11 million and merging with 32 million instructions.

The hexcode plugin with 4.6 million instructions presents the additional plugin with the highest costs.

Compared with other costs, resolving is cheap with only 165 thousand for opening and less than 20 thousand instructions for the other operations.

The null checker with nearly 1.7 million instructions has only a fraction of the hexcode's costs.

The type checker is another major player in terms of resources. It needs 252 million, but only because `Key::getMeta` always throws an exception when no type information is found: exception handling of the type checker costs 223 million instructions. These are some spots where optimisations certainly can be worthwhile.

The glob plugin only needs about 800 thousand instructions. `elektraTrieLookup` also has high costs with about 18 million instructions.

5.3.5 Conclusion

In the current situation, additional plugins have only a fraction of the overall costs. Plugins with side effects that do not iterate over all keys are, as expected, no problem in terms of time penalty. The overhead of single plugins that only iterate over all keys is also insignificant. We conclude that it presents no problem to split up all cross-cutting concerns in different plugins.

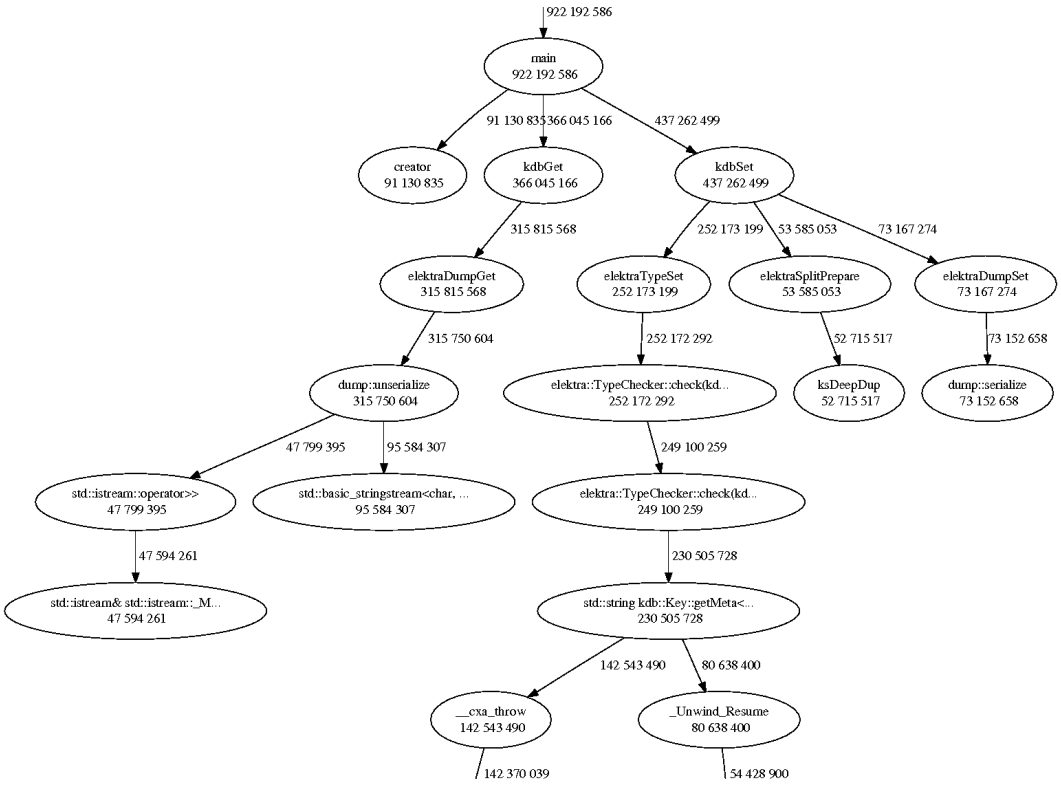

Figure 5.1: Callgraph of the benchmark program

But plugins that change all values of keys should only be used with care. Ideally, only a single plugin should be concerned with all or most encoding issues. But changing all values still costs only 1% of the overall instructions in the example.

As desired, the actual writing and reading compose the largest part. We also see that some programming techniques like throwing exceptions must be reserved for aborting operations. We concluded that the current use of exception in the type plugin is not appropriate.

5.4 Contracts

Contracts play a critical role in the interplay of plugins. We will evaluate how well `kdb mount` works and where the limits of contracts are by describing the lesson learned during the work.

5.4.1 Contracts Checker

The contract checker includes checking of a single contract without any relation to other contracts with `kdb check`. We implemented checks for a well formed, correct and full contract.

These checks were useful during development. When we forgot a clause, we immediately got feedback that something was missing without the need to actually try to mount the plugin. The same was true when we forgot to export a symbol, which occurred quite frequently.

We found out that these checks were especially valuable when we introduced new clauses. With the checker tool for validating a single contract, it was possible with surprising facility to find the plugins that were not up to date or had obvious problems in the contract.

We conclude that checking single contracts can be useful during development of a plugin.

5.4.2 Automation

Currently `kdb mount` works in an interactive way. The user can decide which plugins should be added. The algorithm has some automatism to find out where the plugin should be placed. We will evaluate how well this works.

In Section 5.1 on page 83, we described many combinations we used to add plugins. For all combinations, the algorithm automatically found a way to insert the plugins in the correct place. Currently the algorithm, however, has some limitations.

For example, we inserted the type plugin. Afterwards we tried to insert the struct plugin. The algorithm does not know that these two plugins just need to be reversed in order to fulfil the contract. Instead it will output that an ordering validation happened. The user needs to correct the problem by providing the plugins the other way round.

To sum up, the current contract checker gracefully rejects all non validating contracts. In some situations, however, the user needs to manually fix the problem, because the contract checker does not have an automatic solution.

5.4.3 Limits of Contracts

Contracts can be valuable in a specific domain. It is no problem to describe that a specific encoding is required, in which order it is needed, and if it needs to be reversed or not.

But the power of contracts is soon exceeded when we try to express general statements out of the context of configuration. Fortunately, such statements were not needed for Elektra. The existing contracts describe well what changes are applied to a `KeySet` object – the main task of plugins in Elektra.

We also learned that repairing the data structure so that conditions are met is far more often possible than initially supposed. Only configuration not conforming to a given type will yield a run time error to the application. Problems with incorrect encoding and invalid characters can be hidden from the user. The user currently will, however, experience some differences in which metadata can be stored in a key database.

CHAPTER 6

Related and Future Work

6.1 Related Work

6.1.1 Pluggable Types

Pluggable types[32, 13, 3] have emerged recently as a field of research and describe how type systems can be optional. A pluggable type checker facilitates flexible checks that can be tailored towards the user's needs. Pluggable type checkers available today often extend an existing type system of a programming language with some attributes[32]. Instead, Elektra does not have a built-in type system.

Type annotations[15] express pluggable types in programming languages, but the syntax does not fit for key databases. Elektra uses metadata instead of type annotations.

CORBA Types as specified in IDL[43, 14] provide a full set of types with appropriate mappings to many programming languages. Elektra makes use of this ecosystem and has support for the basic CORBA types. The mappings to variables of programming languages are used inside the type checker plugin.

6.1.2 Error Handling

When errors occur, a decision must be made if the plugin is important enough to stop the request completely or to go on with the other succeeding backends. The Pluggable Authentication Modules, also known as PAM[37], have developed a method to decide that well. PAM uses the three control flags `required`, `optional` and `sufficient` for that. Elektra does not support control flags. Instead, the position of the plugin defines the semantics on how to proceed. Plugins up to the commit plugin are `required` and later plugins are `optional`. The control flag `sufficient` would make sense when support for multiple storage plugins arises.

6.1.3 Contracts

The plugins in modular backends need to fulfil contracts[25, 17] in order to work together correctly. Research on how to specify constraints in object-oriented database systems exists[18].

Refinement and Inclusion

The behavior of a plugin can be specialised and extended through contract refinement and a contract inclusion mechanism[21]. They allow us to create new contracts using previously defined contracts. So the contract language can be viewed as a plugin interconnection language. No such mechanism is currently implemented in Elektra.

Openness

Contrary to the black box approach proposed in [4], we believe that the availability of source code is needed to adopt good practices. For example, GStreamer separates its plugins into ”good”, ”bad” and ”ugly” plugins to appreciate the code and the licence. Only with source code and the right to copy and modify it, is it possible to develop similar plugins by starting with another plugin as a template. With that technique, writing of boiler plate code can be avoided. If programmers see that some plugins have the same code, these plugins can be refactored. A too restrictive licence prevents such actions.

We believe that the possession model is of rather limited use for Elektra because the adaption of plugins for specific use cases is rather the rule than the exception. We assume that many specific plugins will be delivered with the respective software.

6.2 Future Work

6.2.1 Core

OWNER in Elektra’s terminology is represented as metadata of a `Key`. It describes from which user’s home directory the `Key` was read. The metakey does not exist when the owner equals the current user.

The administrator might want to copy keys between different users. This is currently not possible because the resolver plugin always resolves for the current user and `Split` is not aware of the owner.

The problem does not affect users, because they should not access configuration of other users. Administrators can circumvent this problem by exporting the configuration from one user and importing it to another.

The owner concept is deeply integrated into Elektra’s way how to handle key names and many documents show how to use it. So it is considered to be implemented again in later versions.

6.2.2 Plugins

Changing Features at Run Time

Plugins, that take care of character escaping and encoding, are needed so that limited storage plugins work properly. Removing them at run time is not a good idea. But other plugins that provide validation, notification and logging may be turned off. This could be done with information passed to the respective plugin.

Another solution works completely without the cooperation of plugins, by removing the plugin from the chain of execution entirely. For example, if a notification daemon crashes, the affected plugins could be removed until the service is up again.

We need to handle different situations for plugins. Some plugins are not allowed to be removed because they are required. Some must stay because the administrator forces a specific policy. Other plugins, however, can be removed.

Multiple Storage

From high-availability systems a challenging requirement arises. How is it possible to make configuration storage redundant?

To achieve that, every backend needs to write out configuration multiple times. Such backends will achieve a live-backup for configuration. The usage of different storage plugins writing to different machines even increases the availability of the configuration.

To be more failure tolerant, we also have to improve the reading of configuration. Instead of stopping when the first storage plugin yields an error, the next one should be tried instead. We need a strict policy to ensure that these storage plugins always set an error if something goes wrong.

6.2.3 Transactions using Global Plugins

We will see in the following examples that some tasks in `kdbSet()` have to happen respectively before, or after any other event occurs. A way to achieve this is by a special backend. The advantage of this approach is that plugins can also be reused for these tasks. These plugins will be referred to as GLOBAL PLUGINS in this section.

Atomic Notification

Until now, each backend has its own notification plugin. This means, that for every backend that commits, the applications will be notified. Listeners to these events, however, want to know when every backend during a `kdbSet()` transaction is settled. A global plugin can notify that a sequence of events is finished. Only then is it the time for other programs to update their configuration.

Journalling

When a program or computer crashes, committing may abort unexpectedly. If some backends have already committed, while others have not, an unsatisfying situation arises. The premise that `kdbSet()` is executed fully or not is broken. But the program does not exist anymore and no further actions are possible. Journalling provides a way to fix the problem when the next `kdbSet()` by another process is executed.

It works by writing a journal log instead of directly moving files. Once the log is completely written and flushed to disk, a global plugin does the actual renaming of the files. When the program crashes in the first phase while writing the log, this is no problem because no configuration file has been changed up to that point. The unfinished log file has to be discarded. On the other hand, the program can replay the finished journal whenever the applications aborts at a later time. No intermediate state without knowing what needs to be done will occur.

Using git for configuration

Since the availability of efficient and decentralised revision control systems, it became usual to utilise them for configuration files. Not only that unintended changes can be reverted, but also the history of changes can be displayed and checked out. Moreover, different branches can be used to switch between different settings.

Thus Elektra works with text configuration files, *git* can of course be used even without the knowledge of Elektra. But Elektra can help the user by automatically adding all processed files to the index and commit them after a successful `kdbSet()`.

Even better, Elektra can write git object files on its own and do a git-checkout after this has been done. This would yield atomic properties like journalling. Additionally, the advantage is that it leaves the option to rollback, even on finished commits. Using git, instead of inventing a new type of objects files, allows the usage of the available and optimised git tools.

6.2.4 Concurrency

kdbDup

Elektra can be used in a thread-safe way by instantiating Elektra's data structures per thread. That means all functions in Elektra are reentrant. Elektra currently does not support thread-specific locking for its data structures.

To achieve higher performance for multithreaded applications, fast duplications of `KDB` are needed. If implemented correctly, *bootstrapping* Elektra multiple times could be avoided.

Concurrent Processing

The plugins must be processed in the correct sequence because they change keys and have side effects. The processing of backends is however completely independent up to, and after, the commit step. Each of

these two parts can be implemented concurrently. If any backend fails before commit, special care must be taken so that no backend commits its changes. The concurrent processing of the plugins is possible for both `kdbGet()` and `kdbSet()`.

6.2.5 Discarded Ideas

These ideas were discarded on purpose. They are likely to introduce unforeseeable problems and the problems can also be solved in another way. More research is needed before they can be implemented.

Handling Multiple Files per Backend

We decided not to implement any other situation than one file per backend. On the one hand, it is simple to mount multiple backends if more files are needed.

On the other hand, the complexity and inefficiency can increase drastically. Storage plugins would need `KeySet` operations like difference to deduce which files to remove or create. File name resolving would not be easy anymore. The previously trivial test if an update is needed or a conflict occurred may become slow depending on the number of files involved.

For example, KDE has many configuration files in the path `~/.kde/share/config`. In a KDE installation with many programs it can be up to one thousand individual files.

We decided that such an approach can be implemented if the current solution is known to be unable to handle it. More tests are needed to decide if such setups work in a satisfying manner with the current equipment. The disadvantage of the current situation is that new files are not tracked automatically. But this can be achieved by install scripts that automatically mount the files necessary for the configuration. To sum up, currently the correct way to insert additional files into the key hierarchy is to mount them.

User Mount Point Configuration

The mount point configuration resides only below `system/elektra/mountpoints`. We decided against providing `user/elektra/mountpoints` because of security concerns.

It must be assured that the user only mounts backends where he or she is allowed to do so. But these conditions might change. As mentioned for kiosk in Section 2.3.3 on page 28, the administrator may want to enforce some user configuration. But this would be pointless if the user could circumvent it.

Another open item is how, and if, user's own plugins can be used. Plugins provided by the user open a door for code injection. These problems have to be discussed and clarified before any attempts in this direction are made.

Distributing Configuration

Elektra currently does not provide any distribution or decentralised features because it does not have any plugins communicating with a configuration daemon. These features are supposed to be on a higher level and not in backend code. Instead, software like Cfengine[12, 11] and Puppet can use Elektra to change local configuration[10]. Augeas[28] was written specifically for that purpose. It is, however,

not intended to be a configuration library for applications. Its bidirectional approach, on how to read and write configuration using lenses[8], is interesting. Unfortunately, it is not capable of mapping more complicated configuration files, but actually it would be an interesting storage plugin for Elektra because it supports many simple configuration files.

CHAPTER 7

Conclusion

After an introduction describing what configuration is and what features modern applications expect, we introduced Elektra and showed how it tried to solve these problems and why it stagnated. Multiple plugins together in one of Elektra's backends solved all these problems well with the benefit that plugins were reusable. These plugins could be built together as the requirements presuppose. The additional flexibility and the separation of concerns comes at the price that more run time errors are likely. Therefore, we introduced contracts so that plugins can express the input they require and the output they offer.

We looked at many applications such as encoding, converting, escaping and hiding that can now be orchestrated within a backend safely because of contracts. We also learned that resolver plugins were in charge of the operating system-dependent problems such as determining the path to the configuration file, detecting conflicts and checking for updates.

The counterpart in functionality is the storage plugin, in contrast, characterised by being portable. It finishes the job the resolver plugin is not competent in: reading and writing to the configuration storage. Another duality is represented by the pluggable type checker consisting of two parts. The first part creates the information which checks have to be done. And the second part actually performs the investigation.

We introduced key database semantics which no longer try to resemble file system semantics, but are designed to hold configuration in simple key value pairs. To still denote specialities of configuration files, allow communication between plugins and let the user extend semantics or behaviour of the key database, we proclaimed metadata.

We learned a lot about the implementation: the data structures building the foundation and the algorithm orchestrating the plugins. We investigated a way how to guarantee Elektra's semantics and how to build up the internals that make sure that only keys from the correct backends are aggregated.

A detailed and comprehensive explanation of error handling and algorithms followed because they glue together all other parts. Afterwards, we evaluated what Elektra is capable of doing and where the limits are. In the related and future work, we enumerated the tasks yet to be done, hoping that researchers will try to answer these questions.

Bibliography

[1] A. Alkalay. Linux registry. Talk at KDE Community World Summit, August 2004.

[2] C. Amsüss. private conversation.

[3] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *ACM SIGPLAN Notices*, 41(10):74, 2006.

[4] F. Bachmann, L. Bass, C. Buhrman, S. Cornella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical concepts of component-based software engineering. Technical report, 2000.

[5] Waldo Bastian. XDG Base Directory Specification. `http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html`, June 2010.

[6] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™). `http://www.yaml.org/spec/`, October 2009.

[7] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, page 507. ACM, 2006.

[8] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 407–419, New York, NY, USA, 2008. ACM.

[9] EB Bruce and WG Daniel. Metadata standards and Metadata Registries: An Overview. In *The International Conference on Establishment Surveys II. Buffalo*. Citeseer, 2000.

[10] M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49(1-3):1–46, 2003.

[11] M. Burgess and A. Couch. Autonomic computing approximated by fixed-point promises. In *Proceedings of the First IEEE International Workshop on Modeling Autonomic Communication Environments (MACE), Multicon Verlag*, pages 197–222, 2006.

[12] M. Burgess et al. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3):309–402, 1995.

[13] B. Chin, D. Marino, S. Markstrum, and T. Millstein. Enforcing and validating user-defined programming disciplines. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, page 86. ACM, 2007.

[14] M. Ebner, A. Yin, and M. Li. Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In *Testing of communicating systems XIV: application to Internet technologies and services: IFIP TC6/WG6. 1 Fourteenth International Conference on Testing of Communicating Systems (TestCom 2002), March 19-22, 2002, Berlin, Germany*, page 443. Springer Netherlands, 2002.

[15] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 2008.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Professional Computing Series. Addison Wesley, 1995.

[17] C. Garion and L. van der Torre. Design By Contract. *Coordination, organization, institutions and norms in agent systems I*, July 2005.

[18] A. Geppert and K.R. Dittrich. Specification and implementation of consistency constraints in object-oriented database systems: Applying programming-by-contract. In *Proceedings. GI-Conference BTW*, 1994.

[19] P. Gühring. private conversation.

[20] Carsten Haitzler. Eet library documentation. `http://docs.enlightenment.org/api/eet/html/`, 2008.

[21] I.M. Holland. Specifying reusable components using contracts. In *ECOOP'92 European Conference on Object-Oriented Programming*, page 287. Springer, 1992.

[22] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[23] S. Josefsson. The base16, base32, and base64 data encodings, 2003.

[24] M.F. Krafft. *The Debian system: concepts and techniques*. Open Source Press, 2005.

[25] M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76.

[26] Simon Law and Patrick Patterson. UniConf, GConf, KConfig, D-BUS, Elektra, oh my! `http://alumnit.ca/wiki/attachments/uniconf_universal.pdf`, 2005. Desktop Developers' Conference.

[27] R. Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[28] David Lutterkort. Augeas - a configuration API. `http://www.kernel.org/doc/ols/2008/ols2008v2-pages-47-56.pdf`, 2008.

[29] Wilson Mar. Escape Characters. `http://www.wilsonmar.com/1eschars.htm`, August 2010.

[30] MD McIlroy, EN Pinson, and BA Tague. Unix time-sharing system forward. *The Bell system technical journal*, 57(6 part 2):1902, 1978.

[31] B. Meyer. Applying design by contract. *Computer*, 25(10):51, 1992.

[32] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008.

[33] B. Peirce. *Linear associative algebra*. Van Nostrand, 1882.

[34] Avery Pennarun. Uniconf. `http://alumnit.ca/wiki/attachments/uniconf.pdf`, May 2003.

[35] Markus Raab and Patrick Sabin. Implementation of Multiple Key Databases for Shared Configuration. `ftp://www.markus-raab.org/elektra.pdf`, March 2008.

[36] D. Robbins. Common threads: Advanced filesystem implementer's guide, Part 1. *IBM Developer Works, http://www.ibm.com/developerworks/library/l-fs.html*, 2001.

[37] V. Samar. Unified login with pluggable authentication modules (PAM). In *Proceedings of the 3rd ACM conference on Computer and communications security*, page 10. ACM, 1996.

[38] J. Siméon and P. Wadler. The essence of XML. *ACM SIGPLAN Notices*, 38(1):1–13, 2003.

[39] T.A. Standish. *Data structures, algorithms and software principles in C*. Addison Wesley, 1995.

[40] B. Stroustrup. Exception safety: concepts and techniques. *Advances in exception handling techniques*, pages 60–76, 2001.

[41] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995.

[42] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. *Computational Science-ICCS 2004*, pages 440–447, 2004.

[43] Girish Welling and Maximilian Ott. Customizing idl mappings and orb protocols. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 396–414, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[44] Clifford Wolf. The craft of api design. http://www.clifford.at/papers/2008/apidesign/, September 2008.

[45] C.P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.

Index

List of Figures

List of Tables

List of Listings