TU WIEN Informatics

Mapping the Semantics of Elektra's Configuration Database to File Systems

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Bachelorstudium Software & Information Engineering

by

Alexander Firbas
Registration Number 11775819

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn. Markus Raab, BSc

Vienna, 6th September, 2021

Alexander Firbas	Markus Raab

Erklärung zur Verfassung der Arbeit

Alexander Firbas

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.

Wien, 6. September 2021

Alexander Firbas

Abstract

The need to configure software with sophistication exceeding traditional mechanisms gave rise to the software configuration management system “Elektra”.

In an effort to increase both compatibility and usability for administrators and application developers alike, by removing the need to learn Elektra-specific commands, this thesis introduces a mechanism to represent Elektra’s configuration database as a file system.

Differences and similarities between Elektra’s configuration database and traditional file systems are analyzed, yielding a design for a bidirectional mapping between the two to overcome the differences. Said mapping was implemented and became part of Elektra’s tooling.

As Elektra’s configuration database is not completely isomorphic to classical file systems, making design choices in order to bridge the gaps was imperative. Above other metrics like performance or strict adherence to the implemented protocols, maximizing usability was the primary concern.

Furthermore, possible semantics for write operations in Elektra’s cascading namespace were discussed, leading to their incorporation into Elektra’s tooling.

Contents

CHAPTER 1

Introduction

1.1 Aim & Motivation

Complex software and its unforeseeable deployments demand for mechanisms to effectively conquer the associated challenges of configuring its operation. Elektra goes beyond the usual paradigm of application specific configuration files and provides, among others, support for programmatic manipulation of configuration data through a unified `API`, automatic validation, an expandable architecture, and a suite of tools to administrate a systems configuration.

This thesis aims to further enhance Elektra's tooling by offering its core functionality as a file system. To that end, it explores under which assumptions and restrictions the key database of the feature rich library for software configuration management Elektra [teaa] is structurally isomorphic to a regular file system, and how this can be exploited to create a bidirectional representation of it by means of a FUSE file system [lt].

Because the notion of a file system is universal in computing, an application providing its services by means of such enables other applications to interface with it without the need of modification. Furthermore, system administrators and regular users alike are freed from the burden of having to learn how to use yet another tool, but can instead leverage their existing knowledge of tools and workflows.

Therefore, making Elektra available as a file system could benefit the initiative in the following ways.

First of all, for system administrators, there is no, or at least a reduced requirement to learn Elektra specifics, such as the detailed commands to query or alter information in the database. Instead, the core functionality of the database is entirely accessible by means of standard UNIX tools like `ls`, `cp`, `vi`, and so forth. All this potentially eases the learning effort needed, at least when core functionalities are of concern.

Furthermore, by analogy to the well-known virtual UNIX `/proc` file system, enhanced debugging capabilities are offered to system administrators, application developers and endusers alike.

Research Questions:

To make the goal of the following work more precise, the following two research questions have been formulated to guide this thesis:

- **RQ1:** In what ways is the data model of Elektra isomorphic to a file system, where are differences, and where have assumptions to be made?
- **RQ2:** What are possible semantics for write operations in the cascading namespace?

1.2 Methodology & Structure of the Thesis

First, a brief theoretical discussion outlining the problems at hand and related work is given. Building on that, a concrete implementation of the proposed file system, given by means of collaborative participation in free and open source ElektraInitiative development, is presented.

After describing and giving a rationale for the numerous design decisions and trade-offs taken, the potential enhancements and disadvantages are analyzed and discussed in detail. To conclude, the limitations of this work and possible further work are discussed.

CHAPTER 2

Background & Related Work

2.1 Software Configuration Management

When software becomes more complex and the users' needs more diverse, there seems to occur an increasing demand for more sophisticated means for software configuration; the role of the human in this regard is explored e.g. in [FDY18]. The focus of this thesis lies on the usability of configuration management software.

Various paradigms and philosophies have arisen aiming to satisfy this demand. A well-known method, commonly used on UNIX systems and its descendants is to make use of plain text configuration files, readily editable with text editors without the need for specific tooling. For example, on GNU/Linux systems user accounts are defined in the plain text configuration file `/etc/passwd`.

Additional demands such as a need for validation of configuration and context adapting configuration can be achieved with the help of tools like `libelektra`, on which this thesis builds.

2.2 Elektra

The following introduction to Elektra builds heavily upon the work of [Raa10] and [Raa17].

Elektra is an extensible free and open source library that provides for software configuration management by means of a global key database, denoted by `kdb`. In this database, the atomic unit of storage is that of a `Key`, storing textual or binary data, and is uniquely identified by a `namespace` chosen from a predefined set combined with a name. These names are not flat, but rather structured into a `/`-delimited sequence of name-parts, akin to the naming scheme employed in modern file systems. This enables the natural notion of a key being located *below* another key, endowing the database its hierarchical nature.

Additionally, for each `Key` a set of `MetaKeys` may be stored. A `MetaKey` is similar to a regular `Key`, but is always a member of the "meta:" namespace and is uniquely identified only in conjunction with the `Key` it is associated with.

Definition ***Core Definitions***

- *`namespace` pre-defined collection of unique identifiers*
- *`Key` atomic unit of storage in Elektra, stores textual or binary data and is identified by a namespace and an ordered sequence of textual name-parts, in notation delimited by "/"*
- *`Key` B is* ***below*** *`Key` A iff they share the same `namespace` and the sequence of name-parts of A forms a prefix of B.*
- *`MetaKey` a `Key` which is a member of the "meta:" namespace. It is always associated with a `Key`; the combination of which identifies a textual value associated with the `MetaKey`.*
- *`Context` process context, i.e. all attributes of a system process that have an effect on the operating system's behavior when handling requests (current working directory, (system) user id, the process's environment and arguments, ...)*

The operations of the database (e.g., resolving and writing keys) are only well-defined with regards to a context.

Elektra provides for the following predefined namespaces, where each one can be thought of as a root-node of its own hierarchy of keys, that differ in their context-dependant semantics:

- `system:` is used to store information that applies to the whole system and is static with regards the context. Modification requires super-user privileges.
- `user:` is used to store data belonging to the current user.
- `dir:` depends on the current working directory and allows for per-directory configuration, as is common for e.g. the configuration of webservers.
- `proc:` allows for the inspection of the process calling `Elektra` and is ephemeral, analogous to the procfs-filesystems in UNIX.
- `/` (cascading namespace): This namespace is special in that it does not have the capacity to store keys itself, but rather provides a context-dependant view on the union of all other namespaces. Any key accessed is actually resolved to some other namespace. In case multiple keys of equal name exist in different namespaces, in its simplest form, the key with the most "specific" namespace is chosen. For example, keys below `user:` mask keys below `system:`.

Note that this enumeration is not exhaustive, but some special namespaces such as `default:` are not directly relevant to this thesis.

Elektra offers numerous additional features going beyond what is needed for this thesis such as:

- automatic validation of configuration data with respect to a specification (`spec:` namespace)
- mounting of configuration files into the database

It is evident that there are some structural analogies of Elektra's database and traditional file systems. This intuitive notion will later be made precise using FUSE.

2.3 FUSE

File System in Userspace (FUSE) [lt] is a free and open source library that enables the development of file systems outside of kernel-space, i.e. in userspace.

The design of common operating systems, for example GNU/Linux, mandate filesystems be implemented either as part of, or runtime-loadable module of the kernel, executed in privileged mode. This allows for potentially increased performance, as inefficiencies with regard to process context switches are reduced.

FUSE circumvents this requirement by means of a kernel module that implements the operating system dependant interface required for filesystems, but does not handle requests in place, but rather acts as a proxy that forwards any request to some user program.

For this purpose, FUSE specifies an application programming interface that describes the operation of a generic file system. In most cases, there is a strict correspondence of this API and the set of UNIX system calls both in naming and semantics. This interface is not required to be implemented in full: for example, any information-altering operation could be omitted to provide a read-only file system.

This results in the benefits of:

- a rich programming environment and interoperability, as usually only a restricted and specialized tool set is available in kernel space
- the safety and stability advantages of user space programs: failure is constrained and may not corrupt the entire system state, access to resources can be constrained by permission systems
- ease of use in both operation and development, as there is no need to directly deal with the kernel (loading modules, recompiling the kernel, . . .)

This flexibility comes at the cost of possibly reduced performance, as explained in [VTZ17].

FUSE is widely adopted and has been integrated into the Linux kernel. Prominent examples of file systems implemented with FUSE are NTFS-3G [teab], enabling the use of the proprietary Microsoft file system NTFS on other platforms, or SSHFS [teac], leveraging an SSH-connection to mount a remote file system locally.

The question of how Elektra can be exposed as a (FUSE) file system is discussed in the following chapters.

CHAPTER 3

Design & Implementation

3.1 Configuration Database as File System

Although the established model of a file system and Elektra's configuration database are heavily inspired by the mathematical notion of a tree, there are signification deviations. First, those differences are discussed, and building on that, a proposal for a semantic of a mapping of Elektra's hierarchy into a file-system accounting for the differences, with the goal of maximizing usability, is given.

3.1.1 Representing the Key Database as a File System

As previously discussed, a query to Elektra's key database (`kdb`) is only well-defined with respect to a context, i.e. a subset of an UNIX-process context (user id, current working directory, ...).

This necessitates the question of which (process) context to choose while querying `kdb`. To select any single specific context means to restrict the view on the database to that chosen context. For example, the `user:` namespace can now only be used to query information for the particular user associated with the chosen context.

A more general approach is to avoid this decision beforehand and delegate it to the end user. The set of all active processes on a system, enumerated by their respective process identifiers ("pid") poses a feasible set of options to provide:

- The set is extensible by ad-hoc creation of new process possessing the desired attributes.
- Any active process to be configured via `kdb` is already present. This may ease the debugging process, as the precise configuration data as observed from the

application to be configured is made available without the need to modify the application at hand.

From these considerations and the structure found in the virtual `/proc` file system on Linux systems, the structure of the FUSE-file system discussed in this thesis was devised:

At the root level of the file system, for each active process on the system, a directory named after the corresponding `pid` is made available. As a convenience feature, these directories may be queried for extended file attributes (`xargs`) that describe their process context. As this first layer of the file system merely gives a representation of processes on the system, it does not support write operations.

Here, due to the behavior of the `user:` namespace, processes whose working directory is located below the mountpoint are hidden to prevent infinite recursions.

On the layer of directories directly below, i.e. directly below a path of the form `<mountpoint>/<pid>/`, the Elektra hierarchy can now be mounted both for read and write operations. For each of the predefined namespaces, a directory possessing the according name is made accessible.

How this can be accomplished in detail is discussed in the next section.

```
|-- 41
|   |-- cascading:
|   |   |-- dir_and_file_at_once
|   |   |   `-- leaf
|   |   |-- dirkey
|   |   |-- elektra
|   |   |   |-- modules
|   |   |   `-- version
|   |   |-- info
|   |   `-- person
|   |       `-- name
|   `-- user:
|       |-- dir_and_file_at_once
|       |   |-- @elektra.value
|       |   `-- leaf
|       |-- info
|       `-- person
|           `-- name
```

Figure 3.1: Truncated illustration of the discussed structure using exemplary data

3.1.2 Differences between Conventional File Systems & Elektra

The clear distinction between "files" and "directories" made in classical file systems, wherein files cannot have any descendants, and directories may only store information (beside metadata) by means of descendant files and directories and not directly, does not exist in Elektra. Rather, the only type of entity is that of a `Key`. On the set of all keys in the database, a partial order is induced by the notion of one key being below another key. This forms the basis for recursive operations whereby more than one key is affected at once.

However, this does not imply that all keys are comparable in the mathematical sense, i.e. no total order is formed. This means there is no "smallest" element, i.e. the hierarchy is not necessarily rooted in a single element. As described, each namespace induces a seperate root. This is in contrast to the model imposed by UNIX-like systems, where the whole hierarchy is unified under a single root `/`.

Furthermore, Elektra's semantics do not mandate the hierarchy to be free of "holes": The existence of the keys `system:a` and `system:a/b/c` does not imply the existence of `system:a/b`. A traditional traversal depending on direct descendants (analogous to the UNIX-`tree` command) therefore would in general fail to discover all descendant elements.

In UNIX-like systems, additional to file system attributes of predeterminded semantics (e.g. `mtime` in representing a time stamp of last modification), a common way to support arbitrary metadata is to use so-called "Extended file attributes", commonly referred to as `xattrs`. An attribute is represented via a custom name and corresponding value.

In Elektra, the canonical way to represent metadata is through `MetaKeys`. Analogous to extended file system attributes, they allow for key-value pairs to be persisted along with keys. A restriction on `MetaKeys` is that the namespace identifer `meta:` needs to be a prefix of their names. In to comparison to `xattrs`, only textual data may be stored in `MetaKeys`.

As described in the first chapter, queries on Elektra's key database are always associated with a context, without which the semantics are not well-defined. In comparison, regular file systems, except hidden files due to a user not possessing the required permissions, the structure and contexts are invariant to different users, current working directory, etc.

Further features commonly found in file systems, such as a mechanism to control access, have no corresponding representation in Elektra.

Elektra	UNIX File Systems (e.g. **ext4** [et])
one root per namespace	single unifying root
(Meta) Key entity type	files, directories, (symbolic) links ...
hierarchy can contain "holes"	connectedness is enforced
binary metadata not supported	binary metadata is supported
context dependent semantics	structure is static
maximum size of key names and contents constrained by system memory	filename length and maximum file size subject to filesystem-dependent constraints

Figure 3.2: Summarized differences

3.1.3 Bridging the Differences

Definition *File System Entity* *A file system entity here denotes anything that is addressable by a path in a file system, e.g. a regular file, a directory, a symbolic link, ...*

Mapping Keys to File System Entities

As discussed, a traditional file system necessitates the distinction of entities as either files or directories. A simple way of addressing this would be to interpret all keys for which there are no keys below them, i.e. leaf nodes as files, and all other nodes as directories. At least two problems arise:

- Under this semantic, the value of any non virtual key that has descendants is presented as a directory. Therefore, as directories do not have a value associated to them in contrast to files, the key's value is inaccessible.

  In the proposed implementation, this is solved by exposing a "virtual" file directly below the affected key with base name `®elektra.value` that exposes the contents of the key directly above. This imposes the restriction that keys may no longer possess that name.

  As an example, consider the two keys `system:/application` with value "Application" and `system:/application/version` with value "1.0". The first key will be interpreted as a directory due to the second key below, which poses as a file. The virtual key `system:/application/®elektra.value` now acts as as a symbolic link to make the contents ("Application") of the masked key available.

  This of course comes at the cost of forbidding the aforementioned special name.

- There must not be empty directories, as they would necessarily be treated as files. This is in violation of the implicit contract between a file system and common UNIX-tools such as `mkdir`. As the goal is to favor usability over other considerations, a special mechanism allowing for empty directories is proposed: Whenever a call

to the FUSE-driver demanding a directory be created is issued, this intent is persisted by setting the special `MetaKey` `meta:fuse/directory` on the newly created key. In all pertaining operations of the FUSE file system driver, this special flag takes precedence over the general case described above, therefore establishing compatibility and matching user expectations.

Virtual Directories Ensuring File System Connectedness

Definition ***Virtual Directory*** *A virtual directory denotes a directory of the FUSE file system, which is not in direct correspondence to any key.*

Restricting the predicate "below" only to direct descendants, it is consistent with Elektra's semantics for keys to exist for which no path in the induced tree of keys originating from their respective namespace exists.

In an implementation, without any special measures, these keys would still be addressable, but not discoverable, as directory listings only take into account direct ancestors. Therefore, while the naive approach is truthful to the structure of the underlying database, usability is diminished.

The reconciliation chosen in the implementation at hand is to let the file system driver make accessible the missing directories connecting inaccessible keys with the rest of the hierarchy. These directories do not have backing keys, which means that the strict correspondence between file-system entities and keys is lost. Special care needs to be taken for certain operations, e.g. renaming such a "virtual" directory results in the need to create a new key, as there is no backing key to rename.

Mapping Metadata

Due to the above described difference, to ensure metadata persisted via the FUSE file driver behaves as expected, there is need for a transparent translation of key names either amending or stripping the prefix. Otherwise, applications not using Elektra could not leverage metadata. Due to Elektra not supporting binary `MetaKeys`, an attempt to write such a key results in the operation aborting with an error.

As most keys found in a typical Elektra configuration are backed by an actual file (accessible, e.g., by the `kdb file` command), its regular attributes, such as ownership information can be mirrored to the file exposed by the FUSE-driver. Furthermore, for convenience's sake, the path of this file can be queried as an extended file attribute.

CHAPTER 4

Discussion

In the previous chapters the considerations resulting in a concrete implementation of the file system driver have been examined. The following chapter aims to put the work done into perspective and discuss obstacles faced and potential future enhancements.

4.1 Semantic Limitations

As the `proc:` namespace for a given process is both only accessible from that process and arbitrarily mutable, a faithful representation of it can in general not be given using the tool implemented as-is.

Rather, given some target process, the mounted file system provides a viewpoint of a state as it would have been prior to any alterations made by the process. Additional configuration of the `proc:` namespace, as for example provided by optional plugins operating on said namespace, has to be specified a priori and is not extracted from the target process.

A possible route for amelioration could be to make Elektra fully introspectable in a future version, yielding a direct source of truth and obsolescence for the context mocking mechanism described in this thesis. This however poses a non-trivial undertaking whose design trade-offs are yet to be discussed.

4.2 Performance

Performance was not a design goal, instead simplicity and therefore making the implementation less prone to programming faults was.

Especially the context mocking aspect reduces the possibility of some performance enhancements, as the view on the database is dependent on context.

However, the intended use case of interactive inspection and modification by an administrator, or the seldom reads usually necessary for configuring an application put this issue into perspective.

4.3 File System Permissions & Ownership

As Elektra has no inherent system of permissions and ownership, whereas this is a core aspect of a file system, there is no obvious mechanism to project permission- and ownership-related attributes bidirectionally in all cases.

Therefore, `chmod` and `chown` are currently not implemented and there also does not seem to be an intuitive way of doing so.

Furthermore, the FUSE implementation does not signal a "not supported error" in case the mentioned operations are invoked to enable compatibility with common tools like `cp`.

4.4 Platform Support

As core parts of the implementation depend upon mechanisms commonly only found on UNIX platforms, for example `/proc` virtual file system or extended file system attributes, the implementation is currently only supported on GNU/Linux distributions.

Therefore, the support for other operating system paradigms, e.g. Microsoft Windows, would require major architectural changes and is therefore out of scope of this thesis.

4.5 The Cascading Namespace

The canonical way to interact with Elektra outside of application development is by use of the command-line tool `kdb` which is shorthand for key-database.

Consider the following example illustrating an interaction directly with the `kdb` (version 0.9.4):

```
1  kdb set user:/key "Initial value"
2  #> Create a new key user:/key with string "Initial value"
3  kdb get /key
4  #> Initial value
5  kdb set /key "Modified value"
6  #> Using name system:/key
7  #> Create a new key system:/key with string "Modified value"
8  kdb get /key
9  #> Initial value
```

Figure 4.1: Shell interaction illustrating current semantics of the cascading namespace

First, we as some user, set a key to an initial value (as seen in lines 1, 2). As expected, the lookup to the cascading namespace delivers the set value (lines 3, 4).

Now, we use the same key name to modify the value (line 5). This, however, results in the unexpected outcome of creating a new key in the `system:` namespace (lines 6, 7).

When accessing the key again, illustrating this behavior, the initial (not the modified!) value is returned (lines 8, 9).

The core issue here is that key names are resolved differently depending on whether a read or write operation is performed. This violates an implicit assumption inherent in file systems that addresses always refer to the same entity.

As the described implementation only acts as a delegate, translating file system requests to Elektra and vice-versa, the semantic illustrated here is also inherent to the file system created.

As the implementation of the file system driver depends on `kdb`, any attempt to improve the situation in the FUSE implementation would only mask the problems present in the core tooling itself or introduce inconsistencies with said tooling. Therefore, the semantics of `kdb` have been altered to incorporate the results found in the following discussion.

Discussion of Possible Write-Semantics

To be able to evaluate a possible semantic, desirable properties are proposed to serve as a quality measure.

A To guarantee a property akin to the classical notions of consistency found in e.g. [DMEM15], stating that values written to some address have to be accessible under the same address afterwards, which can also be expressed as follows:

$\texttt{read}(\texttt{write}(\texttt{database\_state}, \texttt{address}, \texttt{value}), \texttt{address}) = \texttt{value}$

(With respect to some database model, where both `read` and `write`, denote a state transformation and `address` and `value` each representing any valid address/value with respect to that database.)

B Transparency. The semantics of an operation should aim to minimize mental workload and surprise.

The analysis can be conducted by evaluating two cases separately:

- Writes on keys that already exist:

  In this case, a query to the cascading namespace together with the context in which it is made references a concrete key in a well-defined manner. If property **A** is to be guaranteed, the concrete key affected cannot differ from the key identified by the query. This could be achieved by reusing the lookup semantics already

implemented in Elektra and applying them to write operations of existing keys as well.

- Writes on keys yet to be created:

  A query to a key in the cascading namespace which is not currently present cannot be unambigously mapped to any specific namespace (`system:`, `user:`, ...). A semantic thus either needs to define a mapping in this case, or disable write operations altogether.

  Disabling the write operations in this case reduces the feature set and does create the need for case distinctions on behalf of calling software, but is simple, prevents many unintended consequences, therefore arguably abiding **B** better than other choices.

  If on the other hand writes are to function in any case, different strategies may be chosen:

  – A single static choice, e.g. the most general namespace `system:` offers high predictability, but it can be argued that this contradicts the intuitive understanding of the bottom-up-approach used in privilege systems (e.g. not starting applications as root by default), or abstaining from the use of global variables in computer programs.

  – Making the choice dependent on context. For example, a sufficiently specific namespace could be selected, e.g. `dir:`. This does not suffer from the problem above, but is nonetheless an arbitrary design choice.

  In any case, since the choice is not rejected (as in the first discussed possibility), the mechanism is arbitrary to any user not intimately knowing the semantics of Elektra. Therefore, from a usability perspective, principle **B** will be violated for most users.

Regardless of the concrete semantic chosen, the use of the cascading namespace may imply uncertainty on behalf of the user/application, since otherwise a concrete key could be of interest rather than a lookup. Thus, a write operation to the cascading namespace may entail unexpected consequences, for example global configuration changes where only locale ones were desired, violating property **B**.

The current semantic implemented in `kdb` violates both principles, since, as demonstrated property **A** does not hold, and the absence of **A** already implies the absence of **B**.

Based on the preceding analysis, disabling write operations to the cascading namespace altogether is preferable over the current semantics. If, however, writes are desired, the option of restricting writes to already existing keys proves best.

Chosen & Implemented Semantics in `kdb`

For understanding the chosen implementation, the following `kdb` commands are introduced:

- `kdb set` can set the value of an individual key.
- `kdb meta-set` can set the value of a `MetaKey`.
- `kdb editor` uses the preferred editor to edit parts of the key-database.
- `kdb import` imports keys from the standard input.
- `kdb export` exports keys to the standard output.
- `kdb rm` deletes one or more keys.

Based on the discussion before, the following changes have been implemented:

- `kdb set` and `kdb meta-set`: Writes to the cascading namespace are now only allowed if the lookup succeeds. Otherwise, the operation is ambiguous and therefore aborted.
- `kdb editor` and `kdb import`: The use of the cascading namespace is now disabled entirely, since the use of those tools strongly implies that it is known exactly which keys are to be modified. Therefore, using the cascading namespace would create non-transparent behavior.
- `kdb rm`: Since no keys are created by this command, the ambiguity described above does not arise here. Whenever a (cascading) key is deleted, its identity is known.

Chosen & Implemented Semantics in the FUSE Module

As the creation of new file system entities in the cascading namespace is an ambiguous operation (with the option of removing the ambiguity by introducing a static or dynamic decision procedure for determining a namespace not being a feasible choice as outlined above), it has been disabled.

However, without this core functionality, a semi-writable file system would be introduced. This behavior is unexpected by both users and tools alike. For example, the popular text editor `vim` relies on being able to create temporary files in the directory of the file currently being edited.

Thus, it has been decided to represent the cascading namespace as a read-only (sub-)file system.

CHAPTER 5

Conclusion

After giving an introduction to Elektra and FUSE, the differences between the semantics of Elektra and common file systems were described and an implementation bridging those differences has been elaborated on and implemented for the ElektraInitiative.

For the implementation the guiding principle for any design decision was to favor compatibility with existing tools and workflows over strict adherence to Elektra's structure.

This has been achieved by creating a virtual file system which makes the whole Elektra hierarchy bidirectionally accessible for users and tools alike.

Due to Elektra and file systems being two fundamentally different concepts, the mapping does not preserve every minor detail. However, the intended use case is not affected by those shortcomings.

Furthermore, the tool provides a new convenient debugging opportunity not previously present in the Elektra project.

The core tool `kdb` of Elektra has also been enhanced to avoid non-transparent behavior with regard to write operations in the cascading namespace.

Further work enabling full introspection of Elektra could be used to overcome the limitations discussed in Section 4.1.

List of Figures

Bibliography

[DMEM15] Hendrik Decker, Francesc D. Muñoz-Escoí, and Sanjay Misra. Data consistency: Toward a terminological clarification. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Marina L. Gavrilova, Ana Maria Alves Coutinho Rocha, Carmelo Torre, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2015*, pages 206–220, Cham, 2015. Springer International Publishing.

[et] ext4 team. ext4 file system manual. https://man7.org/linux/man-pages/man5/ext4.5.html. accessed on Mar. 12, 2021.

[FDY18] Syahrul Fahmy, Aziz Deraman, and Jamaiah H. Yahaya. The role of human in software configuration management. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA 2018, page 56–60, New York, NY, USA, 2018. Association for Computing Machinery.

[lt] libfuse team. Fuse github repository. https://github.com/libfuse/libfuse. accessed on Mar. 10, 2021.

[Raa10] Markus Raab. A modular approach to configuration storage. Master's thesis, TU Wien, 2010.

[Raa17] Markus Raab. *Context-aware configuration.* PhD thesis, TU Wien, 2017.

[teaa] ElktraInitiative team. Elektrainitiative. https://www.libelektra.org/. accessed on Mar. 10, 2021.

[teab] NTFS-3G team. Ntfs-3g, sourceforge. https://sourceforge.net/projects/ntfs-3g/. accessed on Mar. 10, 2021.

[teac] SSHFS team. Sshfs github repository. https://github.com/libfuse/sshfs. accessed on Mar. 10, 2021.

[VTZ17] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX Association.