1 Elektra

1.1 Definitions

In this abridgement of the thesis[47] 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.

Global key database provides global access to all key databases of all applications.

Elektra is a library implementing access to a global key database. We improved Elektra significantly while working on this thesis as described in this abridgement.

To elektrify an application means to change the code so that it uses Elektra afterwards.

1.2 Why Elektra?

Figure 1: Elektra's Logo

Configurations can be used to change the behaviour of software for the users' needs. Because these settings stay the same across restarts of the program, they need to be stored permanently.

Nearly every system developed its own way to read preferences. Because the graphical user interface can be tweaked in many ways, the most encompassing systems emerged from this area. Some got a de facto standard for a desktop environment or an operating system. But they have a common problem: they are bound to the platform for which they were developed. Additionally, many libraries exist 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.

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 immediately useful for what it is written for: to access configuration.

1.3 Classes

The shared SYSTEM CONFIGURATION is identical for every user. It contains, for example, information about 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.

The 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 and preferred applications.

1.3.1 Key

Figure 2: Three Classes[3]

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.

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.

METADATA is data about data. The situation of Elektra 0.7[1] has now changed fundamentally by introducing arbitrary metadata for `Key` objects. The purpose of metadata is to distinguish between configuration and information about configuration[13].

1.3.2 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.

1.3.3 KDB

While objects of `Key` and `KeySet` only reside in memory, Elektra's third class `KDB` provides access to the global key database. `KDB` – an abbreviation of key database – is responsible for 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.

1.4 Concepts

1.4.1 Backend

Elektra has introduced BACKENDS to support the storage of key databases in different formats. 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.

1.4.2 Mounting

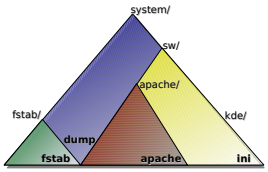

Figure 3: Mounting in Elektra

MOUNTING in Elektra[48] specifically allows us to map a part of the global key database to be handled by a different configuration storage. It 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 3, we see several such

[1] previous version of Elektra

subtrees, for example `system/sw` or `system/sw/apache`. Backends (written in bold letters) handle the configuration storage in these subtrees.

1.4.3 Abstraction

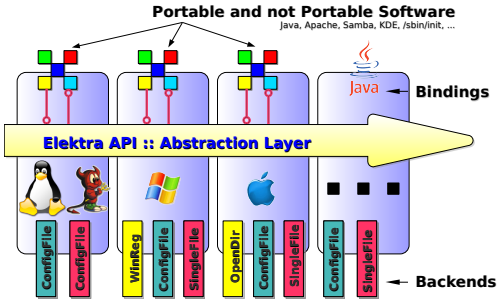

Figure 4: Elektra as Abstraction Layer, thanks to Avi Alkalay[3]

To support a global key database, a mutual agreement on some level is needed. Each elektrified application lies on top of this abstraction layer and can talk to each part of the global key database using the classes presented before.

2 Approach

2.1 Problem

It turned out that the file system semantics and the way Elektra 0.7 handles the capabilities of backends and metadata of keys are inappropriate for a key database.

2.1.1 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. Additionally, programmers cannot just take features of other backends because the code is interwoven with other code.

2.1.2 Dependences

In version 0.7, Elektra's core directly communicates with the backends. 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 to other libraries. 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. This fact made it hard to develop further features.

2.1.3 Build System

It has been impossible to turn features of backends on and off at run time. A partial and problematic solution is the former complex *build system* in Elektra 0.7 that allowed us to switch some functionality on and off at compile time. Precompiled versions of Elektra 0.7, however, either lack needed features or have considerable space and time overhead because of unnecessary features.

2.1.4 Development Time

Writing backends has revealed itself to be a time-consuming task because of the many requirements exposed to backends. Let us assume that a programmer receives a list of requirements the configuration system needs to handle. 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 needed.

2.2 New Approach

Implementing too many features in one backend is problematic. Many different aspects clutter the code, making the backends unmaintainable. 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' minds, but there was no place where it would fit in without making the system unmaintainable, unportable, complex and full of unwanted external dependences.

To solve this dilemma, we propose that MULTIPLE PLUGINS together build up a backend. Each plugin implements a single concrete requirement and it does that well[41]. The key set processed by one plugin will be passed to the next. Using this approach, the plugins provide the desired separation of concerns inside a backend.

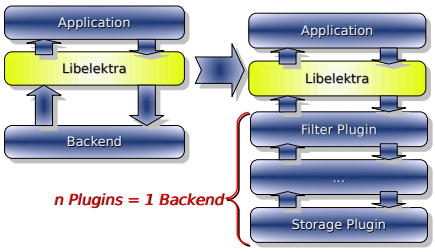

Figure 5: Introduction of Multiple Plugins

This architecture allows plugins to have external dependence. Not every plugin has the burden to be portable anymore. Maintainers can decide if a plugin should be built

for a specific platform or not. Users then 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 mounted 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.

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, how to extend it and how to write a new one. Considering that point of view, the programmer will decide to use Elektra.

2.2.1 Storage Plugin

Plugins concerned with reading and writing to permanent storage are called STORAGE PLUGINS. Their purpose is to make configuration permanent in key databases and to parse the preferences from there. Metadata can help to reconstruct configuration-specific information, like comments, ordering and line numbers.

2.2.2 Resolver Plugin

The resolver plugin has the responsibility for non-portable tasks like resolving the configuration's file name and overwriting this file in an atomic way. It guarantees that the configuration is only updated if needed and detects conflicts.

The idea of extracting all 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 method of resolving to be adapted differently to fit the user's needs even better.

2.2.3 Contract

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 in a way 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. The contract contains well-defined CLAUSES, has no hidden clauses[42] and is described using a key set.

The CONTRACT CHECKER[2] revises contracts of plugins during the mounting of backends. It can refuse to add a plugin to the backend because of a conflict or a constraint. As long as not all contracts are satisfied the contract checker denies the mounting and waits for more plugins to be attached.

[2] implemented in the kdb commandline tool

2.2.4 Ordering

Multiple plugins open many doors in which way they can be arranged in a backend. Elektra now uses three arrays of plugins. The plugins to get and to set configuration are separated because the order of execution sometimes differs. Additionally, the resolver plugin requires a third list to do a proper rollback when the writing fails. The order of plugins inside these three arrays is controlled by the contracts.

3 Implementation

3.1 Architecture

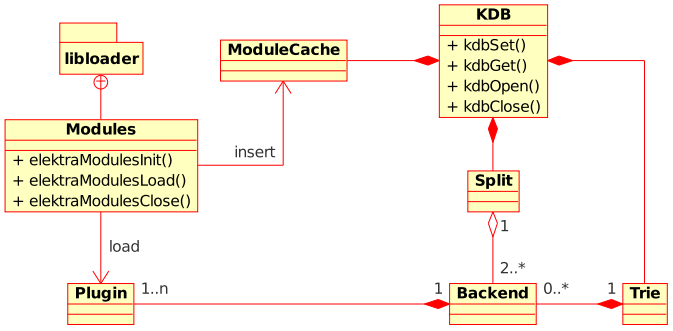

Figure 6: Architecture

We changed the internal architecture from the ground up to support multiple plugins. Elektra now provides a small internal API to load modules independently from the operating system. This API also hides the fact that modules must be loaded dynamically if they are not available statically. From these modules *plugins* can be created that are gathered together in the *backend* data structure. We see that the concepts introduced earlier have a representation in a data structure.

Those backends can be mounted anywhere in the key hierarchy. The mapping of key names to a specific backend is done with the data structure *trie*. The *split* object holds all keys in regard to the backend they belong to.

3.2 Mount Point Configuration

One important aspect of a configuration library is the out-of-the-box experience. A so-called DEFAULT BACKEND is responsible in the case that nothing was configured so far. 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. Applications and administrators can mount specific backends. Each of these

backends are built up by a number of plugins. On opening the global key database, the system BOOTSTRAPS itself, starting with the default backend.

3.3 Error Handling

It is sometimes unavoidable that ERRORS or other problems occur that ultimately have an impact on the user. Elektra now gathers all information in these situations. All `KDB` methods take a `Key` object as a parameter. This key is also passed to every plugin. The idea is to add the error and warning information as metadata to this key. This approach provides flexibility, because a key can hold a potentially unlimited number of metadata. So the library always informs the user about what has happened, but does not print or log anything itself.

3.3.1 Error Specification

The error specification in Elektra is written in colon-separated entries of a simple text file. Each entry has a unique identifier and general information about the error. 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"[30].

3.3.2 Exceptions

Elektra was designed so that both plugins and applications can be written in languages that provide exceptions as shown in Figure 7. 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 appropriate metadata describing the error.

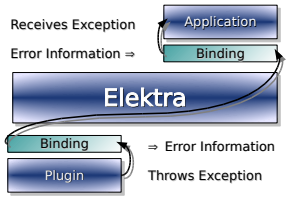

Figure 7: Exception Flow

3.4 Algorithm

3.4.1 kdbSet

`kdbSet()` stores a key set permanently. Robust and reliable behaviour is the most important issue for `kdbSet()`. `kdbSet()` *guarantees* the following properties:

1. Additional in-memory comparisons are preferred to suboptimal storage access. Modifications to permanent storage are only made when the respective configuration was changed.

2. When errors occur, every plugin gets a chance to roll back its changes. If every plugin does this correctly, nothing is changed in the key database. Plugins developed during the thesis meet this requirement.

3. When no error occurs, the whole `KeySet` is written to permanent storage.

The plugins can fail for a variety of reasons within `kdbSet()`. The most frequent occurrences are CONFLICTS. A conflict means that between 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()` to be up to date with the key database. Afterwards it is up to the application to decide which configuration to use or how to merge it together.

3.4.2 kdbGet

`kdbGet()` is responsible for retrieving configuration. Retrieving configuration is 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 remaining task is to check if the configuration is up to date using the resolver plugin, to query all requested backends for their configuration and then merge everything.

4 Plugins

Plugins implement specific features regarding configuration. Figure 8 presents an overview of the plugins we developed in this thesis.

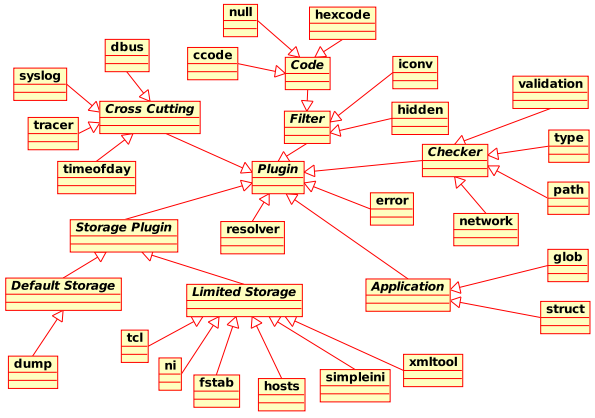

Figure 8: Overview of Plugins

Every plugin provides a full contract to give information how it will work together with other plugins. Most parts of the contract are obligatory. Plugins cannot be loaded

without this information.

Every plugin can have a PLUGIN CONFIGURATION given during mounting. For plugins providing the same feature the plugin configuration should be interchangeable, providing the advantage that the plugins are a drop-in replacement for each other.

4.1 Filter

FILTER PLUGINS process keys and their values in both directions. In one direction they undo what they do in the other direction. They can be chained together.

4.1.1 Character Reduction and Conversion

To store a large range of different characters in a sink that allows only a subset of characters to occur there, a plugin must *reduce* the range of characters in a reversible way. Contracts of storage plugins define which characters must disappear so that the storage plugin can be used without problems. The contract checker deduces a plugin configuration for the filter plugin defining which characters should be escaped.

Another filter plugin converts values between different encodings. On the one hand, it can be used to support applications that do not work with the encoding of the key database. On the other hand, it can be used to support key databases with different encodings.

4.2 Pluggable Checker

Elektra works completely without integrated type checking on keys. Instead it provides PLUGGABLE CHECKERS as plugins. They are executed before the storage plugin writes out the configuration and make sure that the configuration is valid, consistent and complete. Pluggable checkers assure that problematic configuration is never passed to the storage plugin. To provide maximum flexibility, the applying and checking of types is separated. This establishes a *two-phase type checking*.

4.2.1 Apply Metadata

In a first phase, concrete dynamic type information is applied as metadata to keys by matching key names with *glob* expressions or with *recursive structures*. For some storage plugins missing keys are as fatal as not validated keys – it would not be possible to write a valid configuration file. So in this step additional structure checks can take place. For example, the *struct plugin* verifies if a specific child, sibling or parent `Key` is present.

4.2.2 Check Plugins

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

A common and successful type system happens to be CORBA. The TYPE CHECK PLUGIN supports all basic CORBA types. Additionally, new types can be created by unification of existing types.

The VALIDATION PLUGIN works as a powerful tool to check strings using regular expressions. They provide a way to reduce the set of allowed values for a key value. We also implemented plugins for specific purposes, like network addresses or if a path exists or not.

5 Evaluation

5.1 Configuration Libraries

Dozens of other libraries exist, that can be used for storage plugins. We investigated these parsers and decided to use one of them for a case study. The implementation of this proof-of-concept plugin took slightly more than *one hour*. Arbitrary strings work out-of-the-box without any filter plugin and the plugin also gracefully handles all kinds of syntactic errors found in configuration files.

5.2 Benchmark

In the current situation, additional plugins have only a fraction of the overall run time costs. Plugins with side effects are no problem in terms of time penalty. The overhead of plugins that iterate over all keys is also insignificant. We conclude that it presents no problem to split up all cross-cutting concerns in different plugins.

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 benchmark. As desired, the actual writing and reading consumes by far the largest part of the needed run time.

5.3 Modularity

We implemented 4 plugins[3] for notification and logging and 5 more plugins related to filtering. Additionally, 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 our 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.

[3] All plugins implemented during the thesis are shown in Figure 8.

[4] The two plugins that apply the metadata will help them.

References

[1] D. Abrahams. Exception-safety in generic components. *Generic Programming*, pages 69–79, 2000.

[2] J. Adamek and F. Plasil. Erroneous architecture is a relative concept. In *Software Engineering and Applications (SEA) conference*, pages 715–720. Citeseer.

[3] A. Alkalay. Linux registry. Talk at KDE Community World Summit, August 2004.

[4] C. Amsüss. private conversation.

[5] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *ACM SIGPLAN Notices*, 41(10):74, 2006.

[6] 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.

[7] Waldo Bastian. XDG Base Directory Specification. `http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html`, June 2010.

[8] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™). `http://www.yaml.org/spec/`, October 2009.

[9] Jon Louis Bentley. *Writing Efficient Programs*. Prentice Hall, first edition, 1982.

[10] 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.

[11] 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.

[12] Frederick P. Brooks. *The mythical man-month: essays on software engineering*.

[13] EB Bruce and WG Daniel. Metadata standards and Metadata Registries: An Overview. In *The International Conference on Establishment Surveys II. Buffalo*. Citeseer, 2000.

[14] M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49(1-3):1–46, 2003.

[15] 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.

[16] M. Burgess et al. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3):309–402, 1995.

[17] 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.

[18] SC Crawley and KR Duddy. Improving Type-Safety in CORBA. In *Proceedings of IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*.

[19] 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.

[20] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 2008.

[21] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. *ACM SIGPLAN Notices*, 39(1):122, 2004.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Professional Computing Series. Addison Wesley, 1995.

[23] C. Garion and L. van der Torre. Design By Contract. *Coordination, organization, institutions and norms in agent systems I*, July 2005.

[24] 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.

[25] Object Management Group. Omg idl syntax and semantics.

[26] P. Gühring. private conversation.

[27] Carsten Haitzler. Eet library documentation. `http://docs.enlightenment.org/api/eet/html/`, 2008.

[28] Helmut Herold. *C-Kompaktreferenz*. Addison-Wesley, first edition, 2002.

[29] I.M. Holland. Specifying reusable components using contracts. In *ECOOP'92 European Conference on Object-Oriented Programming*, page 287. Springer, 1992.

[30] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[31] S. Josefsson. The base16, base32, and base64 data encodings, 2003.

[32] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, second edition, 1988.

[33] Kerninghan and Plauger. *The Elements of Programming Style*.

[34] M.F. Krafft. *The Debian system: concepts and techniques*. Open Source Press, 2005.

[35] M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76.

[36] 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.

[37] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, first edition, 2000.

[38] R. Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[39] David Lutterkort. Augeas - a configuration API. `http://www.kernel.org/doc/ols/2008/ols2008v2-pages-47-56.pdf`, 2008.

[40] Wilson Mar. Escape Characters. `http://www.wilsonmar.com/1eschars.htm`, August 2010.

[41] MD McIlroy, EN Pinson, and BA Tague. Unix time-sharing system forward. *The Bell system technical journal*, 57(6 part 2):1902, 1978.

[42] B. Meyer. Applying design by contract. *Computer*, 25(10):51, 1992.

[43] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum, Akad. Verl., fourth edition, 2002.

[44] 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.

[45] B. Peirce. *Linear associative algebra*. Van Nostrand, 1882.

[46] Avery Pennarun. Uniconf. `http://alumnit.ca/wiki/attachments/uniconf.pdf`, May 2003.

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

[48] Markus Raab and Patrick Sabin. Implementation of Multiple Key Databases for Shared Configuration. `ftp://www.markus-raab.org/elektra.pdf`, March 2008.

[49] D. Robbins. Common threads: Advanced filesystem implementer's guide, Part 1. *IBM Developer Works, http://www.ibm.com/developerworks/library/l-fs.html*, 2001.

[50] 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.

[51] J. Siméon and P. Wadler. The essence of XML. *ACM SIGPLAN Notices*, 38(1):1–13, 2003.

[52] T.A. Standish. *Data structures, algorithms and software principles in C.* Addison Wesley, 1995.

[53] B. Stroustrup. Exception safety: concepts and techniques. *Advances in exception handling techniques*, pages 60–76, 2001.

[54] Bjarne Stroustrup. *The design and evolution of C++.* ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995.

[55] Gary V. Vaughn, Ben Ellison, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool.* New Riders, first edition, 2000.

[56] 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.

[57] 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.

[58] E. Wohlstadter, S. Jackson, and P. Devanbu. Design and implementation of distributed crosscutting features with DADO. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 706–707, 2004.

[59] Clifford Wolf. The craft of api design. `http://www.clifford.at/papers/2008/apidesign/`, September 2008.

[60] 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.