Sharing Software Configuration via Specified Links and Transformation Rules

Markus Raab
markus.raab@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

Abstract

Configuration files are the dominant tool for local configuration management today. Up to now, applications cannot access any configuration file of their system, because they lack the knowledge where the configuration files reside, which syntax they use, and how a value is interpreted correctly. As a result, software systems are often poorly integrated. In this paper, we propose a specification for configuration files to mitigate these issues. Developers specify links and transformation rules to share configuration items between applications. We implemented tools and a library that integrate existing configuration files as well as a C/C++ code generator that makes sure that newly written applications are consistent with their specification. Our approach bridges across configuration file standards. It integrates configuration files we do not have control over. In a case study we demonstrate that our approach saves time, when unmodified applications need to be integrated into a coherent system. Additionally, we show that the run-time overhead of these links does not have significant impact on applications.

1 Introduction

Only a few factors determine how well-integrated an application, with respect to a software system, is: logging, external interfaces, user interfaces (such as dialogues), and user interaction (such as shortcuts and menus). In modern software these aspects are configurable. We only need to configure the software in a way that the system feels as if made from one piece. Currently, such endeavor is cumbersome in a heterogeneous system. Specific technologies only provide solutions within their respective field. Thus usually many technologies are involved, and a naïve approach often fails.

We faced this issue, next to many domain-specific ones, during an one-year project. The staff of the software engineering project varied between 3 and 5 software developers, who wrote about 50.000 lines of C and C++. The aim of the project was to engineer a platform for integrating different software applications from multiple customers. The platform enabled the modification of more than 200 configuration items which affect the behavior and features of the platform and the integrated applications.

Due to complexity of that specific system, caused by the domain-specific issues, we present our approach by the means of another issue instead: Nearly every graphical user interface (GUI) provides a shortcut for quitting the application. Also, nearly every application provides a way to use a different shortcut to change the default (that is often Ctrl+Q). Currently, no generic way exists that one application uses the shortcut of another application.

Submission to: 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS), Pörtschach, Austria, Oct-2015

We propose the following solution to give our applications access to configuration items of other applications: First, during installation of applications, we register their configuration files with information about their syntax. During startup of our application, we parse all necessary configuration files into key/value pairs with properties. A specification describes links and transformations between these abstract configuration items. Finally, we map the configuration items to variables of the programming language the application uses.

In the present paper, we will explain how a specification is used to drive the last two steps. In this specification, the developer defines which configuration items of other applications are used and how values are transformed. The specification is independent from a concrete programming language and extensible, e.g. for an editor:

```
[/our_editor/quit]
fallback/#0=/kde/kate/ActionProp/Default/file_quit
fallback/#1=/vim/map/:qa<CR>
fallback/#2=/emacs/keyboard-escape-quit
```

The first and the only required line of the specification defines the configuration item. We will call this unique id *key name*. The other lines of the section are *properties* of this configuration item. In this example, we introduced one property, called FALLBACK. Using this property we establish a link to another key name. It tells the system that, whenever the configuration item itself was not found, the value should be used from a fallback configuration item instead. We see that we add an array of such fallbacks, using the syntax #<*number*> for indexing.

The novelty of the approach is that it is transparent for any external tool in which order keys will be searched for. The specification is present at runtime and will be automatically enforced by a library. Additionally, code generation makes sure that applications use the configuration items in a type-safe way.

Our paper will answer the question: "Which properties in a specification are needed to share configuration items?". This question is significant because it enables the developer to reuse configuration items in less time than building ad-hoc solutions for every single integration needed.

The paper is structured as follows: In Section 2 we explain the details of our approach and in Section 3 we describe Elektra, a framework that implements our approach. In Section 4 we evaluate and benchmark Elektra, then in Section 5 we compare our approach with other work and finally we conclude our paper in Section 6.

2 Elektra

2.1 Technology

Configuration files have a countless number of syntactic differences. E.g. lenses [1] are one way to describe syntax of configuration files. The repository of Augeas [7] (only covers some parts of Linux configuration files) already contains 181 lenses. Semi-structured data, e.g. JSON, YAML, XML and self-describing data [16], e.g. S-expressions and also JSON [3], are very popular for configuration.

To handle this diversity, as a first step, we transform such configuration files to an abstract syntax tree (AST). We use an AST that consists only of key/value pairs with properties. Using this abstraction we get rid of specialties in syntax. We use plugins as described in [9] that parse the configuration files and yield an AST. After the transformation to an AST, the nodes refer to key names and the properties store details that are necessary to reconstruct the configuration file. Let us consider a JSON file as example:

```
{
    "boolean_key": true
}
```

Elektra's JSON plugin will transform this file to an AST. If we serialize the AST to the syntax we already used for the previous example we would get the following output:

```
[/boolean_key]
value=true
type=boolean
```

The example illustrates that self-describing data already has properties even without a specification. In a specification we directly link to `/boolean_key` regardless if a specification was written for it. Nothing novel so far, but using this technique we get an abstraction over the concrete syntax of configuration files. As a specialty, our approach handles specifications the same way as configuration files.

2.2 Resolve Sources

The location of the configuration's file name differs between different OS and distributions of the same OS. In our approach, the configuration file names from all applications are registered globally at installation time when we know the file name and its syntax. We use further configuration files to store this information. For every configuration file we add an OS-dependent plugin, called *resolver*, to handle the dynamic properties of the file name resolving. Depending on the OS context, resolvers will yield different file names.

Our approach introduces the abstraction that we will call *key database*. The key database is a tiny middleware between the plugins and the applications. Its main responsibility is the splitting and merging of the AST: plugins always get their part of the AST. The key database bootstraps itself whenever it is opened. It is library-based, that means the bootstrap process happens during the start of every application:

1. First the key database reads from a hard coded configuration file to know where the other configuration files are and which plugins (resolver, syntax of configuration file and others) should be used for each of them.

2. The resolvers determine the full configuration file names with information from the OS.

3. With the information from the previous steps, the key database builds up a data structure. In the data structure we lookup key names without any knowledge of the file name (and its syntax).

For example, consider that an application registers a JSON configuration file `app.j`, with the content as displayed above, in the key database at `/myapplication`. Then the full filename is the concatenation of a directory and `app.j` and the key name is `/myapplication/boolean_key`.

2.3 Namespace

Another dimension of configuration items is their *namespace*. Elektra supports following namespaces related to configuration files:

spec if the configuration file contains the specification.

dir if the configuration file is in a special directory (e.g. `.htaccess` of the apache web server).

user if the configuration file is in the user's home directory.

system if the configuration file is located system wide (e.g. below `/etc`).

In our approach, applications lookup all keys using the method `lookup` of a library. It has two arguments: the complete configuration `conf` (AST with all key/value pairs) and a key `key` without a namespace. No namespace is encoded in the application's source code. Instead, keys in different namespaces are considered for every lookup. The algorithm is straight-forward:

```
lookup(conf, key)
{
    s = lookupByKey(conf, spec / key);
    if (!s) return lookupBySpec(conf, s);

    ret = lookupByKey(conf, dir / key);
    if (ret) return ret;

    ret = lookupByKey(conf, user / key);
    if (ret) return ret;

    ret = lookupByKey(conf, system / key);
    if (ret) return ret;
    return 0;
}
```

First we search for the key, that contains the specification by using the `spec` namespace. If found, the internal method `lookupBySpec` is invoked with it. Otherwise, we do a cascading search of all namespaces. The operator `/` specifies that `key` is in the given namespace. The specification is only configuration represented by keys and defines how the lookup of keys work.

An abstraction of the configuration has been established. It enables us to uniquely identify configuration items. We do not have to care about the path to configuration files anymore.

2.4 Links

We already introduced the property array FALLBACK. It specifies which configuration items should be used as fallback when the configuration item itself was not found. The property array OVERRIDE complements the linking functionality. If this property is available, the configuration items linked will be preferred to the item itself. The property array NAMESPACE defines which namespaces should be considered and in which order. Finally, the property DEFAULT completes the linking functionality. It will be used if every configuration item mentioned was not found.

Given the AST `conf` and the key with properties `key` we define the search order by the following algorithm:

```
lookupBySpec(conf, key)
{
    for (number: 0 .. length(key, override)-1)
    {
        m = lookupByProperty(key, "override/#<number>");
        k = lookup(conf, m, withoutDefault);
        if (k) return k;
    }

    if (lookupByProperty(key, namespace))
    {
        for (number: 0 .. length(key, namespace)-1)
        {
            m = lookupByProperty(key, "namespace/#<number>");
            k = lookupByKey(conf, m / key);
            if (k) return k;
        }
    }
    else // if no property namespace exists
    {
        k = lookup(conf, key, withoutKey);
        if (k) return k;
    }

    for (number: 0 .. length(key, fallback)-1)
    {
        m = lookupByProperty(key, "fallback/#<number>");
        k = lookup(conf, m, true, withoutDefault);
        if (k) return k;
    }
    return lookupByProperty(key, "default");
}
```

The method `lookupBySpec` iterates over the three property arrays FALLBACK, NAMESPACE, and OVERRIDE. We see that the algorithm works recursively, but with special options for recursive invocations. The code shown here is not cluttered with those branches for clarity. If the property NAMESPACE is not specified (else branch), we use the cascading lookup as defined previously in the method `lookup`, but obviously do not search for the same key in `spec` again. The expression `m / key` means that key is in the namespace as stated by `m` and `"#<number>"` is the syntax for indexing. If neither an override, the key itself, nor any fallback was found, the algorithm returns the value as specified in the property DEFAULT.

For example, suppose we have no other configuration than the specification:

```
[/our_editor/quit]
namespace/#0=system
fallback/#0=/vim/quit
default="Ctrl+Q"

[/vim/quit]
namespace/#0=user
default=":q"
```

Then a call to `lookup` with `/our_editor/quit` as key will:

1. first lookup the key `/our_editor/quit` in the namespace `spec` successfully and call `lookupBySpec` with this key,
2. then skip override (because no property OVERRIDE is present),
3. then fail in searching for the key in the `system` namespace (because no configuration file is present),
4. then lookup the key `/vim/quit` in `spec` successfully and call `lookupBySpec` with this key,
5. skip override again,

6. then fail in searching for the key `/vim/quit` in the `user` namespace and

7. finally use the default value `Ctrl+Q`.

2.5 Value Transformation

The override/fallback mapping covers only the rare situation that the provider and consumer of the configuration item happens to interpret the same bit pattern in exactly the same way. In general, we need a unidirectional mapping of values from one bit pattern to another one if we want to reuse a configuration item. The straight forward way is to use a map:

```
[/our_editor/shortcut/quit]
transform=/kate/quit
transform/map/Ctrl+A=CTRL+A
transform/map/Ctrl+B=CTRL+B
transform/map/Ctrl+B=CTRL+C
... (23 more)
```

The property TRANSFORM is similar to override/fallback. On keys with this property, however, the key itself does not exist in a file. Instead its value is the result of a transformation specified by one of the properties TRANSFORM/TYPE, where "type" describes the type of the transformation. In the example, the transformation type is `map`. Such a mapping is practical for situations where an enumeration of all values is straight-forward, but cumbersome for others. Thus we use existing programming languages for sophisticated transformations:

```
[/our_editor/shortcut/quit]
transform=/kate/quit
transform/haskell=map toUpper
```

Which is much shorter than the previous example but has the same behaviour for valid input. In general, the transform code snippets must take one argument and return the transformed value. Obviously this specification language now is expressive enough for any transformation of values between applications, but we lack semantics when such a transformation should not happen or fails.

2.6 Skip Transformation

Sometimes the transformation is not possible or avoided on purpose:

- If the input value is not within the domain.
- If a runtime error (e.g. failed memory allocation) occurs.
- If someone decides that in the specific case it is better not to use the configuration item.

To support these semantics we use, depending on the programming language, either exceptions, errors, optional values or omit the return value. In such situations the configuration item will not be present for the application. The behavior is identical to a not-found key name.

For example, a typical request is that the application should only be adapted for an OS or desktop when the application is executed within the specific environment:

```
[/our_editor/shortcut/quit]
default=:qa!
transform=/kate/quit
transform/python=if kde_running():
    return value.upper()
```

In this example `our_editor` will use the quit shortcut from kate (which is part of KDE) only if KDE is the currently running desktop. Otherwise the key `/our_editor/shortcut/quit` will not exist.

2.7 Types and Code Generation

In our approach, every parameter has a type. If no property TYPE is given, an abstract top type will be assumed. This behavior guarantees that configuration items without the property TYPE still type-check safely [6].

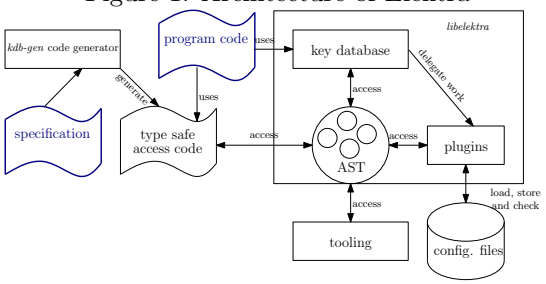
Types give a better understanding how the parameter is used and provide a foundation to check if a concrete configuration is valid. In our approach we generate code for easy access to the configuration items in the same

way as [12]. Then types become a necessity because only with types the compiler ensures that the usage of the configuration values is correct.

We conclude that every type used in the property TYPE must have an exact counterpart in the type systems of the target languages. CORBA IDL already defines a consistent mapping for many programming language and can be used with our approach.

3 Implementation

Figure 1: Architecture of Elektra



We call our framework Elektra. It consists of following parts:

1. A library (called *libelektra*) that transforms configuration files to an AST. It contains an implementation of the key database as discussed before.
2. We use a code generator (called *kdb-gen*) to ensure that the usage of the variables match the specification in statically typed programming languages. kdb-gen generates both C and C++ front ends and different documentation artifacts by using different templates.
3. The front end is a type safe access code that is generated by the specification.

The user of our approach has to implement only two parts (bold, blue boxes in Figure 1):

1. The specification with the properties FALLBACK, OVERRIDE, etc. (as discussed in Section 2).
2. The program code of the application consuming the configuration files has to be adapted to use Elektra (also called elektrify). The programmer must make sure that the program code uses the type safe access code and not directly configuration files nor environment variables so that the links will work.

We see in Figure 1 that the program code, that uses other configuration files, needs to use the key database in order to enable the consumption of other configuration files. Because of the plugins, however, other applications providing configuration files do not need to be modified. Elektra can use their configuration files directly.

3.1 Front end

The generated classes (C++) and functions (C) provide type safe and context aware access to configuration items [12] [10]. This layer is very thin. It is only responsible for looking up the configuration value and lexical casting the resulting string to a native data variable. It is straight forward to provide support for additional programming languages.

If desired, the front end can have a built-in copy of the specification. Using this technique, the application starts up without any configuration files.

In order to support the properties OVERRIDE and FALLBACK, we enhanced the lookup of values in the AST. Using the property TRANSFORM it will be transformed with the given rules. In rather static languages, e.g. C++, the transformation rule will be included in the generated code. In our C++ implementation we use policy-based design. Using policies programmers can modify the behavior of the front end. Additionally, it supports a separation of hand-written and generated parts.

E.g. the specification,

```
[/myapp/shortcut/quit_myapp]
default=CTRL+Q
type=string
transform=/kate/quit
transform/cpp=
    std::transform(value.begin(), value.end(),
    value.begin(), ::toupper);
    return value
```

will generate the following policy code (shortened for the sake of brevity):

```
class QuitMyappGetPolicy
{
public: typedef std::string type;
static type get(kdb::KeySet &ks, kdb::Key const& ) {
    type value = "CTRL+Q";
    kdb::Key found = ks.lookup("/kate/quit", 0);
    if (found) {
        value = found.get<std::string>();
        std::transform(value.begin(), value.end(),
        value.begin(), ::toupper);
        return value;
    }
    return value;
} };
```

The policy classes `GetPolicy` are responsible to lookup a configuration item in the case of a cache miss of the type safe access code [12]. The AST is denoted as `kdb::KeySet`. We see that the default value and the transformation are hard coded.

The generic tools we saw in Figure 1 cannot rely on code generation. Instead they read the specification dynamically. The implementation for both cases (code generated and dynamically) is a straightforward implementation of the pseudo-code as given in Section 2.

3.2 Key Database

The key database is a very thin layer that delegates all the work to Elektra's plugins. The plugins are responsible to resolve the configuration file name, as described in Section 2 and to parse and write configuration files. Different parsers are used for different formats:

1. JSON, INI and XML libraries handle semi-structured data.
2. Elektra's augeas [7] plugin handles most Linux configuration files, e.g. sshd, security/limits.conf.
3. Hand-written parsers handle other INI dialects and other configuration files, e.g. hosts, fstab.

The only responsibility left to the key database is to bootstrap the system and to pass the correct parts of the AST to the correct plugins.

4 Evaluation

We implemented the described artefacts and plugins in Elektra. Based on the experiences of the one-year project mentioned in Section 1 and later measurements, we will discuss the development time. The rest of the evaluation, however, is based on programs we wrote specifically for that purpose.

4.1 Development time

Measuring the overall time in a larger project was unfortunately not possible for us because using our approach individual configuration related tasks are done in a few minutes, which is difficult to track. We will only discuss development time of individual tasks. Additionally, we will show representative code to give a better understanding of necessary effort. The time measurement is based on the commits of our version control system.

The basic setup to use Elektra only consists of the following straight-forward lines:

```
#include <editor.hpp>

int main()
{
    using namespace kdb;
    KDB kdb;
    KeySet conf;
    Context c;
    Parameters par(conf,c);
    std::cout << par.myapp.shortcut.quitMyapp << "\n";
}
```

In the first line we include the code generated from the specification. After creating a handle to the key database, an AST for the configuration (called `KeySet`) and a `Context` [12], we finally create an instance of the generated class `Parameters`. Then we directly access configuration variables with the key name. The only difference to the specification is the usage of `.` instead of `/` to denote the key name.

The needed time to add one parameter is noteworthy: In only two minutes we were able to add a new configuration item that was fully documented and used in the application. The needed time does not change significantly if a small number of links exist. To add transformation keys, however, can take much longer, especially, if the original configuration item is not documented properly. To add command-line parsing ability for all parameters in a small application, we only needed six minutes (the template for generating the code already exists within Elektra).

We developed a large application (50.000 lines of C and C++) based on a specification describing configuration with Elektra. In that project we used several properties not described in this paper. Nevertheless, we experienced an improved development time and especially debugging time compared to another project with rather traditional means of configuration access: the direct use of a data structure along with a XML Schema Definition (XSD) validating configuration files written in XML. While one configuration change in our approach needed only a single change in the specification, up to more than 10 places needed to be modified in the project using XSD.

Adding new plugins to support new configuration file standards, unsurprisingly, takes significantly longer. Small tasks, e.g. integrating existing configuration parsers or writing a template similar to existing ones, were done by us within a day. For example, the INI plugin `ni`, that parses the syntax of the examples in this paper, has 158 lines of C code and was developed within a day (10:41:54 - 16:22:01). To support parsing properties following code in Elektra was needed:

```
Ni_node mcur = NULL;
while ((mcur = Ni_GetNextChild(current, mcur)) != NULL)
{
        keySetMeta (k, Ni_GetName(mcur, NULL), Ni_GetValue (mcur, NULL));
}
```

Explanations of the API and other details about development of the plugins are given in [9].

Adding new templates to support new programming languages or properties often takes a similar amount of time. For example, to add long option parsing support took less than one hour (10:53:30 - 11:33:04). The whole template that parses command line options has 261 lines of code. Templates are written in cheetah [15] with many utilities provided by Elektra, e.g. the fallback mechanism for C and C++ is included easily in new templates.

Here is a part of the template that implements the property FALLBACK:

```
@set $fa = $support.fallback(info)
@if len(fa) > 0
@for $f in $fa
    found = ks.lookup("$f", 0);
    if (found) {
        value = found.get<$support.typeof(info)>();
        $support.transform($info, $fa.index($f));
    }
@end for
@end if
```

Note that in cheetah template code (lines starting with `@`) is interwoven with C code. The array `fa` contains the properties FALLBACK. For every property FALLBACK the code for a `lookup` invocation and the value transformation is generated. The generated code from that template is shown in the earlier example `class QuitMyappGetPolicy`.

4.2 Links within a single specification

We implemented a word counting utility in C that internally relies on the links as described in this approach. The `wc` tool has the following features: it counts lines, words, chars, bytes and the length of the longest line. Without any option the tool will print lines, words and chars. With any configuration item given, it will only print the requested counters. Such interdependencies within configuration item are easily represented with the following links:

```
[/sw/wc/show/max_line_length]
type=boolean
default=false
opt=L
opt/long=max_line_length

[/sw/wc/show/no_default_args]
type=boolean
default=false
override/#0=/sw/wc/show/lines
override/#1=/sw/wc/show/words
override/#2=/sw/wc/show/chars
override/#3=/sw/wc/show/bytes
override/#4=/sw/wc/show/max_line_length
```

Using links avoids an implementation of the override/fallback logic in the application and has following advantages:

Changeability: Even when multiple applications use the specification item `no_default_args` we only have a single place to change the logic.

Independence: The links are available as data and can be used in any programming language or technology.

Traceability: The links exist explicitly and are traceable without program analysis.

Extensibility: Both the configuration items and the links can be extended with any desired property, e.g. `opt` in the example above permits us to generate commandline parameter parsing code that accepts `-L` and `--max_line_length`.

Performance: In a previous paper [12] we show that access of the configuration item has no overhead compared to access of native C++ variables.

4.3 Links between applications

We implemented a minimalist editor with configurable shortcuts in C++. The tool sloccount measured 3,106 total physical source lines of code. In this case study, we confirmed that other editors do not need any modification to share their configuration.

Elektra supports a large number of configuration file standards, including those mentioned in this paper (JSON, XML), those supported by Augeas (e.g. apache, ssh), some basic Linux configuration files (e.g. hosts) and various INI formats. To support a large number of standards is especially important for software integration between applications.

For vim and Emacs, however, none of these parser worked because their configuration file contains code. For such situations we implemented a plugin, called regexstore, that uses regular expressions ignoring all non-matches. In contrast to lenses regexstore only takes care of the parts of the configuration files we are interested in and does not understand the rest of the file. Given regexstore, we are able to integrate even vim and Emacs configuration files. Up to now, we did not find any configuration file we could not integrate and because plugins are written in universal programming languages we argue that any way to store configuration can be integrated into Elektra.

4.4 Benchmark Setup

We conducted the benchmarks on a hp® EliteBook 8570w using the CPU Intel® Core™ i7-3740QM @ 2.70GHz. The operating system is GNU/Linux Debian Wheezy 7.5. We used the gcc compiler Debian 4.7.2-5. We measured the time using `gettimeofday`. We executed each benchmark eleven times for the box-plots. The benchmark setup is identical to our previous benchmark [12].

We implemented a static and a dynamic variant of our algorithm:

In the dynamic variant keys contain the properties. That means that `lookupByProperty` of our algorithm is a dynamic lookup in a data structure. The application reads the specification in configuration files at runtime. In this variant, we have to lookup every property, even if they are not available.

In the static variant the override and fallback mechanism is compiled in the application. In this variant the code generator adds code for every link as specified. We already saw examples of this variant (the class and the template of `QuitMyappGetPolicy`). Only recursive links (not used in our benchmarks) are resolved in the same way as in the dynamic variant. That means also in this variant the specification needs to be read at runtime. Obviously, this approach has no overhead when no links are used.

4.5 Lookup Time

In the first benchmark we will measure the time needed for the lookup in the data structure. We generated 10 variables that have 0 to 9 properties OVERRIDE, e.g. the first and second configuration items:

```
[/benchmark/#0]
default=33
type=unsigned_long

[/benchmark/#1]
default=benchmark
type=unsigned_long
override/#0=/benchmark/override/#0
```

Using these configuration items we make 200,000 lookups. If we would only access the variable, we would not have any performance overhead (with any number of links) as described in [12]. In order to actually measure the lookup time, we synchronize the cache in every iteration. In Figure 2 we see the time grows linearly for a larger number of OVERRIDE-properties.

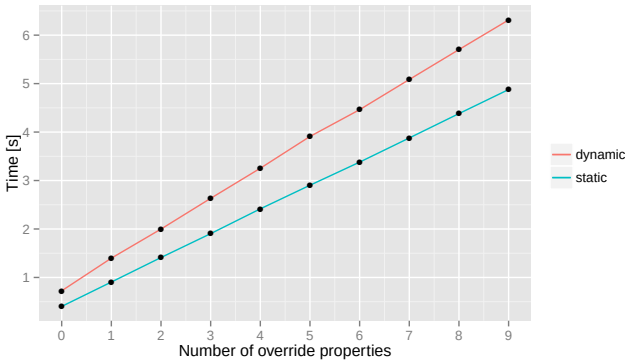


Figure 2: Lookup time in static and dynamic variant with linear scale for 200,000 lookups.

For the second benchmark measuring the dynamic variant we use the same specification and the same number of iterations. We see that we have a larger overhead because of the additional lookups required for every property. In the dynamic variant even not specified properties cause overhead.

Next to the constant difference of factor 1.8 the overhead also grows 22% faster in the dynamic variant because of additional property lookups needed. Another drawback of the dynamic variant is, that it does not allow compiled code to be used. Instead transformations need to be interpreted, again adding overhead. We conclude the static variant is faster, but it has a severe problem: applications directly using the key database

without code generation (e.g. tooling) sees not transformed, i.e. incorrect, values. To answer the question if dynamic or static should be preferred, we have to answer the question if the lookup overhead is significant in an application.

4.6 Overall Runtime Overhead

Considering a full application, in our approach following steps are necessary:

kdbOpen: The bootstrapping as described in Section 2.

kdbGet: The reading of (other) configuration files.

process: The functionality of the application.

Our setup is as follows. We benchmark the word counting tool already described earlier with the LaTeX file document of this paper (32KB size). We have a small configuration file that is read during startup. We profiled the application using Callgrind 3.7.0. Only two configuration files were involved: the bootstrapping configuration file (read during `kdbOpen`) and one configuration file for the application (read during `kdbGet`).

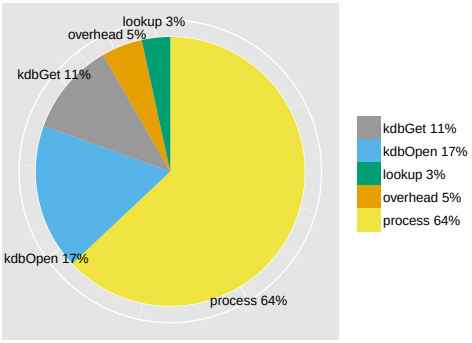


Figure 3: Full application using a static variant

In Figure 3 we see that the processing of the file dominates with 64%. What obviously matters is the size of the configuration files. The transformation to an AST (using the ni plugin [9]) unfortunately is much slower than just counting the words (by a factor of 12 on files with the same size). Because of this `kdbOpen()` takes 17% and `kdbGet()` needs 11% of the overall cycles even with a small configuration files.

Without the links in the specification the number of lookups are reduced to 9 (from otherwise 27) cascading lookups (see the algorithm `lookup` in Section 2). Based on this, we know that the overhead of the links is only 5% in this application.

4.7 Threats to Validity

The main problem of our evaluation is that the applications used are rather small. The development times are only taken from a single project and need to be validated by additional case studies for external validity.

5 Related Work

Configuration management (CM) tools, e.g. [2] solve the problem stated in this paper differently. They copy each value to every place as needed. This approach, however, fails to work, when the user modifies configuration locally. Most computer systems today, however, are configured locally: mobile devices, laptops, tablets and non-business desktop systems. CM tools will not work when no distribution system between the nodes exists.

Reuse of software components facilitates reuse of software configuration. Modern desktop environments have a tight integration based on that principle. In this approach every software component is responsible for its own configuration. These components enable programs to modify settings in one place and taking effect for the whole system. Zdun [17] even argues that the concern "behavioral composition and configuration" should be treated as a first-class entity. While this approach has many advantages, its application is often difficult (e.g. programming language barriers) or even not possible (e.g. because of licensing issues).

Using lenses [1] as implementations allows us to quickly cover a lot of different configuration file formats, but lenses seem to fail [7] when we need complete abstraction from the concrete syntax of the configuration file. Type-safe lenses are only based on regular expressions and the resulting AST is very similar to the configuration file syntax and its internal order. In our approach we do not have such a limitation because we can transform keys to the desired structure.

Ontologies are used for sharing data. Gruber [5] describes general design criteria so that every specification has a minimal ontological commitment, e.g. we should tolerate that one date is "1993" and the other "March

1993". Enforcing a canonical representation would be an encoding bias. In our approach we solve this problem by transformations. Gruber also introduces references to uniquely identify publications. In our approach key names cover this concept.

XPointer [4] permits us to create links within XML documents. The main difference is that they are heavily dependent on XML technology. So they cannot be used for configuration files, where XML is not dominant. XPointer is not able to achieve the same as the properties OVERRIDE, FALLBACK and DEFAULT we described in this paper. XInclude [8] is also tied to XML technology, but different from XPointer, XInclude has an element fallback that is similar to the property DEFAULT as described in this paper. For the other properties we described, no equivalents exist.

Context-oriented programming [12] [10] is supplementing the approach described in this paper. It allows applications to be aware in which context they are, but does not enable them to be aware of other applications.

We are positive that possible extensions of our approach also improve safety [11].

A different type of configuration links are explained in [13] and formally developed in [14]. Similarities to our approaches are advantages regarding specification evolution, and the potential usage for internal fallbacks as we discussed in the evaluation. They are different because they:

- only refer from target to source specifications while Elektra supports references within specifications and directly to configuration items,
- are evaluated during generation of configuration and therefore cannot be as flexible as Elektra's run-time evaluation,
- only provide propositional logic to determine selection while Elektra facilitates programming languages to determine if transformation should be skipped, and
- seem not to support transformation rules which further limits their use.

6 Conclusion and Further Work

This paper describes a novel way to establish links between configuration items. We use data as a specification to define abstraction and to describe access on data. This specification alone suffices to share configuration, and even saves time in the process. As the specification is just data we can easily extend the approach to other programming languages. Nevertheless, the specification is powerful enough to allow applications to use any configuration item of any configuration file. Our approach avoids the introduction of a new configuration file format. Instead, existing configuration file formats are integrated using a global abstract syntax tree.

In the benchmark we compared a static and a dynamic implementation of our approach. The lookup time is not significant in either way, and the dynamic implementation has the advantage that it also works with tooling that does not use the code generator. So we prefer the dynamic variant.

To answer our research question: Three properties, namely FALLBACK, OVERRIDE and TRANSFORM are needed to share configuration between applications. Additionally, these links are also useful to implement fallback and override logic within a single program. In general, using the specification is superior to hard coding logic in the application because:

- External tools use the specification and thus present the same configuration as the application.
- The specification is enforced for every application accessing the configuration.
- It is transparent to the administrator which configuration values are to be preferred.

Currently, types need to be annotated manually for every key. As further work we plan to reconstruct the types using the links. By reconstructing the types of the transformation rules (e.g. when Haskell and ML are used), we will even infer transformed types. Furthermore, other transformations are waiting to be explored and evaluated. Last, but not least, we want to evaluate techniques which allow our approach to be used for unmodified binaries, e.g. by preloading `getenv()`.

Our contributions are:

- Describing an approach in which applications are aware of other applications' configurations, leveraging easy application integration.

- Implementing our approach for popular semi-structured data formats and Linux configuration files, downloadable from `http://www.libelektra.org`.
- Comparing a static and dynamic variant of the implementation.
- Providing experimental validation using a case study of significant complexity and evaluate performance.

References

[1] 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, NY, USA, 2008. ACM.

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

[3] Douglas Crockford. Json: The fat-free alternative to xml. In *Proceedings of XML*, volume 2006, 2006.

[4] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. Xpointer framework. *W3c recommendation*, 25, 2003.

[5] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5–6):907 – 928, 1995.

[6] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures*, 35(1):48–62, 2009.

[7] David Lutterkort. Augeas–a configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[8] Jonathan Marsh, David Orchard, and Daniel Veillard. Xml inclusions (xinclude) version 1.0. *W3C Working Draft*, 10, 2006.

[9] Markus Raab. A modular approach to configuration storage. *Master's thesis, Vienna University of Technology*, 2010.

[10] Markus Raab. Global and thread-local activation of contextual program execution environments. In *Proceedings of the IEEE 18th International Symposium on Real-Time Distributed Computing Workshops (ISORCW/SEUS)*, pages 34–41, April 2015.

[11] Markus Raab. Safe management of software configuration. In *Proceedings of the CAiSE'2015 Doctoral Consortium*, pages 74–82, urn:nbn:de:0074-1415-4, 2015. http://ceur-ws.org/Vol-1415/.

[12] Markus Raab and Franz Puntigam. Program execution environments as contextual values. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, pages 8:1–8:6, NY, USA, 2014. ACM.

[13] Mark-Oliver Reiser. *Core Concepts of the Compositional Variability Management Framework (CVM): A Practitioner's Guide*. TU, Professoren der Fak. IV, 2009.

[14] Mark-Oliver Reiser. *Managing complex variability in automotive software product lines: subscoping and configuration links*. Südwestdt. Verlag für Hochschulschriften, 2009.

[15] Tavis Rudd, Mike Orr, and Ian Bicking. Cheetah-the python-powered template engine. In *10th International Python Conference.—2002*, 2007.

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

[17] Uwe Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems & Structures*, 32(1):56 – 82, 2006.